V1.3

MCP Module Technical Documentation

Multi-Domain Agent System Integration

Version: 1.3

Date: August 26, 2025

Target Domains: Finance, Productivity, Education, Sports, Software Development

---

1. Executive Summary

This document outlines the technical architecture and implementation strategy for the Model Context Protocol (MCP) module that will power our multi-domain agent system. The MCP module serves as a dynamic bridge between the base agent system (which generates agent configurations) and actual tool implementations, enabling seamless integration with external APIs, databases, and services.

Input: The MCP module takes as input the dynamically generated base agent JSON file (e.g., `BA_op.json`) which varies based on user prompts. This JSON contains agent definitions with abstract tool requirements that need to be matched to actual MCP server capabilities at runtime.

Key Design Principles:

- **Fully Dynamic**: No hardcoded configurations, tool names, or server definitions
- **Input Agnostic**: Processes ANY base agent JSON generated from user prompts
- **Semantic Matching**: Uses Mistral 7B to understand meaning, not exact names
- **Discovery-Based**: Finds whatever MCP servers are available at runtime
- **Domain-Aware**: Optimized for 5 specific domains but adaptable to any

- **LLM-Powered**: Uses Mistral 7B for intelligent tool matching
- **Protocol Compliant**: Adheres to MCP specification v2025-03-26
- **Scalable**: Supports multiple concurrent agent-server connections

---

2. System Architecture

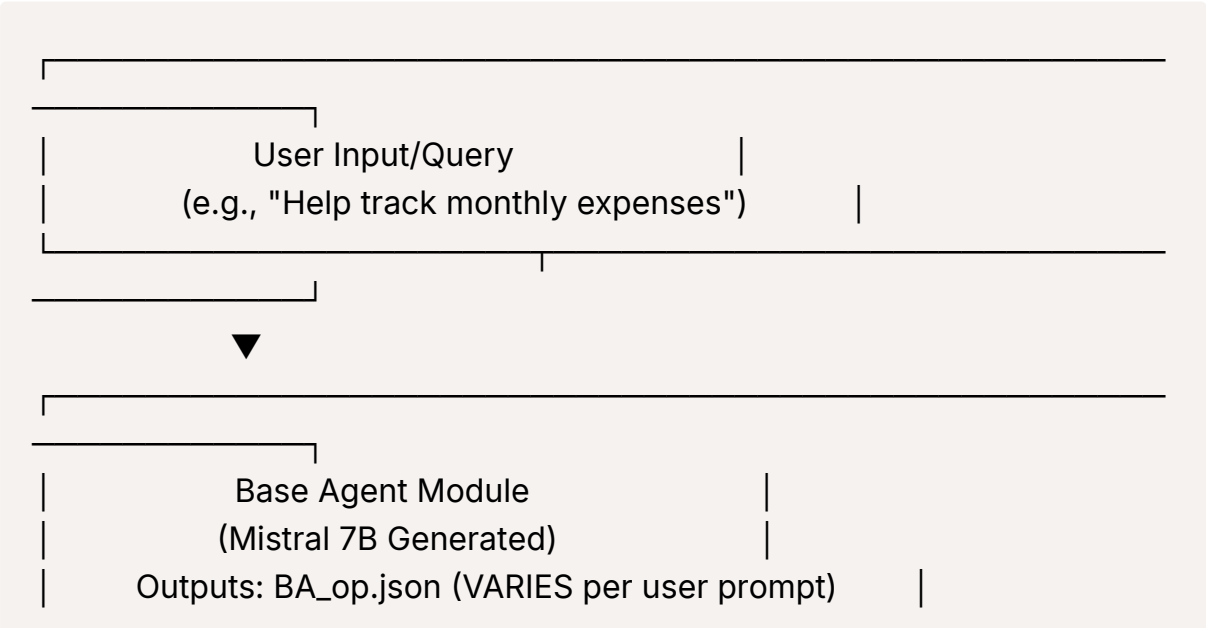
2.1 MCP Primitives Overview

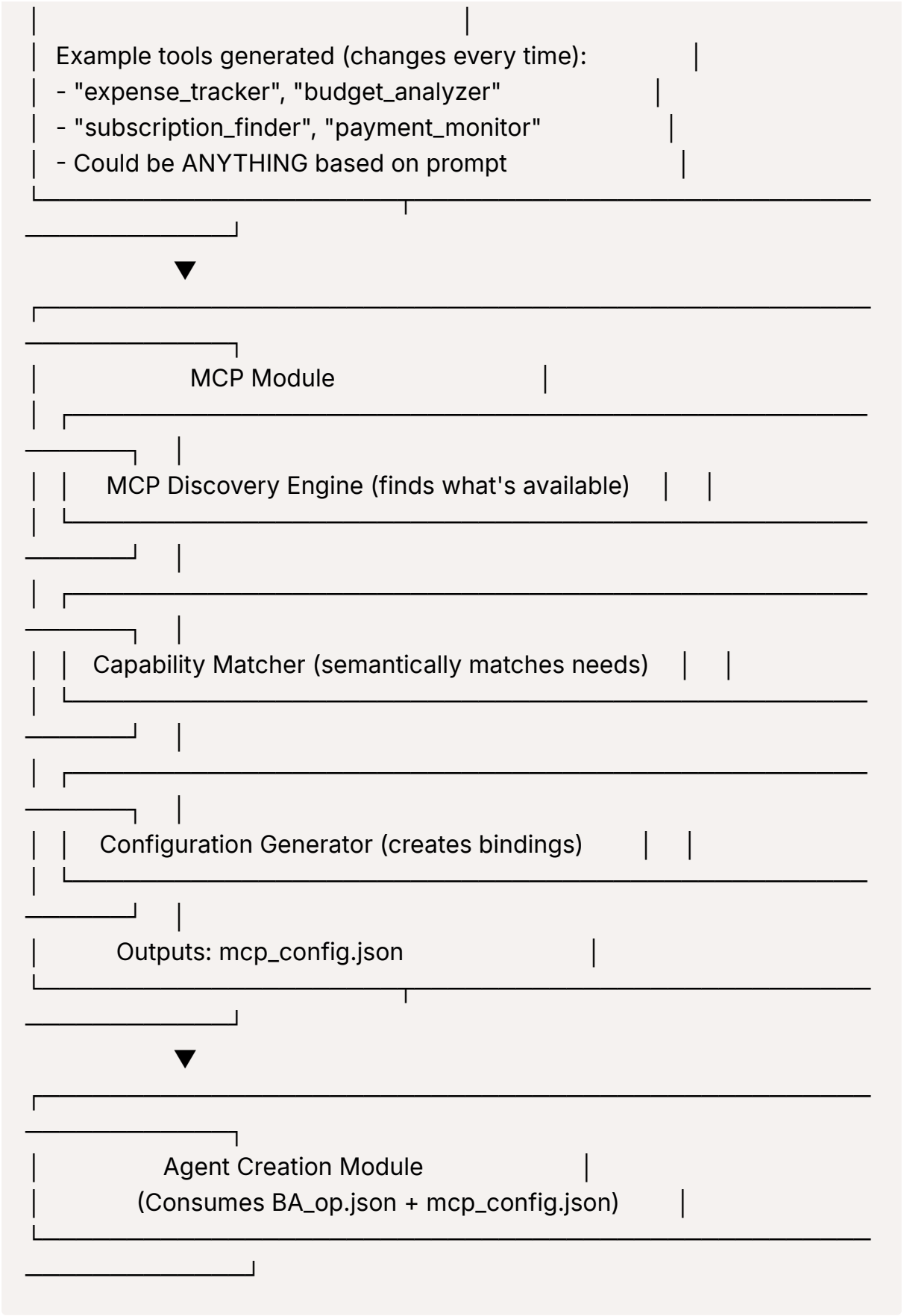
Based on the official MCP specification and Python SDK, MCP servers expose three core primitives:

Primitive	Control	Description	Example Use in Our System
Tools	Model-controlled	Functions the LLM decides to invoke for actions	`bank_statement_parser`, `budget_calculator`, `code_optimizer`
Resources	Application-controlled	Read-only data sources for context	`market_data://`, `email_templates://`, `config://settings`
Prompts	User-controlled	Reusable interaction templates	Code review prompts, debug assistants, learning assessments

The MCP module must discover and match ALL THREE types dynamically based on what the base agent requires.

2.1 High-Level Architecture





2.2 Component Details

2.2.1 MCP Discovery Engine

Purpose: Dynamically discover available MCP servers and their capabilities

Components:

- **Server Scanner**: Scans for MCP servers using multiple methods
- **Capability Introspector**: Queries servers for their capabilities
- **Registry Manager**: Maintains a live registry of available servers
- **Health Monitor**: Tracks server availability and performance

2.2.2 Capability Matcher

Purpose: Intelligently match agent tool requirements to MCP server capabilities

Components:

- **Semantic Analyzer**: Uses Mistral 7B to understand tool requirements
- **Scoring Engine**: Ranks server capabilities against requirements
- **Fallback Resolver**: Identifies alternative servers for resilience

2.2.3 Configuration Generator

Purpose: Generate complete MCP configuration for agent creation

Components:

- **Protocol Negotiator**: Determines optimal transport (stdio/HTTP/WebSocket)
- **Authentication Manager**: Handles OAuth 2.1 and API key management
- **Schema Validator**: Ensures configuration compliance with MCP spec

---

3. Python SDK Implementation Patterns

3.1 Creating MCP Servers with FastMCP

Based on the official Python SDK, here's how MCP servers expose the three primitives:

```
from mcp.server.fastmcp import FastMCP
from mcp.server.fastmcp.prompts import base
```

```
# Create MCP server instance
mcp = FastMCP("finance-analyzer")

# 1. TOOLS - Model-controlled functions for actions
@mcp.tool()
def analyze_bank_statement(file_path: str, date_range: dict) → dict:
    """Parse and categorize bank transactions"""
    # Tool implementation
    return {
        "total_expenses": 2500.00,
        "categories": {"food": 800, "transport": 300},
        "subscriptions": ["netflix", "spotify"]
    }

@mcp.tool()
def calculate_budget(income: float, expense_categories: list) → dict:
    """Create budget based on income/expenses"""
    return {"budget_plan": {...}, "savings_potential": 500}

# 2. RESOURCES - Application-controlled read-only data
@mcp.resource("finance://market-data/{symbol}")
def get_market_data(symbol: str) → str:
    """Provide real-time market data"""
    return f"Current price for {symbol}: $150.25"

@mcp.resource("finance://tax-rules/{year}")
def get_tax_rules(year: str) → str:
    """Get tax regulations for a specific year"""
    return f"Tax rules for {year}: ..."

# 3. PROMPTS - User-controlled interaction templates
@mcp.prompt()
def financial_advice(user_profile: str, goal: str) → str:
    """Generate personalized financial advice prompt"""
    return f"Based on profile: {user_profile}, provide advice for: {goal}"

@mcp.prompt()
def budget_review(current_budget: str) → list[base.Message]:
```

```
    """Interactive budget review prompt"""
    return [
        base.UserMessage("Review this budget:"),
        base.UserMessage(current_budget),
        base.AssistantMessage("I'll analyze your budget for optimization oppo
rtunities")
    ]
```

3.2 Server Capability Declaration

When discovered, MCP servers declare their capabilities:

```
# Server response to discovery query
{
    "capabilities": {
        "tools": {
            "listChanged": True  # Supports dynamic tool updates
        },
        "resources": {
            "subscribe": True,    # Supports resource subscriptions
            "listChanged": True   # Supports dynamic resource updates
        },
        "prompts": {
            "listChanged": True   # Supports dynamic prompt updates
        }
    }
}
```

4. MCP Server Architecture Examples

IMPORTANT NOTE: The server configurations shown below are **EXAMPLES ONLY** of what might be discovered at runtime. These servers are **NOT predefined or hardcoded**. The actual servers discovered will have different names, capabilities, and configurations based on what's running in the environment. The MCP module will semantically match whatever tools are required (from the base agent JSON) to whatever servers are discovered, regardless of naming conventions.

3.1 Finance Domain MCP Servers (Example Discoveries)

Server: `finance-core-mcp`


```
{
  "name": "finance-core-mcp",
  "version": "1.0.0",
  "transport": "stdio",
  "capabilities": {
    "tools": [
      {
        "name": "analyze_bank_statement",
        "description": "Parse and categorize bank transactions",
        "inputSchema": {
          "type": "object",
          "properties": {
            "file_path": {"type": "string"},
            "date_range": {"type": "object"}
          }
        }
      },
      {
        "name": "calculate_budget",
        "description": "Create budget based on income/expenses",
        "inputSchema": {
          "type": "object",
          "properties": {
            "income": {"type": "number"},
            "expense_categories": {"type": "array"}
          }
        }
      },
      {
        "name": "portfolio_analysis",
        "description": "Analyze investment portfolio",
        "inputSchema": {
          "type": "object",
          "properties": {
```

```
          "holdings": {"type": "array"},
          "market_data": {"type": "boolean"}
        }
      }
    },
    {
      "name": "subscription_tracker",
      "description": "Identify and analyze recurring payments",
      "inputSchema": {
        "type": "object",
        "properties": {
          "transactions": {"type": "array"}
        }
      }
    }
  ],
  "resources": [
    {
      "name": "market_data",
      "description": "Real-time market data feed",
      "uri": "finance://market-data/*"
    },
    {
      "name": "tax_rules",
      "description": "Current tax regulations",
      "uri": "finance://tax-rules/*"
    }
  ]
 }
}
```

Server: finance-apis-mcp

```
{
 "name": "finance-apis-mcp",
 "version": "1.0.0",
 "transport": "http",
 "capabilities": {
```

```
    "tools": [
     {
      "name": "plaid_connect",
      "description": "Connect to bank via Plaid API",
      "annotations": {"auth_required": true}
     },
     {
      "name": "alpha_vantage_query",
      "description": "Get stock market data"
     },
     {
      "name": "stripe_payments",
      "description": "Process payment operations"
     }
    ]
   }
}
```

3.2 Productivity Domain MCP Servers

Server: `productivity-gmail-mcp`

```
{
  "name": "productivity-gmail-mcp",
  "version": "1.0.0",
  "transport": "http",
  "capabilities": {
    "tools": [
     {
      "name": "gmail_summarize",
      "description": "Summarize lengthy emails using AI",
      "inputSchema": {
       "type": "object",
       "properties": {
        "email_id": {"type": "string"},
        "summary_length": {"type": "string", "enum": ["brief", "detailed"]}
       }
      }
```

```
    },
    {
      "name": "smart_reply",
      "description": "Generate contextual email responses"
    },
    {
      "name": "email_categorize",
      "description": "Auto-categorize emails"
    }
   ],
   "resources": [
    {
      "name": "email_templates",
      "uri": "gmail://templates/*"
    }
   ]
  }
}
```

Server: productivity-calendar-mcp

```
{
  "name": "productivity-calendar-mcp",
  "version": "1.0.0",
  "transport": "stdio",
  "capabilities": {
   "tools": [
    {
      "name": "schedule_meeting",
      "description": "Intelligently schedule meetings"
    },
    {
      "name": "generate_meeting_notes",
      "description": "Auto-generate meeting notes from recording"
    },
    {
      "name": "conflict_resolver",
      "description": "Resolve calendar conflicts"
```

```
    }
   ]
  }
}
```

Server: productivity-task-mcp

```
{
  "name": "productivity-task-mcp",
  "version": "1.0.0",
  "transport": "websocket",
  "capabilities": {
    "tools": [
      {
        "name": "thought_to_task",
        "description": "Convert spoken ideas to actionable tasks",
        "inputSchema": {
          "type": "object",
          "properties": {
            "audio_input": {"type": "string", "format": "base64"},
            "context": {"type": "string"}
          }
        }
      },
      {
        "name": "dependency_mapper",
        "description": "Auto-generate task dependencies"
      }
    ]
  }
}
```

3.3 Education Domain MCP Servers

Server: education-research-mcp

```
{
  "name": "education-research-mcp",
  "version": "1.0.0",
  "transport": "http",
  "capabilities": {
    "tools": [
      {
        "name": "paper_summarizer",
        "description": "Extract key insights from academic papers",
        "annotations": {"model_required": "mistral-7b"}
      },
      {
        "name": "citation_generator",
        "description": "Generate proper citations"
      },
      {
        "name": "literature_review",
        "description": "Conduct systematic literature review"
      }
    ],
    "resources": [
      {
        "name": "arxiv_papers",
        "uri": "education://arxiv/*"
      },
      {
        "name": "pubmed_database",
        "uri": "education://pubmed/*"
      }
    ]
  }
}
```


Server: education-learning-mcp


```
{
  "name": "education-learning-mcp",
```

```
  "version": "1.0.0",
  "transport": "stdio",
  "capabilities": {
    "tools": [
      {
        "name": "career_guidance",
        "description": "AI-based career path recommendation"
      },
      {
        "name": "skill_gap_analyzer",
        "description": "Identify skill gaps and suggest resources"
      },
      {
        "name": "adaptive_study_planner",
        "description": "Create personalized study schedules"
      }
    ],
    "prompts": [
      {
        "name": "learning_style_assessment",
        "description": "Assess student's learning style"
      }
    ]
  }
}
```

3.4 Sports Domain MCP Servers

Server: `sports-performance-mcp`

```
{
  "name": "sports-performance-mcp",
  "version": "1.0.0",
  "transport": "websocket",
  "capabilities": {
    "tools": [
      {
        "name": "biomechanics_analyzer",
```

```
      "description": "Analyze athlete movement and form",
      "inputSchema": {
       "type": "object",
       "properties": {
        "video_data": {"type": "string", "format": "base64"},
        "sport_type": {"type": "string"}
       }
      }
     },
     {
      "name": "training_optimizer",
      "description": "Optimize training regimen"
     },
     {
      "name": "injury_risk_assessment",
      "description": "Predict injury risks"
     }
    ]
  }
}
```

Server: sports-analytics-mcp

```
{
  "name": "sports-analytics-mcp",
  "version": "1.0.0",
  "transport": "http",
  "capabilities": {
    "tools": [
     {
      "name": "match_predictor",
      "description": "Predict match outcomes using ML"
     },
     {
      "name": "player_statistics",
      "description": "Comprehensive player stats analysis"
     },
     {
```

```
      "name": "strategy_recommender",
      "description": "Recommend game strategies"
    }
   ],
   "resources": [
    {
     "name": "historical_data",
     "uri": "sports://historical/*"
    }
   ]
  }
}
```

3.5 Software Development Domain MCP Servers

Server: `dev-cicd-mcp`

```
{
 "name": "dev-cicd-mcp",
 "version": "1.0.0",
 "transport": "http",
 "capabilities": {
  "tools": [
   {
    "name": "pipeline_optimizer",
    "description": "Optimize CI/CD pipelines"
   },
   {
    "name": "build_analyzer",
    "description": "Analyze build performance"
   },
   {
    "name": "deployment_automator",
    "description": "Automate deployment processes"
   }
  ]
```

```
  }
}
```

Server: dev-code-mcp

```
{
  "name": "dev-code-mcp",
  "version": "1.0.0",
  "transport": "stdio",
  "capabilities": {
    "tools": [
      {
        "name": "code_optimizer",
        "description": "Suggest code optimizations",
        "annotations": {"read_only": true}
      },
      {
        "name": "code_search",
        "description": "AI-powered code search"
      },
      {
        "name": "refactor_assistant",
        "description": "Automated refactoring suggestions"
      }
    ]
  }
}
```

Server: dev-documentation-mcp

```
{
  "name": "dev-documentation-mcp",
  "version": "1.0.0",
  "transport": "http",
  "capabilities": {
    "tools": [
      {
```

```
      "name": "api_doc_generator",
      "description": "Generate API documentation"
    },
    {
      "name": "test_case_generator",
      "description": "Auto-generate test cases"
    },
    {
      "name": "swagger_builder",
      "description": "Build OpenAPI/Swagger specs"
    }
   ]
  }
}
```

5. Dynamic Discovery Implementation

5.1 Handling Variable Base Agent JSON Input with All Primitives

The MCP module processes base agent JSON that may require any combination of tools, resources, and prompts. The base agent dynamically generates requirements for all three primitive types based on the user's prompt:

```
// Example base agent output (BA_op.json) - varies with each prompt
{
  "agents": [
    {
      "agent_id": "agent_1",
      "agent_name": "Financial Analyst",
      "tools": [
        {
          "name": "bank_statement_parser",
          "purpose": "Parse and analyze bank statements",
          "auth_required": false
        },
        {
          "name": "expense_tracker",
```

```
        "purpose": "Track and categorize expenses",
        "auth_required": false
      }
    ],
    "resources": [
      {
        "name": "transaction_history",
        "purpose": "Access historical transaction data",
        "uri_pattern": "finance://transactions/*"
      },
      {
        "name": "market_data_feed",
        "purpose": "Real-time market information",
        "uri_pattern": "finance://market/*"
      }
    ],
    "prompts": [
      {
        "name": "budget_advice",
        "purpose": "Interactive budget consultation",
        "parameters": ["income", "expenses", "goals"]
      },
      {
        "name": "investment_guidance",
        "purpose": "Personalized investment advice",
        "parameters": ["risk_tolerance", "time_horizon"]
      }
    ]
  }
 ]
}
```

The MCP module must discover servers that can provide all three types of capabilities and semantically match them to the agent's requirements.

5.2 Discovery Strategy for All Primitives

The MCP module discovers servers and ALL their capabilities:

The MCP module implements a three-tier discovery mechanism:

Tier 1: Environment-Based Discovery

```
# Pseudo-code for environment discovery
def discover_from_environment():
    # Check environment variables
    mcp_servers = os.environ.get('MCP_SERVERS', '').split(',')

    # Check configuration file
    config_path = Path.home() / '.mcp' / 'servers.json'
    if config_path.exists():
        servers.extend(json.load(config_path))

    return servers
```

Tier 2: Network Discovery

```
# Pseudo-code for network discovery
def discover_from_network():
    # Scan local ports (3000-4000 range)
    for port in range(3000, 4001):
        try:
            response = http_get(f"http://localhost:{port}/mcp/describe")
            if response.status == 200:
                servers.append(response.json())
        except:
            continue

    # Query discovery service
    discovery_response = http_get("http://localhost:5000/mcp/registry")
    servers.extend(discovery_response.json())

    return servers
```

Tier 3: Process Discovery

```
# Pseudo-code for process discovery
def discover_from_processes():
```

```
    # Find running MCP server processes
    for proc in psutil.process_iter(['pid', 'name', 'cmdline']):
        if 'mcp-server' in proc.info['cmdline']:
            # Extract connection info from process
            server_info = extract_server_info(proc)
            servers.append(server_info)

    return servers
```

5.3 Complete Dynamic Discovery Flow for All Primitives

How the MCP Module Processes Tools, Resources, and Prompts

Step 1: Parse Base Agent JSON (Variable Input)

```
def process_base_agent_json(ba_json):
    """
    Process ANY base agent JSON - extracts all three primitive types
    """
    requirements = {
        'tools': [],
        'resources': [],
        'prompts': []
    }

    for agent in ba_json['agents']:
        # Extract tools (for actions)
        for tool in agent.get('tools', []):
            requirements['tools'].append({
                'agent_id': agent['agent_id'],
                'name': tool['name'],
                'purpose': tool['purpose']
            })

        # Extract resources (for context/data)
        for resource in agent.get('resources', []):
            requirements['resources'].append({
                'agent_id': agent['agent_id'],
```

```
            'name': resource['name'],
            'uri_pattern': resource.get('uri_pattern')
        })

    # Extract prompts (for interactions)
    for prompt in agent.get('prompts', []):
        requirements['prompts'].append({
            'agent_id': agent['agent_id'],
            'name': prompt['name'],
            'purpose': prompt['purpose']
        })

return requirements
```

Step 2: Discover MCP Servers and Query All Capabilities

```
async def discover_and_introspect():
    """
    Discover servers and their tools, resources, and prompts
    """
    discovered_servers = discover_all_mcp_servers()

    for server in discovered_servers:
        # Query for all three primitive types
        capabilities = await query_server_capabilities(server)

        server['discovered_capabilities'] = {
            'tools': capabilities.get('tools', []),      # Functions for actions
            'resources': capabilities.get('resources', []), # Data sources
            'prompts': capabilities.get('prompts', [])    # Interaction templates
        }

    return discovered_servers
```

Step 3: Semantic Matching Using Mistral 7B

```
def match_all_primitives(requirements, discovered_servers):
    """
```

```
    Match all three primitive types semantically
    """
    matches = {
        'tools': {},
        'resources': {},
        'prompts': {}
    }

    # Match Tools (functions for actions)
    for tool_req in requirements['tools']:
        prompt = f"""
        Required tool: {tool_req['name']} - {tool_req['purpose']}

        Available discovered tools:
        {[s['tools'] for s in discovered_servers]}

        Match based on what the tool DOES, not its name.
        Example: "bank_statement_parser" ≈ "pdf_analyzer", "document_read
er"
        """
        matches['tools'][tool_req['name']] = mistral_llm.match(prompt)

    # Match Resources (data sources)
    for resource_req in requirements['resources']:
        prompt = f"""
        Required resource: {resource_req['name']}
        URI pattern: {resource_req['uri_pattern']}

        Available discovered resources:
        {[s['resources'] for s in discovered_servers]}

        Match based on data type and access pattern.
        Example: "transaction_history" ≈ "finance://transactions/*"
        """
        matches['resources'][resource_req['name']] = mistral_llm.match(prom
pt)

    # Match Prompts (interaction templates)
```

```
    for prompt_req in requirements['prompts']:
        prompt = f"""
        Required prompt: {prompt_req['name']} - {prompt_req['purpose']}

        Available discovered prompts:
        {[s['prompts'] for s in discovered_servers]}

        Match based on interaction purpose.
        Example: "financial_advice" ≈ "budget_guidance", "money_tips"
        """
        matches['prompts'][prompt_req['name']] = mistral_llm.match(prompt)

    return matches
```

Step 4: Using the Python SDK Client to Connect

```
from mcp import ClientSession
from mcp.client.stdio import stdio_client

async def connect_and_use_matched_capabilities(matches):
    """
    Connect to matched servers and use their capabilities
    """
    # Connect to server with matched capabilities
    server_params = StdioServerParameters(
        command=matches['server_executable'],
        args=matches['server_args']
    )

    async with stdio_client(server_params) as (read, write):
        async with ClientSession(read, write) as session:
            await session.initialize()

            # Use matched tool
            if 'bank_statement_parser' in matches['tools']:
                actual_tool = matches['tools']['bank_statement_parser']['actual_n
ame']
                result = await session.call_tool(actual_tool, {
```

```
                "file_path": "/path/to/statement.pdf"
            })

        # Use matched resource
        if 'market_data' in matches['resources']:
            actual_uri = matches['resources']['market_data']['actual_uri']
            data = await session.read_resource(actual_uri)

        # Use matched prompt
        if 'financial_advice' in matches['prompts']:
            actual_prompt = matches['prompts']['financial_advice']['actual_n
ame']
            advice = await session.get_prompt(actual_prompt, {
                "user_profile": "student",
                "goal": "save money"
            })
```

5.4 Real-World Example Flow with All Primitives

```
1. User: "Help me manage my student finances"
   ↓
2. Base Agent generates requirements:
   Tools: ["bank_statement_parser", "budget_planner"]
   Resources: ["transaction_history", "tax_rules"]
   Prompts: ["financial_advice", "savings_goals"]
   ↓
3. MCP Module discovers servers:
   Server A (port 3002):
     - Tools: ["pdf_analyzer", "expense_calculator"]
     - Resources: ["finance://docs/*", "finance://data/*"]
     - Prompts: ["money_tips", "budget_help"]
   Server B (process "fin-mcp"):
     - Tools: ["budget_creator", "subscription_finder"]
     - Resources: ["tax://rules/*"]
     - Prompts: ["saving_strategies"]
   ↓
4. Mistral 7B matches semantically:
   "bank_statement_parser" → Server A's "pdf_analyzer" (89%)
```

```
   "budget_planner" → Server B's "budget_creator" (92%)
   "transaction_history" → Server A's "finance://data/*" (85%)
   "tax_rules" → Server B's "tax://rules/*" (95%)
   "financial_advice" → Server A's "money_tips" (87%)
   "savings_goals" → Server B's "saving_strategies" (90%)
   ↓
5. Creates comprehensive bindings for all three primitive types
```

Once servers are discovered, the module queries their capabilities:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-03-26",
    "capabilities": {
      "tools": {},
      "resources": {},
      "prompts": {}
    },
    "clientInfo": {
      "name": "MCP-Module",
      "version": "1.0.0"
    }
  }
}
```

Server response includes full capability manifest:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "protocolVersion": "2025-03-26",
    "capabilities": {
      "tools": {
        "listChanged": true
```

```
      },
      "resources": {
        "subscribe": true,
        "listChanged": true
      },
      "prompts": {
        "listChanged": true
      }
    },
    "serverInfo": {
      "name": "finance-core-mcp",
      "version": "1.0.0"
    }
  }
}
```

6. Intelligent Matching with Mistral 7B for All Primitives

6.1 Semantic Matching Pipeline for Tools, Resources, and Prompts

The matching pipeline handles ALL three MCP primitive types from the base agent JSON:

```
class SemanticMatcher:
    def __init__(self, llm_model="mistral-7b"):
        self.llm = load_local_llm(llm_model)
        self.embedding_model = load_embedding_model()

    async def match_all_primitives(self, requirements, discovered_servers):
        """
        Match tools, resources, and prompts semantically
        All names are dynamically generated - no exact matching
        """
        matches = {
            'tools': await self.match_tools(requirements['tools'], discovered_ser
vers),
```

```
            'resources': await self.match_resources(requirements['resources'],
discovered_servers),
            'prompts': await self.match_prompts(requirements['prompts'], disco
vered_servers)
        }
        return matches

    async def match_tools(self, required_tools, servers):
        """Match required tools to discovered tool capabilities"""
        for tool_req in required_tools:
            prompt = f"""
            Required TOOL from base agent: {tool_req['name']}
            Purpose: {tool_req['purpose']}

            This is a MODEL-CONTROLLED action that the LLM will invoke.

            Available discovered tools from MCP servers:
            {[s['capabilities']['tools'] for s in servers]}

            Match based on functional purpose, not name.
            Examples:
            - "bank_statement_parser" ≈ "pdf_analyzer", "document_reader"
            - "expense_tracker" ≈ "spending_monitor", "transaction_analyzer"
            """
            # Semantic matching logic

    async def match_resources(self, required_resources, servers):
        """Match required resources to discovered URIs"""
        for resource_req in required_resources:
            prompt = f"""
            Required RESOURCE from base agent: {resource_req['name']}
            URI pattern needed: {resource_req.get('uri_pattern', 'any')}

            This is APPLICATION-CONTROLLED read-only data.

            Available discovered resources from MCP servers:
            {[s['capabilities']['resources'] for s in servers]}
```

```
        Match based on data type and access pattern.
        Examples:
        - "transaction_history" ≈ "finance://transactions/*"
        - "market_data" ≈ "finance://market/*", "stocks://quotes/*"
        """
        # Semantic matching logic

    async def match_prompts(self, required_prompts, servers):
        """Match required prompts to discovered templates"""
        for prompt_req in required_prompts:
            prompt = f"""
            Required PROMPT from base agent: {prompt_req['name']}
            Purpose: {prompt_req['purpose']}

            This is a USER-CONTROLLED interaction template.

            Available discovered prompts from MCP servers:
            {[s['capabilities']['prompts'] for s in servers]}

            Match based on interaction purpose.
            Examples:
            - "financial_advice" ≈ "budget_guidance", "money_tips"
            - "study_planner" ≈ "learning_schedule", "study_guide"
            """
            # Semantic matching logic
```

6.2 Comprehensive Matching Prompt Template

```
COMPREHENSIVE_MATCHING_PROMPT = """
You are matching requirements from a dynamically generated base agent J
SON to discovered MCP server capabilities.

IMPORTANT: The base agent generates abstract names based on user pro
mpts. The MCP servers have their own naming. Names will NEVER match e
xactly - you must understand semantic meaning.

MCP has three primitive types:
1. TOOLS: Model-controlled functions for actions (e.g., API calls, computati
```

ons)
2. RESOURCES: Application-controlled read-only data (e.g., files, database
s)
3. PROMPTS: User-controlled interaction templates (e.g., consultations, gui
dance)

Required capabilities from base agent:
- Tools: {required_tools}
- Resources: {required_resources}
- Prompts: {required_prompts}

Discovered MCP servers and their capabilities:
{discovered_servers}

Task:
For each required capability:
1. Identify which primitive type it is
2. Find semantically matching capabilities from discovered servers
3. Consider functional purpose, not names
4. Provide confidence scores

Output Format:

```
{
  "tools": {
    "required_tool_name": {
      "matched_server": "server_name",
      "matched_tool": "actual_tool_name",
      "confidence": 0-100,
      "reasoning": "explanation"
    }
  },
  "resources": {
    "required_resource_name": {
      "matched_server": "server_name",
      "matched_uri": "actual_uri_pattern",
      "confidence": 0-100
    }
  },
```

```
  "prompts": {
    "required_prompt_name": {
      "matched_server": "server_name",
      "matched_prompt": "actual_prompt_name",
      "confidence": 0-100
    }
  }
}
"""
```

6.3 Confidence Scoring for All Primitives

```
def calculate_confidence(requirement, capability, primitive_type):
    """
    Calculate matching confidence for any primitive type
    """
    if primitive_type == "tool":
        # Tools: Match based on action/function similarity
        factors = [
            semantic_similarity(requirement['purpose'], capability['descriptio
n']),
            parameter_compatibility(requirement, capability),
            action_type_match(requirement, capability)
        ]

    elif primitive_type == "resource":
        # Resources: Match based on data type and URI pattern
        factors = [
            uri_pattern_match(requirement['uri_pattern'], capability['uri']),
            data_type_similarity(requirement, capability),
            access_pattern_match(requirement, capability)
        ]

    elif primitive_type == "prompt":
        # Prompts: Match based on interaction purpose
        factors = [
            interaction_similarity(requirement['purpose'], capability['descriptio
n']),
```

```
        parameter_match(requirement['parameters'], capability['argument
s']),
        template_compatibility(requirement, capability)
    ]

    return weighted_average(factors)
```

6. Transport Protocol Selection

6.1 Protocol Decision Matrix

Factor	stdio	HTTP/SSE	WebSocket
Latency	Low (local)	Medium	Low
Throughput	High	Medium	High
Stateful	Yes	No	Yes
Remote Support	No	Yes	Yes
Streaming	Yes	Yes (SSE)	Yes
Authentication	Process-based	OAuth 2.1/API Keys	Token-based
Use Case	Local tools, File I/O	REST APIs, Stateless ops	Real-time, Streaming

6.2 Dynamic Protocol Selection Algorithm

```
def select_transport_protocol(tool_requirements, server_location, performa
nce_needs):
    # Local server with file operations → stdio
    if server_location == "local" and tool_requirements.involves_files:
        return "stdio"

    # Remote server or needs authentication → HTTP
    if server_location == "remote" or tool_requirements.needs_auth:
        return "http"

    # Real-time or streaming requirements → WebSocket
    if performance_needs.real_time or tool_requirements.streaming:
        return "websocket"
```

```
    # Default fallback
    return "stdio" if server_location == "local" else "http"
```

7. Configuration Generation

7.1 Processing Variable Base Agent Input

The MCP module processes whatever base agent JSON is provided, adapting to different domains, tool requirements, and agent configurations:

```
class MCPConfigurationGenerator:
    def generate_config(self, base_agent_json):
        """
        Generate MCP configuration for ANY base agent JSON
        Adapts to whatever tools and agents are defined
        """
        # Extract variable requirements from input
        tool_requirements = self.extract_requirements(base_agent_json)

        # Discover available servers (not predefined)
        discovered_servers = self.discover_servers()

        # Match requirements to discoveries
        matches = self.semantic_match(tool_requirements, discovered_server
s)

        # Generate configuration
        return self.build_config(matches)
```

8.2 Output JSON Structure with All Primitives

The MCP module generates a comprehensive configuration that includes bindings for tools, resources, and prompts:

```
{
  "mcp_configuration": {
    "version": "2025-03-26",
    "input_metadata": {
```

```
  "base_agent_json_id": "workflow_20250812_195444",
  "primitives_requested": {
    "tools": 12,
    "resources": 8,
    "prompts": 5
  },
  "primitives_matched": {
    "tools": 11,
    "resources": 8,
    "prompts": 4
  }
 },
 "discovery_metadata": {
  "discovered_servers": 7,
  "total_capabilities": {
    "tools": 45,
    "resources": 23,
    "prompts": 15
  }
 },
 "servers": {
  "discovered-finance-server": {
    "endpoint": "stdio://localhost/usr/local/mcp/finance",
    "transport": "stdio",
    "capabilities": {
      "tools": {
        "list": ["pdf_analyzer", "budget_creator"],
        "listChanged": true
      },
      "resources": {
        "list": ["finance://transactions/*", "finance://market/*"],
        "subscribe": true,
        "listChanged": true
      },
      "prompts": {
        "list": ["financial_advice", "budget_review"],
        "listChanged": true
      }
```

```
    }
   }
  },
  "tool_bindings": {
   "agent_1": {
    "bank_statement_parser": {
     "matched_to": {
      "server": "discovered-finance-server",
      "method": "pdf_analyzer",
      "confidence": 89,
      "semantic_match_reason": "Both parse financial documents"
     }
    }
   }
  },
  "resource_bindings": {
   "agent_1": {
    "transaction_history": {
     "matched_to": {
      "server": "discovered-finance-server",
      "uri": "finance://transactions/*",
      "confidence": 92,
      "access_pattern": "read-only"
     }
    },
    "market_data": {
     "matched_to": {
      "server": "discovered-finance-server",
      "uri": "finance://market/{symbol}",
      "confidence": 95,
      "refresh_interval": 60000
     }
    }
   }
  },
  "prompt_bindings": {
   "agent_1": {
    "financial_advice": {
```

```
        "matched_to": {
          "server": "discovered-finance-server",
          "prompt_name": "financial_advice",
          "confidence": 98,
          "arguments": ["user_profile", "goal"]
        }
      }
    }
  }
 }
}
```

---

9. Implementation Roadmap

Note: All implementation will use the official Python SDK and support all three MCP primitives (tools, resources, prompts).

Phase 1: Core Infrastructure with SDK (Week 1-2)

- [ ] Set up Python MCP SDK environment
  - [ ] Install `mcp[cli]` package
  - [ ] Configure development environment
- [ ] Implement MCP Discovery Engine
  - [ ] Server discovery using ClientSession
  - [ ] Query tools via `list_tools()`
  - [ ] Query resources via `list_resources()`
  - [ ] Query prompts via `list_prompts()`
- [ ] Build Server Registry Manager
  - [ ] Store discovered capabilities for all primitives
  - [ ] Health monitoring for servers

Phase 2: Protocol Implementation (Week 3-4)

- [ ] Implement transport layers using SDK
  - [ ] [ ]]stdio transport with `stdio_client`

- [ ] HTTP transport with `streamablehttp_client`
- [ ] WebSocket support (if needed)
- [ ] Authentication module
  - [ ] OAuth 2.1 with `OAuthClientProvider`
  - [ ] Token storage implementation

Phase 3: Intelligent Matching (Week 5-6)

- [ ] Integrate Mistral 7B for semantic matching
  - [ ] Tool matching pipeline
  - [ ] Resource URI pattern matching
  - [ ] Prompt template matching
- [ ] Build confidence scoring system
- [ ] Implement fallback resolution for all primitives

Phase 4: Domain MCP Servers with FastMCP (Week 7-10)

- [ ] Create servers using FastMCP framework
  - [ ] Implement `@mcp.tool()` decorators
  - [ ] Implement `@mcp.resource()` decorators
  - [ ] Implement `@mcp.prompt()` decorators
- [ ] Domain implementations:
  - [ ] Finance (tools, resources, prompts)
  - [ ] Productivity (tools, resources, prompts)
  - [ ] Education (tools, resources, prompts)
  - [ ] Sports (tools, resources, prompts)
  - [ ] Software Dev (tools, resources, prompts)

Phase 5: Integration & Testing (Week 11-12)

- [ ] Integration with Base Agent Module
- [ ] Test all three primitive types
- [ ] Use MCP Inspector for debugging

- [ ] Performance optimization
- [ ] Documentation completion

9. Development Guidelines

9.1 Code Structure with Python SDK

```
mcp-module/
├── src/
│   ├── discovery/
│   │   ├── __init__.py
│   │   ├── server_discovery.py      # Uses ClientSession to discover
│   │   ├── capability_query.py      # list_tools(), list_resources(), list_prompt
s()
│   │   └── registry.py              # Store all discovered primitives
│   ├── matching/
│   │   ├── __init__.py
│   │   ├── semantic_matcher.py      # Match all three primitive types
│   │   ├── tool_matcher.py          # Specific tool matching logic
│   │   ├── resource_matcher.py      # URI pattern matching
│   │   ├── prompt_matcher.py        # Prompt template matching
│   │   └── mistral_integration.py   # LLM for semantic understanding
│   ├── client/
│   │   ├── __init__.py
│   │   ├── mcp_client.py            # Main client using SDK
│   │   ├── stdio_handler.py         # stdio_client wrapper
│   │   ├── http_handler.py          # streamablehttp_client wrapper
│   │   └── auth_handler.py          # OAuthClientProvider integration
│   ├── servers/                     # FastMCP server implementations
│   │   ├── __init__.py
│   │   ├── base_server.py           # FastMCP base class
│   │   ├── finance/
│   │   │   ├── finance_tools.py     # @mcp.tool() implementations
│   │   │   ├── finance_resources.py # @mcp.resource() implementations
│   │   │   └── finance_prompts.py   # @mcp.prompt() implementations
│   │   ├── productivity/
│   │   ├── education/
```

```
│   │   ├── sports/
│   │   └── development/
│   ├── config/
│   │   ├── __init__.py
│   │   ├── generator.py          # Generate config for all primitives
│   │   └── validator.py          # Validate bindings
│   └── main.py
├── tests/
│   ├── test_discovery.py         # Test all primitive discovery
│   ├── test_matching.py          # Test semantic matching
│   ├── test_tools.py             # Tool-specific tests
│   ├── test_resources.py         # Resource-specific tests
│   ├── test_prompts.py           # Prompt-specific tests
│   └── test_integration.py       # End-to-end tests
├── docs/
├── pyproject.toml                # Project configuration
└── requirements.txt
```

9.2 Key Dependencies

```
# requirements.txt
mcp[cli]>=1.0.0         # Official MCP Python SDK with CLI
transformers==4.36.0    # Mistral 7B integration
pydantic==2.0           # Data validation for structured output
httpx==0.25.0           # HTTP client for discovery
psutil==5.9.0           # Process discovery
redis==5.0              # Server registry cache (optional)
pytest==7.4             # Testing
pytest-asyncio==0.21    # Async testing
```

9.3 Example Server Implementation

```
# src/servers/finance/finance_server.py
from mcp.server.fastmcp import FastMCP, Context
from mcp.server.fastmcp.prompts import base
from pydantic import BaseModel
```

```
# Initialize server
finance_mcp = FastMCP("finance-analyzer")

# --- TOOLS (Model-controlled actions) ---
@finance_mcp.tool()
async def analyze_bank_statement(
    file_path: str,
    ctx: Context
) → dict:
    """Analyze bank statement and categorize transactions"""
    # Log progress
    await ctx.report_progress(0.5, "Parsing statement...")

    # Tool implementation
    return {
        "total_expenses": 2500.00,
        "categories": {"food": 800, "transport": 300},
        "subscriptions_found": ["netflix", "spotify", "gym"]
    }

# --- RESOURCES (Application-controlled data) ---
@finance_mcp.resource("finance://transactions/{account}/{month}")
def get_transactions(account: str, month: str) → str:
    """Provide transaction data for specific account and month"""
    # Resource implementation
    return f"Transaction data for {account} in {month}: ..."

@finance_mcp.resource("finance://market-data/{symbol}")
def get_market_data(symbol: str) → str:
    """Real-time market data for symbol"""
    return f"Current price for {symbol}: $150.25"

# --- PROMPTS (User-controlled templates) ---
@finance_mcp.prompt()
def financial_consultation(
    income: float,
    expenses: float,
    goal: str = "save money"
```

```
) -> list[base.Message]:
    """Interactive financial consultation"""
    return [
        base.UserMessage(f"My income: ${income}, expenses: ${expense
s}"),
        base.UserMessage(f"Goal: {goal}"),
        base.AssistantMessage("Let me analyze your finances and provide pe
rsonalized advice...")
    ]

# Run server
if __name__ == "__main__":
    # Can run with different transports
    finance_mcp.run(transport="stdio")  # or "streamable-http"
```

9.3 Error Handling Strategy

```
class MCPError(Exception):
    """Base MCP exception"""
    pass

class ServerDiscoveryError(MCPError):
    """Failed to discover MCP servers"""
    pass

class CapabilityMismatchError(MCPError):
    """No matching capability found"""
    pass

class TransportError(MCPError):
    """Transport layer communication failed"""
    pass

class AuthenticationError(MCPError):
    """Authentication with MCP server failed"""
    pass

# Error recovery patterns
```

```
async def resilient_tool_call(tool_name, params, primary_server, fallback_se
rver=None):
    try:
        return await primary_server.call_tool(tool_name, params)
    except (TransportError, TimeoutError) as e:
        if fallback_server:
            logger.warning(f"Primary server failed: {e}, using fallback")
            return await fallback_server.call_tool(tool_name, params)
        raise
    except AuthenticationError:
        await refresh_authentication(primary_server)
        return await primary_server.call_tool(tool_name, params)
```

---

10. Python SDK Integration

10.1 Core Dependencies

Based on the official MCP Python SDK:

```
# requirements.txt
mcp[cli]>=1.0.0          # Official MCP Python SDK with CLI tools
```

10.2 MCP Module Implementation with SDK

```
from mcp import ClientSession
from mcp.client.stdio import stdio_client
from mcp.client.streamable_http import streamablehttp_client
from mcp.server.fastmcp import FastMCP
import asyncio

class MCPModule:
    """
    Main MCP Module that discovers, matches, and connects to MCP server
s
    """

    def __init__(self, llm_model="mistral-7b"):
```

```
        self.llm = load_local_llm(llm_model)
        self.discovered_servers = []

    async def discover_servers(self):
        """Discover all available MCP servers"""
        servers = []

        # Try connecting to potential servers
        for port in range(3000, 4001):
            try:
                async with streamablehttp_client(f"http://localhost:{port}/mcp") a
s (r, w, _):
                    async with ClientSession(r, w) as session:
                        await session.initialize()

                        # Get server capabilities
                        tools = await session.list_tools()
                        resources = await session.list_resources()
                        prompts = await session.list_prompts()

                        servers.append({
                            "endpoint": f"http://localhost:{port}/mcp",
                            "transport": "http",
                            "capabilities": {
                                "tools": [t.name for t in tools.tools],
                                "resources": [r.uri for r in resources.resources],
                                "prompts": [p.name for p in prompts.prompts]
                            }
                        })
            except:
                continue

        self.discovered_servers = servers
        return servers

    async def match_capabilities(self, base_agent_json):
        """Match base agent requirements to discovered capabilities"""
        requirements = self.extract_requirements(base_agent_json)
```

```
    matches = {}

    for req_type in ['tools', 'resources', 'prompts']:
        matches[req_type] = {}

        for requirement in requirements[req_type]:
            # Use LLM for semantic matching
            best_match = await self.semantic_match(
                requirement,
                self.discovered_servers,
                req_type
            )
            matches[req_type][requirement['name']] = best_match

    return matches

async def execute_matched_tool(self, tool_binding, params):
    """Execute a tool using its matched server and method"""
    server = tool_binding['server']
    actual_tool = tool_binding['actual_tool']

    # Connect based on transport type
    if server['transport'] == 'stdio':
        server_params = StdioServerParameters(
            command=server['command'],
            args=server['args']
        )
        async with stdio_client(server_params) as (read, write):
            async with ClientSession(read, write) as session:
                await session.initialize()
                return await session.call_tool(actual_tool, params)

    elif server['transport'] == 'http':
        async with streamablehttp_client(server['endpoint']) as (r, w, _):
            async with ClientSession(r, w) as session:
                await session.initialize()
                return await session.call_tool(actual_tool, params)
```

```
    async def read_matched_resource(self, resource_binding):
        """Read a resource using its matched URI"""
        server = resource_binding['server']
        actual_uri = resource_binding['actual_uri']

        # Similar connection logic for resources
        async with self.connect_to_server(server) as session:
            return await session.read_resource(actual_uri)

    async def get_matched_prompt(self, prompt_binding, arguments):
        """Get a prompt using its matched name"""
        server = prompt_binding['server']
        actual_prompt = prompt_binding['actual_prompt']

        async with self.connect_to_server(server) as session:
            return await session.get_prompt(actual_prompt, arguments)
```

10.3 Testing MCP Servers with SDK Tools

The Python SDK provides tools for testing:

```
# Test server with MCP Inspector
uv run mcp dev server.py

# Install server in Claude Desktop
uv run mcp install server.py --name "Finance Analyzer"

# Run server directly
uv run mcp run server.py
```

10.4 Creating Domain-Specific MCP Servers

Example implementation for a finance domain server using FastMCP:

```
from mcp.server.fastmcp import FastMCP, Context
from pydantic import BaseModel

# Create finance MCP server
finance_mcp = FastMCP("finance-analyzer")
```

```
# Tool with structured output
class BudgetAnalysis(BaseModel):
    total_income: float
    total_expenses: float
    savings_potential: float
    recommendations: list[str]

@finance_mcp.tool()
def analyze_budget(transactions: list[dict]) -> BudgetAnalysis:
    """Analyze transactions and provide budget recommendations"""
    # Implementation
    return BudgetAnalysis(
        total_income=5000,
        total_expenses=4200,
        savings_potential=800,
        recommendations=["Cancel unused subscriptions", "Reduce dining ou
t"]
    )

# Resource for data access
@finance_mcp.resource("finance://transactions/{month}")
def get_transactions(month: str) -> str:
    """Get transaction data for a specific month"""
    return f"Transaction data for {month}: ..."

# Prompt for user interaction
@finance_mcp.prompt()
def budget_consultation(income: float, expenses: float) -> str:
    """Interactive budget consultation prompt"""
    return f"Based on income ${income} and expenses ${expenses}, let's op
timize your budget"

# Run the server
if __name__ == "__main__":
    finance_mcp.run(transport="stdio")  # or "streamable-http" for HTTP tra
nsport
```

- Discovery mechanisms
- Protocol message handling
- Transport layer operations
- Semantic matching accuracy

10.2 Integration Tests

- Server connection establishment
- Tool invocation flow
- Resource retrieval
- Error recovery mechanisms

10.3 End-to-End Tests

- Complete workflow from base agent to tool execution
- Multi-domain scenarios
- Concurrent agent operations
- Performance under load

10.4 Test Data Requirements

```
# Sample test configuration
TEST_SCENARIOS = {
    "finance": {
        "input": "Help student manage monthly expenses",
        "expected_tools": ["analyze_bank_statement", "calculate_budget"],
        "expected_servers": ["finance-core-mcp"]
    },
    "productivity": {
        "input": "Summarize my emails and schedule meetings",
        "expected_tools": ["gmail_summarize", "schedule_meeting"],
        "expected_servers": ["productivity-gmail-mcp", "productivity-calendar
-mcp"]
    }
}
```

11. Performance Optimization

11.1 Caching Strategy

- Server capability cache (TTL: 5 minutes)
- Tool matching results cache (TTL: 10 minutes)
- Authentication token cache (TTL: based on expiry)

11.2 Connection Pooling

- Maintain persistent connections for stdio transport
- HTTP connection pooling with max 10 connections per server
- WebSocket connection reuse for streaming operations

11.3 Parallel Processing

- Concurrent server discovery
- Parallel capability introspection
- Batch tool matching for multiple agents

---

12. Security Considerations

12.1 Authentication

- OAuth 2.1 for remote servers
- API key rotation mechanism
- Secure credential storage (environment variables/secrets manager)

12.2 Authorization

- Tool-level access control
- Resource access restrictions
- Rate limiting per client

12.3 Data Protection

- Encrypt sensitive data in transit
- Sanitize tool parameters

- Audit logging for all tool invocations

---

13. Monitoring & Observability

13.1 Metrics to Track

- Server discovery success rate
- Tool matching accuracy
- Average response time per tool
- Error rates by category
- Fallback usage frequency

13.2 Logging Strategy

```
# Structured logging format
{
  "timestamp": "2025-08-26T10:30:00Z",
  "level": "INFO",
  "component": "mcp.discovery",
  "message": "Discovered 5 MCP servers",
  "context": {
    "servers": ["finance-core-mcp", "productivity-gmail-mcp"],
    "discovery_method": "network",
    "duration_ms": 234
  }
}
```

13.3 Health Checks

- Server availability monitoring
- Transport layer health
- Authentication status
- Resource utilization

---

14. Deployment Considerations

14.1 Container Strategy

```
# Dockerfile for MCP Module
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY src/ ./src/
ENV MCP_DISCOVERY_PORT=5000
ENV MCP_CACHE_TTL=300
CMD ["python", "-m", "src.main"]
```

14.2 Environment Variables

```
# Required environment variables
MCP_SERVERS=localhost:3001,localhost:3002
MCP_DISCOVERY_ENABLED=true
MCP_CACHE_REDIS_URL=redis://localhost:6379
MISTRAL_MODEL_PATH=/models/mistral-7b
MCP_LOG_LEVEL=INFO

# Domain-specific API keys
PLAID_CLIENT_ID=xxx
ALPHA_VANTAGE_API_KEY=xxx
GMAIL_API_KEY=xxx
```

14.3 Scaling Strategy

- Horizontal scaling for MCP module instances
- Load balancing for server connections
- Distributed cache for shared state
- Message queue for async tool invocations

---

15. Troubleshooting Guide

Common Issues and Solutions

Issue	Cause	Solution
Server not discovered	Server not running or wrong port	Check server process, verify port configuration
Tool matching fails	Semantic mismatch	Adjust matching threshold, improve tool descriptions
Authentication errors	Expired tokens	Implement token refresh mechanism
Transport timeouts	Network issues or server overload	Increase timeout, implement retry logic
High latency	Too many synchronous calls	Implement async operations, use connection pooling

16. API Reference

16.1 MCP Module Public API

```
class MCPModule:
    async def process_base_agent_json(self, ba_json: dict) → dict:
        """
        Process ANY base agent JSON regardless of content

        Args:
            ba_json: Variable JSON from base agent (changes per prompt)
                    Contains different tools, agents, configurations each time

        Returns:
            MCP configuration matched to discovered servers
        """

    async def discover_servers() → List[MCPServer]:
        """Find whatever MCP servers are currently available"""

    async def match_tools(self, requirements: dict, discoveries: dict) → Tool
Bindings:
        """Semantically match ANY tool requirements to ANY discovered capa
bilities"""
```

```
    async def generate_configuration(self, base_agent_json: dict) → dict:
        """Generate config for whatever base agent JSON is provided"""

    async def execute_tool(self, agent_id: str, tool_name: str, params: dict) →
Any:
        """Execute matched tool using discovered connection method"""
```

16.2 Dynamic Tool Matching API

```
async def match_any_tool(tool_requirement, discovered_servers):
    """
    Match ANY tool requirement to ANY discovered capability

    Example:
        Input: {"name": "budget_planner", "purpose": "Plan finances"}

        Could match to ANY of these discovered tools:
        - "financial_planner" (95% confidence)
        - "budget_calculator" (92% confidence)
        - "expense_manager" (78% confidence)
        - "money_tool" (65% confidence)

    The system doesn't need exact matches - it understands semantics
    """
```

16.3 Server Capability Query

```
# Request
{
  "jsonrpc": "2.0",
  "method": "tools/list",
  "id": 1
}

# Response
{
  "jsonrpc": "2.0",
```

```
  "id": 1,
  "result": {
    "tools": [
      {
        "name": "analyze_bank_statement",
        "description": "...",
        "inputSchema": {...}
      }
    ]
  }
}
```

---

17. Future Enhancements

17.1 Planned Features

- [ ] Dynamic MCP server generation for missing capabilities
- [ ] Multi-language SDK support (Java, Go, Rust)
- [ ] GraphQL interface for complex queries
- [ ] Real-time capability updates via WebSocket
- [ ] Distributed tracing integration
- [ ] A/B testing framework for tool selection

17.2 Research Areas

- Federated learning for tool matching improvement
- Quantum-resistant authentication mechanisms
- Edge deployment for local-first architecture
- Blockchain-based tool registry for decentralized discovery

Appendix A: Glossary

- **MCP**: Model Context Protocol
- **JSON-RPC**: JSON Remote Procedure Call protocol (v2.0)
- **FastMCP**: Python framework for building MCP servers

- **stdio**: Standard Input/Output transport
- **SSE**: Server-Sent Events (being superseded by Streamable HTTP)
- **Streamable HTTP**: Modern HTTP transport for MCP
- **Tool**: Model-controlled function that performs actions (e.g., API calls, computations)
- **Resource**: Application-controlled read-only data source (e.g., files, databases)
- **Prompt**: User-controlled interaction template for LLM interactions
- **Capability**: Feature or function exposed by MCP server (tools, resources, or prompts)
- **Transport**: Communication mechanism between client and server
- **ClientSession**: SDK class for connecting to MCP servers
- **Semantic Matching**: Using LLM to match by meaning rather than exact names

---

Appendix B: References

1. Model Context Protocol Specification v2025-03-26
2. MCP Python SDK: https://github.com/modelcontextprotocol/python-sdk
3. JSON-RPC 2.0 Specification
4. OAuth 2.1 Authorization Framework
5. Mistral 7B Documentation
6. MCP Official Documentation: https://modelcontextprotocol.io
7. FastMCP Framework Documentation
8. MCP Inspector Tool Documentation

Document Control

- **Version**: 1.0
- **Last Updated**: August 26, 2025
- **Authors**: AI Engineering Team

- **Review Status**: Ready for Development
- **Next Review**: September 30, 2025

Critical Design Notes

1. **Three MCP Primitives**: The system handles ALL three MCP primitives - Tools (model-controlled actions), Resources (application-controlled data), and Prompts (user-controlled templates)
2. **Variable Input**: The base agent JSON changes with every user prompt - the MCP module must handle ANY configuration
3. **No Predefinition**: Tool names, resource URIs, prompt names, server names, and capabilities are NEVER predefined or hardcoded
4. **Semantic Matching**: The system relies on LLM understanding for all three primitive types, not exact string matching
5. **Runtime Discovery**: All servers and their capabilities (tools, resources, prompts) are discovered when the module runs
6. **Python SDK Based**: Implementation uses the official MCP Python SDK with FastMCP framework
7. **Example Nature**: All server configurations shown in this document are examples only - actual discoveries will differ

---

This document serves as the authoritative technical specification for the MCP Module implementation. All development efforts should align with the architecture and guidelines outlined herein.