GAI-2104 Lesson Guide

Building Agentic AI with Model Context Protocol

ascendient learning

Table of Contents

Building Agentic AI with Model Context Protocol

MODULE 1

After completing this course, you will be able to:

- ✓ **Analyze and explain** the three-component MCP architecture and how it solves the M×N integration problem
- ✓ **Build and deploy** MCP servers that expose data through resources and enable actions through tools

- ✓ **Implement secure RAG systems** using MCP resource hierarchies with proper access controls
- ✓ **Design reusable prompt templates** and orchestrate multi-step AI workflows with error handling
- ✓ **Create production-ready integrations** between AI models and existing enterprise systems
- ✓ **Apply security best practices** including authentication, input validation, and audit logging in AI deployments

1.1. Why Model Context Protocol?

Imagine you're working on a critical project analysis:

- Your AI assistant excels at reasoning, but it can't access your project files, query your database, or run your analysis tools.
- You must manually copy and paste every piece of information into the chat window.

Does this sound familiar?

Iceberg diagram showing core LLM limitations below iceberg, such as lack of real-time data, tool access, and integration—highlighting the need for MCP to bridge these gaps.

Figure 1. The Iceberg of LLM Limitations

Details

This challenge stems from integration issues, not AI limitations. Today's AI models resemble powerful computers without internet connections—they process information brilliantly but remain isolated from the tools and data that would make them truly powerful.

1.1.1. Traditional LLM Limitations

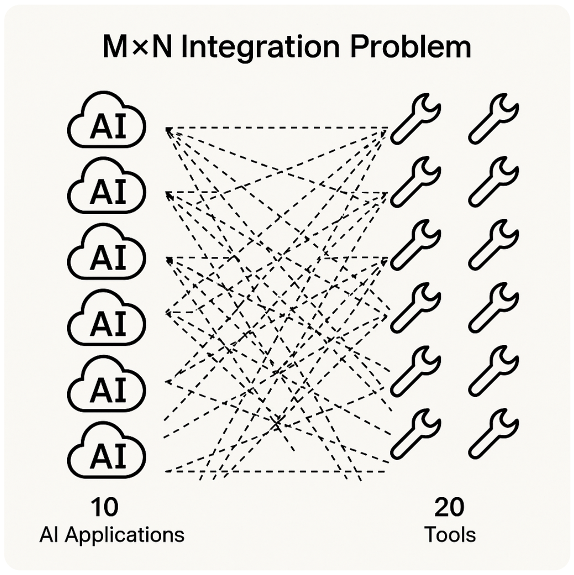
Without MCP, AI assistants face four critical limitations:

- **Isolated Knowledge**: AI only knows what was in its training data
- **Manual Updates**: You must copy-paste context repeatedly
- **No Real-time Data**: Cannot access live information or current files
- **Limited Actions**: Cannot interact with external systems or tools

1.2. The AI Integration Challenge

Every AI application needs to connect with:

- **Data sources**: Databases, APIs, files, cloud storage
- **Tools and services**: CI/CD, monitoring, analytics
- **Human workflows**: Approvals, reviews, escalations
- **Other AI systems**: Models, agents, orchestrators



Without a standard protocol, each connection requires custom integration.

1.2.1. AI Integration at Scale

Consider the problem of scale below.

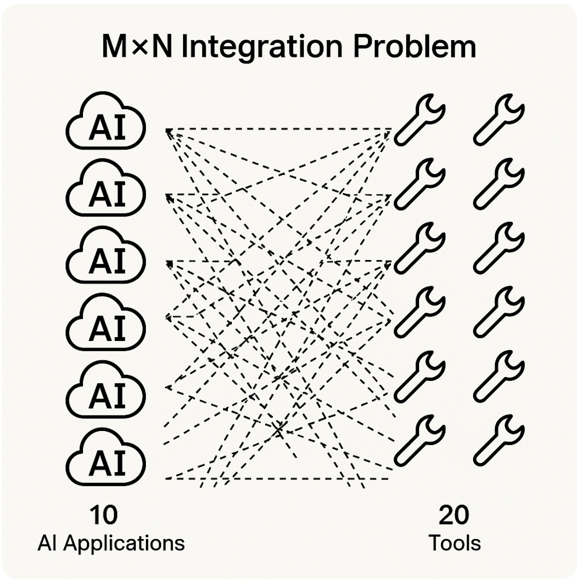
- Let’s say an organization uses:
  - 5 different AI applications
    - e.g. *ChatGPT, Claude, Gemini, internal chatbots, code assistants*
  - 10 essential tools and data sources
    - e.g. *databases, APIs, file systems, monitoring tools, CI/CD pipelines*

Caution

> Without a standard protocol, we would need to build **50 custom integrations** (5 × 10). Each integration (Application to Tool connection) requires dedicated time and resources for development and maintenance.

1.2.2. The M x N Problem

- These limitations create what researchers call the "M×N integration problem."
- If you have 10 AI applications that need to connect to 20 different tools, you would need to build 200 custom integrations [1].
- Each integration requires custom code, authentication, and maintenance.



1.3. Enter Model Context Protocol

- **MCP solves the M×N integration problem**: Instead of M agents needing to interface with N disparate tools creating M×N integration points, MCP introduces a standardized protocol reducing complexity to M+N integrations [1]

ⓘ Note

If you have 10 AI applications and 20 tools:

- Without MCP: 10 × 20 = 200 custom integrations needed
- With MCP: 10 + 20 = 30 MCP implementations needed

- **MCP is a "solution to AI integration bottlenecks"**: Eliminating the need for custom API wiring and manual authentication for each integration [2]
- In November 2024, Anthropic introduced the Model Context Protocol as an open standard to solve this problem [3].

1.3.1. The M + N Solution: MCP

- **MCP reduces integration complexity** from M×N to M+N

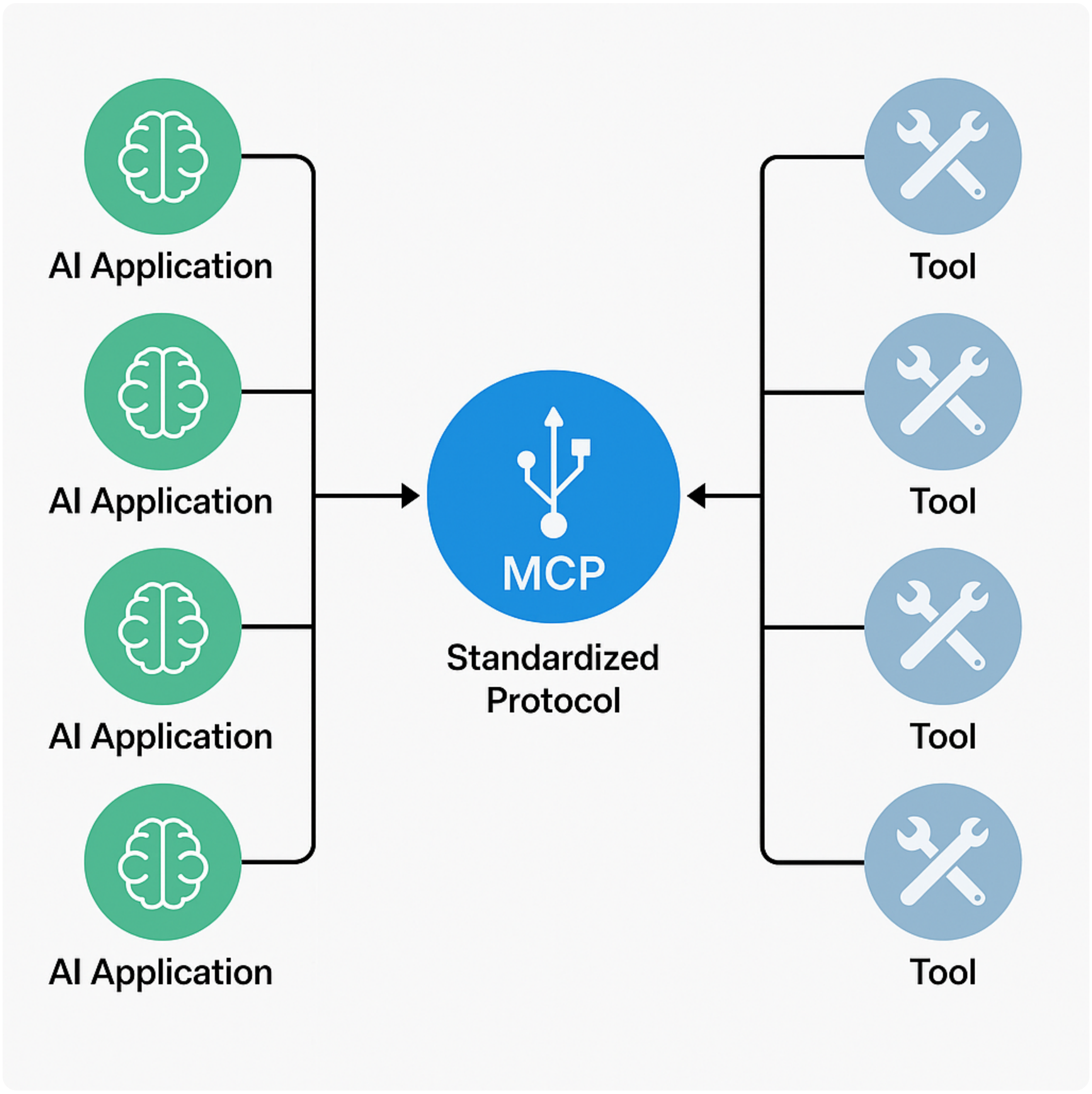


Figure 2. MCP: The M+N Solution to AI Integration Complexity

💡 Tip

Think of MCP as "USB-C for AI" – a universal connector that standardizes how AI applications connect to external data and tools [4].

1.4. MCP Impact: How MCP Changes the Game

Diagram showing the impact of the Model Context Protocol
Figure 3. MCP Impact Diagram

1.4.1. Real-World Impact

- **4,774+ community servers** already built [5]
- **23% of Fortune 500** exploring MCP adoption [6]
- **49.3% baseline task completion** in benchmarks [1]
- **40% token reduction** through intelligent caching [7]

1.5. The Six Stages of MCP Mastery

Chapter 1: START → Course introduction and MCP fundamentals

Chapter 2: CONNECT → Establishing AI connections to the world

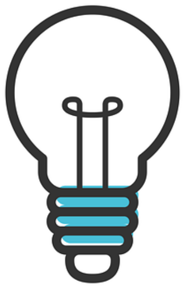
Chapter 3: READ → Accessing and consuming data intelligently

Chapter 4: UNDERSTAND → Recognizing patterns and encoding knowledge

Chapter 5: ACT → Taking controlled actions in systems

Chapter 6: COLLABORATE → Creating bidirectional AI-human workflows

Chapter 7: ORCHESTRATE → Building autonomous production systems



1.5.1. Your Learning Journey

Each chapter builds systematically on previous knowledge:

- **Chapter 1 (START)**: Course Introduction and MCP fundamentals
- **Chapter 2 (CONNECT)**: Master basic server-client communication
- **Chapter 3 (READ)**: Add intelligent data access and RAG capabilities

- **Chapter 4 (UNDERSTAND)**: Implement pattern recognition and knowledge encoding
- **Chapter 5 (ACT)**: Enable controlled system interactions and tool execution
- **Chapter 6 (COLLABORATE)**: Design human-AI collaboration workflows
- **Chapter 7 (ORCHESTRATE)**: Build autonomous multi-agent production systems

> Tip
>
> Think of it as learning to drive: first start the engine, then steer, then navigate, then handle traffic, then coordinate with others, finally manage complex routes.

1.6. Recommended Background

- Experience with any LLM API (OpenAI, Claude, etc.)
- Familiarity with client-server architecture
- Basic software engineering practices
- Curiosity about AI system design

> Note
>
> Don’t worry if you’re new to some concepts. The course is designed to bring you up to speed step-by-step.

Introduction to Model Context Protocol

MODULE 2

- ✓ Understand the fundamental purpose and value proposition of the Model Context Protocol (MCP)
- ✓ Identify the core architecture components: hosts, clients, and servers
- ✓ Analyze the layered protocol structure and message flow patterns
- ✓ Evaluate MCP's role in connecting AI applications to external data sources
- ✓ Apply MCP concepts to real-world AI integration scenarios

2.1. Why Does MCP Matter?

MCP allows organizations to focus on AI innovation, not integration overhead.

> ⓘ Note
>
> According to Gartner, 85% of AI projects fail to deliver due to integration complexity, data silos, or lack of governance. [8]

- **Developers**: reduces development time to build individual integrations to data sources and APIs
  - *e.g. Developers can focus on complex use cases without the burden of tool integration overhead*
- **AI Applications**: allows access to a rich ecosystem of apps, tools and data
  - *e.g. Agents can access your Google Calendar and Notion, acting as a more personalized AI assistant.*
- **End-Users**: benefit from AI applications (agents) that can for access data and integration with applications
  - *e.g. Enterprise chatbots can can connect to databases and integrate with enterprise application ecosystem*

[3]

2.2. MCP Servers for Context Delivery

LLMs require **context** to effectively provide meaningful results in the scope of a given situation or organization

Context can be surfaced to LLMs through several mechanisms:

- **Prompt Templates**: user-defined instructions providing background knowledge and instructions
- **Local Data / Databases**: providing access to file systems and database
- **API integration**: connecting to external data sources, tools, and services

MCP standardizes the various approaches for **context delivery** by providing a unified protocol for connecting AI applications to diverse data sources, tools, and services without custom integration work.

2.3. MCP Core Architecture Components

MCP follows a client-server architecture with three key participants:

- **MCP Host**: AI application (e.g., Claude Desktop, VS Code) that coordinates and manages multiple MCP clients
  - creates one MCP client for each MCP server
- **MCP Client**: Component that maintains one-to-one connection with an MCP server and obtains context for the host
- **MCP Server**: Program that provides context, tools, and resources to MCP clients

Each host can manage multiple clients, with each client maintaining a dedicated connection to one server.

2.3.1. MCP Architecture

```
┌───────────────────────────────────────────────────────┐
│                     HOST PROCESS                      │
│  ┌─────────────────────────────────────────────────┐  │
│  │  • AI/LLM Integration                           │  │
│  │  • User Authorization & Consent                 │  │
│  │  • Context Aggregation                          │  │
│  │  • Security Policy Enforcement                  │  │
│  └─────────────────────────────────────────────────┘  │
│         │              │              │               │
│         ▼              ▼              ▼               │
│  ┌──────────┐   ┌──────────┐   ┌──────────┐           │
│  │ Client 1 │   │ Client 2 │   │ Client N │           │
│  └──────────┘   └──────────┘   └──────────┘           │
└─────────┬──────────────┬──────────────┬───────────────┘
          │              │              │
          ▼              ▼              ▼
    ┌─────────┐    ┌─────────┐    ┌─────────┐
    │Server 1 │    │Server 2 │    │Server N │
    │(Local)  │    │(Remote) │    │(Remote) │
    └─────────┘    └─────────┘    └─────────┘
```

2.3.2. MCP Architecture Diagram

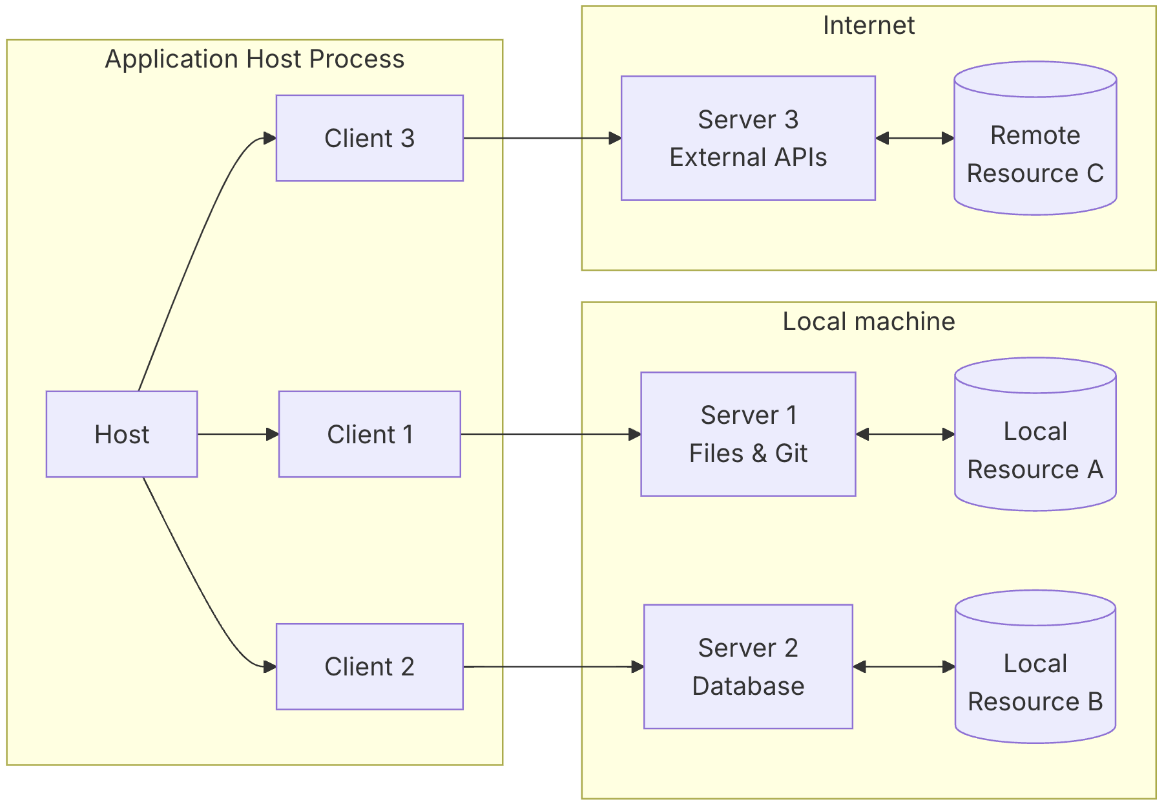


Figure 4. MCP Architecture Diagram

2.4. Example: VSCode with Github and Local Filesystem

VSCode can act as a **Host** application to integrate with **Github** and **Local Filesystem** as below:

- **Host**: VS Code application manages the user interface (receives user actions and manages mcp clients)
- **MCP Clients**: VS Code creates separate client connections for each server
  - Clients individually communicate with each MCP server and retrieve information for host
- **MCP Servers**: GitHub server (remote repositories) + Filesystem server (local files)
  - Github MCP server uses remote protocol and filesystem server sits locally

> Tip
>
> Note that MCP server refers to the program that serves context data, regardless of where it runs. MCP servers can execute locally or remotely.

Result: VS Code can simultaneously browse GitHub repos and access local files through a unified AI assistant interface.

2.5. MCP Server Building Blocks

MCP defines three core primitives (building blocks) that servers can expose:

- **Tools**: Executable functions that AI applications can invoke to perform actions
  - *e.g., file operations, API calls, database queries*
- **Resources**: Data sources that provide contextual information to AI applications
  - *e.g., file contents, database records, API responses*
- **Prompts**: Reusable conversation templates that help structure interactions with language models
  - *e.g., system prompts, prompting workflows*

2.6. MCP Server Primitives Functionality

Servers provide functionality through three building blocks:

Component	Description	Use Cases	Decision Maker
Tools	Executable operations that AI systems can invoke to perform specific actions. These enable the model to interact with external systems and modify state.	Execute database queries Process payment transactions Generate and send reports	AI Model
Resources	Static information sources that supply contextual data to enhance model understanding. These provide read-only access to structured content.	Customer database records Sales performance metrics Product inventory levels	Host Application
Prompts	Structured templates that guide model behavior and define interaction patterns for consistent responses.	SQL query generation Data analysis workflows Report formatting templates	End User

We will use a hypothetical scenario to demonstrate the role of each of these features, and show how they can work together.

2.6.1. MCP Server Primitive Example

2.6.2. Example: Database MCP Server

A database MCP server demonstrates all three primitives:

- **Tool**: `execute_query(sql: string)` - Run SQL queries
- **Resource**: `database://schema` - Database schema information
- **Prompt**: `sql_assistant` - Few-shot examples for SQL generation

This combination gives AI applications both the capability to query databases and the context needed to do so effectively.

- Each primitive supports discovery methods that let clients find what's available:
  - `*/list` - Discover available items (e.g., `tools/list`, `resources/list`)
- Primitives also support retrieval methods to surface responses from server to client
  - `*/get` - Retrieve specific items (e.g., `prompts/get`)
- Tools can also run execution logic with provided parameters
  - `tools/call` - tool execution

2.6.3. Tools: Executable Functions (Act)

Tools are functions that AI applications can invoke to perform actions:

- File operations (read, write, delete)
- API calls to external services
- Database queries and updates
- System commands and deployments

Tools follow a discovery (lookup) → execution pattern:

- `tools/list` - Discover available tools
- `tools/call` - Execute specific tool with parameters

Method	Purpose	Returns
`tools/list`	Discover available tools	Array of tool definitions with schemas
`tools/call`	Execute a specific tool	Tool execution result

2.6.4. Resources: Data Sources (Read)

Resources provide contextual information to AI applications:

- File contents and documentation
- Database records and schemas
- API responses and configuration data
- Real-time metrics and logs

Resources follow a discovery → retrieval pattern:

Method	Purpose	Returns
`resources/list`	Discover available resources	Array of resource definitions
`resources/read`	Retrieve specific resource content	Resource content with metadata

2.6.5. Prompts: Reusable Templates (Specify)

Prompts are *user-controlled* templates that help users interact with LLMs.

They're pre-configured conversation starters or task templates.

Prompts help structure and specify interactions with language models:

- Use well structured system prompts for consistent behavior
- Few-shot Learning prompting by providing examples for task guidance
- Conversation templates for consistent workflows / conversation types
- Domain-specific prompt engineering patterns

Prompts follow a discovery → retrieval pattern:

Method	Purpose	Returns
`prompts/list`	Discover available prompt templates	Array of prompt definitions
`prompts/get`	Retrieve specific prompt with parameters	Prompt content with arguments

2.6.6. Dynamic Discovery Pattern

Each primitive type supports dynamic discovery, allowing MCP clients to:

1. **List** available primitives at runtime
2. **Discover** capabilities without hardcoding
3. **Adapt** to changing server configurations
4. **Scale** to new tools and resources seamlessly

> ⓘ Important
>
> This discovery pattern enables MCP's flexibility - clients work with any server that implements the protocol, regardless of specific tools or resources offered.

2.7. Example: Travel Booking

Use Case: Travel Booking Tools enable AI applications to perform actions on behalf of users. In a travel planning scenario, the AI application might use several tools to help book a vacation:

Flight Search

- Queries multiple airlines and returns structured flight options.

```
searchFlights(origin: "NYC", destination: "Barcelona", date: "2024-06-15")
```

Calendar Blocking

- Marks the travel dates in the user’s calendar.

```
createCalendarEvent(title: "Barcelona Trip", startDate: "2024-06-15", endDate:
"2024-06-22")
```

Email notification

- Sends an automated out-of-office message to colleagues.

```
sendEmail(to: "team@work.com", subject: "Out of Office", body: "...")
```

2.8. Concept Review: MCP Server Methods

Category	Method	Purpose
Resources	`resources/list`	List available resources
Resources	`resources/read`	Read resource content
Resources	`resources/subscribe`	Subscribe to updates
Prompts	`prompts/list`	List available prompts
Prompts	`prompts/get`	Get prompt with arguments
Tools	`tools/list`	List available tools
Tools	`tools/call`	Execute a tool

2.9. In Depth Use Case: E-commerce Order Management

MCP primitives can enable AI applications to streamline order processing and customer service.

In an e-commerce scenario, the AI application would use three types of primitives:

- **Resources** provide access to data that the AI can read and reference
- **Tools** enable AI applications to streamline order processing and customer service.
- **Prompts** provide templated workflows and guidance for consistent responses

2.9.1. E-Commerce Order Management: Resources

Resources provide access to data that the AI can read and reference:

1. **Order Details**

    - Exposes structured order information as a readable resource.

```
resource://orders/ORD-789456
→ Returns: { orderId, items, status, total, shippingAddress, trackingNumber }
```

1. **Product Catalog**

    - Makes product inventory and specifications available.

```
resource://products/SKU-9876
→ Returns: { productId, name, price, stockLevel, warehouse, description }
```

2.9.2. E-Commerce Order Management: Tools

Tools perform actions and operations:

1. **Process Refund**

    - Initiates a refund to the customer’s original payment method.

```
processRefund(orderId: "ORD-789456", amount: 89.99, reason: "Product damaged")
```

1. **Update Order Status**
    - Changes the order status and triggers notifications.

```
updateOrderStatus(orderId: "ORD-789456", status: "shipped", trackingNumber:
"1Z999AA10123456784")
```

1. **Send Customer Notification**
    - Sends email or SMS updates to customers.

```
sendNotification(customerId: "CUST-12345", channel: "email", template:
"order_shipped", variables: {...})
```

2.9.3. E-Commerce Order Management: Prompts

Prompts provide templated workflows and guidance for consistent responses from AI:

Return Request Handler - Pre-configured prompts for processing return requests with company policy.

```
prompt://customer-service/return-request
→ Guides AI through: eligibility check → generate return label → explain timeline
```

Order Issue Resolution - Structured approach to resolving common order problems.

```
prompt://support/order-issue
→ Provides: troubleshooting steps, escalation criteria, compensation guidelines
```

2.10. Concept Check: Categorize MCP Server Primitives

Scenario: Imagine that you are building an MCP server for a library system.

Your Task: Categorize each of the features below as Resource, Tool, or Prompt.

Library System Feature	Type
Display list of available books	Resource, Tool or Prompt?
Check out a book	Resource, Tool or Prompt?
Generate "recommend similar books" conversation starter	Resource, Tool or Prompt?
Search books by author	Resource, Tool or Prompt?
Return a book	Resource, Tool or Prompt?
Show book details (title, author, ISBN)	Resource, Tool or Prompt?

2.11. MCP Clients: Connecting Host to Servers

MCP Clients are instantiated by host applications to communicate with particular MCP servers.

- The host application (Claude Desktop, VSCode, Github Copilot) manages the overall user experience and coordinates multiple clients.
  - Each client handles one direct communication with one server.
- In relation to MCP server primitives, an MCP Client can:
  - invoke tools exposed by a server
  - query for resources exposed by a server
  - interpolates prompts exposed by a server

> ⓘ Important

> Understanding the distinction between host and client is important: the host is the application users interact with, while clients are the protocol-level components that enable server connections.

2.12. Client Functionality

MCP clients serve as the bridge between host applications and MCP servers.

Each client is created by the host and maintains an isolated server connection:

- **Establishes one stateful session per server** - ensuring dedicated communication channels
- **Handles protocol negotiation and capability exchange** - managing version compatibility and feature discovery
- **Routes protocol messages bidirectionally** - facilitating two-way communication between host and server
- **Manages subscriptions and notifications** - handling real-time updates and event streaming
- **Maintains security boundaries between servers** - preventing cross-contamination between different data sources

A host application creates and manages multiple clients, with each client having a 1:1 relationship with a particular server.

2.12.1. Client Primitives

Client primitives enable servers to request additional capabilities from the host through the client:

Feature	Explanation	Example
Sampling	Allows servers to request LLM completions through the client, enabling agentic workflows while maintaining client control over permissions and security.	A travel booking server sends flight options to an LLM and requests the best flight selection for the user.
Roots	Enables clients to specify which directories servers should focus on, communicating intended scope through a coordination mechanism.	A travel server is given access to a specific directory containing the user's calendar for booking coordination.
Elicitation	Enables servers to request specific information from users during interactions, providing structured data gathering on demand.	A travel booking server asks for user preferences on seats, room type, or contact details to finalize reservations.

These client features create bidirectional communication, allowing servers to leverage the host's capabilities while respecting security boundaries. [9]

2.13. MCP Layers

MCP consists of two layers: a data layer and a transport layer.

Data Layer:

- Defines the JSON-RPC-based messaging protocol for client-server communication

- Involves lifecycle management and server/client primitives

Transport Layer manages communication channels and authentication between clients and servers.

- Defines communication mechanisms and channels for data exchange
- Handles transport-specific connection establishment
- Manages message framing and authorization

2.13.1. The Data Layer

- **Message Schema and Semantics**: defines the messaging protocol for client-server communication (JSON-RPC)
- **Lifecycle management**: connection initialization, capability negotiation, and connection termination
- **Server features**: enables core functionality of MCP server primitives (tools, resources, prompts) to / from the client
- **Client features**: enables servers to ask the client to sample from the host LLM, elicit input from the user, and log messages to the client
- **Utility features**: notifications for real-time updates and progress tracking for long-running operations

2.13.2. The Transport Layer

Transports are the communication methods that allow MCP clients and servers to exchange messages.

Key Functions:

- Connection establishment and authentication
- Message framing and secure communication
- Transport abstraction for protocol flexibility

- secure communication between all MCP participants

Two Transport Options: `Stdio` transport and `Streamable HTTP` Transport

The transport layer abstracts communication details from the protocol layer, enabling the same JSON-RPC 2.0 message format across all transport mechanisms. [10]

2.14. Transport Options

`stdio` **Transport:**

- Uses standard input/output streams for direct process communication
- Ideal for local processes on same machine
- Maximum performance, no network overhead

`HTTP` **Transport:**

- Uses HTTP POST for client-to-server messages
- Optional Server-Sent Events (SSE) for streaming
- Enables remote server communication
- Supports standard HTTP authentication (bearer tokens, API keys, and custom headers)
  - OAuth 2.0 recommended for secure tokens

> Tip
>
> Consider the best transport mechanism for a given client-server interaction. `stdio` is ideal for local, desktop, or development environments, while HTTP transport is a remote standard for Cloud and production environments.

2.15. Message Format

All transports use JSON-RPC 2.0 for messages:

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

- `stdio`: One message per line, no embedded newlines allowed
- `HTTP`: Messages sent as JSON in POST body or received via SSE stream
  - `{"jsonrpc": "2.0", "method": "…", …}`

2.16. Client Server Connection Lifecycle

The Model Context Protocol (MCP) establishes a structured connection lifecycle that governs client-server interactions through distinct phases.

1. **Initialization**: Protocol handshake with version alignment and capability discovery
2. **Operation**: Active communication phase for tool execution and resource access
3. **Shutdown**: Clean connection termination with proper resource cleanup

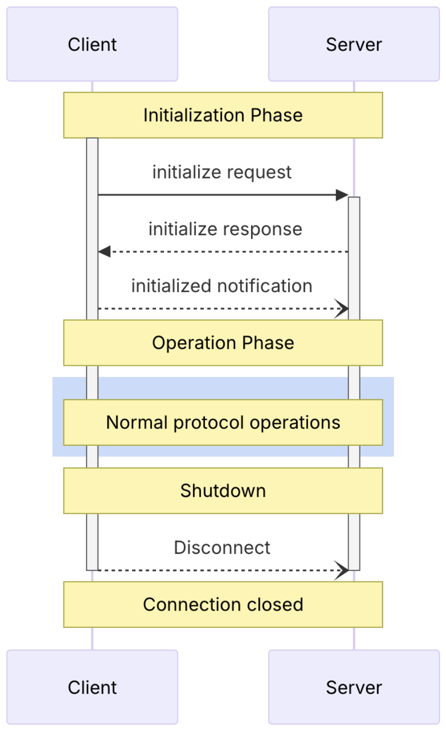


Figure 5. MCP Connection Lifecycle Sequence

2.17. Initialization and Server-Client Handshake

ⓘ Important

This process happens once at the start of every connection and must complete successfully before any actual work can begin.

Initialization is the handshake that happens when an MCP client first connects to an MCP server.

1. Agree on a protocol version
2. Exchange information about themselves
3. Declare what capabilities they support

> **Tip**
>
> Think of it like two people meeting for the first time — they introduce themselves and agree on how they'll communicate.

2.17.1. Handshake Messaging Protocol

JSON-RPC is a stateless, light-weight remote procedure call (RPC) protocol. [11]

1. Client sends a JSON-RPC message request with: protocol version (e.g. 06-18-2025), client capabilities (sampling, elicitation, roots), client information
2. Server responds with: the agreed protocol version (version both parties will use), server capabilities (prompts, resources, tools), server information
3. After receiving the server's response, the client sends a simple notification to confirm it's ready: `` ` { "jsonrpc": "2.0", "method": "notifications/initialized" } ``\`
4. After this notification, normal operations can begin.

2.18. Capability Negotiation

Negotiation is the process of figuring out what both sides (client and server) can do together.

1. Client declares what it supports (e.g., "I can provide filesystem roots")

2. Server declares what it provides (e.g., "I have tools and resources")

3. Both sides only use features that both support

2.18.1. Example Client-Server Negotiation

Imagine a case with the following specifications:

Component	Capabilities
Client	• Roots (with change notifications) • Sampling • Elicitation
Server	• Tools (with change notifications) • Resources (with subscriptions and change notifications) • Logging

Result: - Server can call the client’s sampling capability - Client can use the server’s tools and resources - Server can send log messages to the client - Both will notify each other when their lists change

2.19. Capability Negotiation Deep Dive

1. Client Initialization
2. Client provides its capabilities to Server
3. Server responds with its supported capabilities
4. Client and server agree on protocol version and capabilities
5. Handshake completes in active session with negotiated features

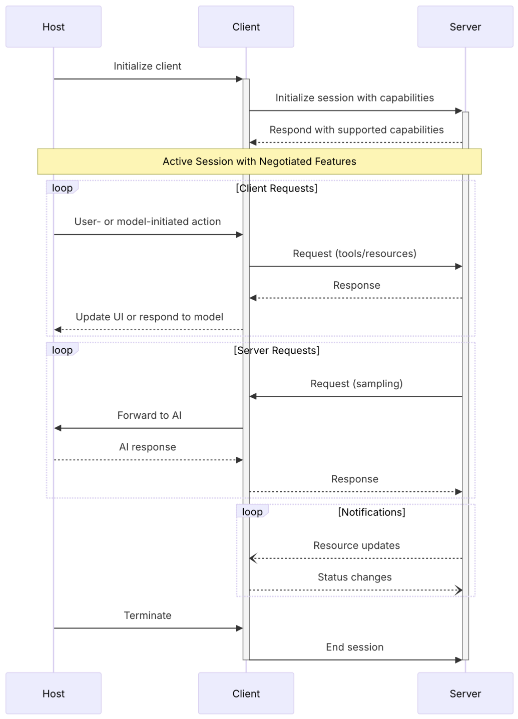


Figure 6. Capability Negotiation Sequence

2.20. Conceptual Model: Request-Response Cycle

MCP client-server interactions post-initialization generally follow the following pattern:

- **Discovery** - "What can you do?"
    - *Client* asks what’s available
    - *Server* lists capabilities
- **Invocation** - "Do this specific thing"

- *Client* requests action
- *Server* performs and responds
- **Notification** - "Things changed"
- *Server* alerts client to updates
- *Client* refreshes if needed

2.20.1. Client-Side Security

- **Client-Side Security Model**: All MCP clients operate within strict security boundaries enforced by the host application
- **Sandboxed Execution**: Each client connection runs in an isolated environment
- **Permission-Based Access Control**: Hosts grant specific permissions to clients based on user authorization and security policies
- **Transport-Level Security**: Both stdio and HTTP transports support authentication mechanisms and encrypted communication channels (Oauth 2.0 now supported)
- **Resource Isolation**: Clients cannot access resources beyond their designated scope, maintaining clear separation between different data sources
- **Audit and Logging**: All client-server interactions are logged for security monitoring and compliance tracking
- **User Consent Framework**: Sensitive operations require explicit user approval

> ⓘ Important
>
> Security is implemented at the host level, not the protocol level. This design allows hosts to enforce organization-specific security policies while maintaining protocol simplicity.

2.21. Tool Use Best Practices

For trust and safety, applications can implement user control through various mechanisms, such as:

- Displaying available tools in the UI, enabling users to define whether a tool should be made available in specific interactions
- Approval dialogs for individual tool executions
- Permission settings for pre-approving certain safe operations
- Activity logs that show all tool executions with their results

Since tools are typically *model-driven*, it is important to check the quality, safety, and reliability of connected tools and servers.

2.22. Creating MCPs with SDKs

Model Context Protocol contains several open-source SDK’s in the following languages:

- TypeScript
- Python
- Go
- Kotlin
- Swift
- Java
- C #
- Ruby
- Rust
- PHP

Official Docs at: https://modelcontextprotocol.io/docs/sdk

2.23. MCP SDK in Python

For this course we will use the Python SDK with `fastmcp` framework.

Documentation: https://github.com/modelcontextprotocol/python-sdk?tab=readme-ov-file

We recommend using the standard `uv` package installer to manage your python projects:

https://docs.astral.sh/uv/

2.24. MCP Server Pattern (Python)

FastMCP uses Python decorators for simplicity:

```python
from fastmcp import FastMCP

mcp = FastMCP("My Server")

@mcp.tool()
def my_tool(param: str) -> str:
    return f"Result: {param}"
```

2.25. Example: Hello MCP!

```python
from fastmcp import FastMCP

mcp = FastMCP("Demo Server")

# Add a tool
@mcp.tool()
def add(a: int, b: int) -> int:
    """Add two numbers"""
    return a + b

if __name__ == "__main__":
    # Run the server
    mcp.run()
```

2.26. Useful Resources

- **Official Documentation**: https://modelcontextprotocol.io
- **Python SDK**: https://github.com/modelcontextprotocol/python-sdk
- **Community Servers**: 4,774+ examples at https://github.com/modelcontextprotocol/servers
- **MCP Inspector**: Dev testing tool included with SDK https://modelcontextprotocol.io/docs/tools/inspector

2.27. References

1. Guo, Zikang, Benfeng Xu, Chiwei Zhu, Wentao Hong, Xiaorui Wang, and Zhendong Mao. "MCP-AgentBench: Evaluating Real-World Language Agent Performance with MCP-Mediated Tools." arXiv preprint arXiv:2509.09734 (2025).

2. Hou, Xinyi, Yanjie Zhao, Shenao Wang, and Haoyu Wang. "Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions." arXiv preprint arXiv:2503.23278 (2025).

3. Anthropic. "Introducing the Model Context Protocol." November 2024. https://www.anthropic.com/news/model-context-protocol

4. H. Rick. "MCP the USB-C for AI." April 2025. https://medium.com/@richardhightower/how-the-model-context-protocol-is-revolutionizing-ai-integration-48926ce5d823

5. L. Edwin. "Model Context Protocol (MCP): Solution to AI integration bottlenecks." May 2025. https://addepto.com/blog/model-context-protocol-mcp-solution-to-ai-integration-bottlenecks/

6. Model Context Protocol. "Model Context Protocol Servers." GitHub repository. https://github.com/modelcontextprotocol/servers
7. Model Context Protocol. "Example Servers." https://modelcontextprotocol.io/examples
8. Model Context Protocol. "Specification." June 2025. https://modelcontextprotocol.io/specification/2025-06-18
9. Model Context Protocol. "Tools." https://modelcontextprotocol.io/docs/concepts/tools
10. Model Context Protocol. "Introduction." https://modelcontextprotocol.io/introduction
11. punkpeye. "awesome-mcp-servers: A collection of MCP servers." GitHub repository. https://github.com/punkpeye/awesome-mcp-servers

Enabling RAG with Resources and Roots

MODULE 3

After this chapter, you will be able to:

- ✓ Distinguish resources (read-only) from tools (read-write)
- ✓ Build and expose resource URIs for different data types
- ✓ Implement resource discovery with templates
- ✓ Stream real-time updates to connected clients
- ✓ Establish roots for focused context management

✓ Optimize token usage with intelligent caching

3.1. The Data Access Challenge

Diagram illustrating the challenge of isolated data silos and how organizations struggle to connect AI systems to their vast, underutilized data stores across multiple systems and formats.
Figure 7. The Hidden Data Problem
The Challenge: Organizations have massive data stores, but 73% of enterprise data remains invisible to AI systems due to integration complexity.

3.2. Limitations of RAG

While RAG focuses narrowly on retrieving external data to augment LLM responses, MCP standardizes this process while adding critical capabilities like tool integration, security, and interoperability.

- RAG is typically limited to structured content and has trouble navigating structured databases natively
- Custom integration required for each database type or data pipeline
- Struggles with access to real-time APIs or databases
- Vector similarity modes may retrieve irrelevant but similar sounding content
    - Version conflicts or similar documents can cause confusion
- Real time updates are challenging in RAG systems
    - Vector indices can become outdated and require re-indexing

> `Tip`
>
> RAG works best for: static knowledge, historical data, FAQs, policies.

3.3. What Are Resources?

MCP Resources are passive data sources that provide read-only access to information for context. [12]

- **Expose data** to MCP Clients through URI-based addressing
- **Accept parameters** for dynamic content generation based on data
- **Downstream Processing**: selecting relevant portions, searching with embeddings, or passing it all to the model (via client).
- **Return structured data** in JSON or other formats from existing data
- **Support caching** for performance optimization (e.g. customer chatbots)

Examples:

- A customer resource that fetches data based on customer ID, returning profile information without modification capabilities.
- Retrieve documents or data fields stored inside of a data lake for enhanced context
- Read calendar information to retrieve availability

3.4. Traditional RAG vs MCP Resources

Feature	RAG	MCP
Data Type	Static, unstructured (PDFs, docs, wikis)	Dynamic, structured/ unstructured (APIs, DBs, filesystems, SaaS tools)
Retrieval Method	Embedding + vector similarity search	On-demand tool/API invocation
Latency	Low (indexed search)	Medium (depends on API/tool latency)
Security	Data stored in vector DBs (can be encrypted)	No data storage; secure OAuth2 access at runtime
Setup Complexity	Requires chunking, embedding, and indexing	Requires tool schema definition and registration
Ideal Use Cases	Document Q&A, knowledge assistants	Analytics bots, CRM lookups, live status fetch

3.5. How Resources Work

3.5.1. Resource Definition

A resource definition includes:

- `uri`: Unique identifier for the resource
- `name`: The name of the resource
- `title`: Optional human-readable name of the resource for display purposes
- `description`: Optional description

- `mimeType`: Optional MIME type
- `size`: Optional size in bytes

> **Tip**
>
> Optional arguments / parameters are still valuable as they aid in resource discovery by models. Make sure to include descriptions whenever possible!

3.5.2. Understanding MCP Resources

Resources provide structured access to organizational data through URI-based endpoints that AI systems can discover and query.

Core Resource Characteristics:

- **URI-based addressing**: Each resource has a unique identifier (e.g., `file:///path/document.md`)
- **MIME type declarations**: Proper content type handling for different data formats
  - MIME type = two-part identifier used to indicate the nature and format of a document or file [13]
- **Read-only access**: Resources retrieve data without modifying underlying sources
- **AI-friendly discovery**: Applications can explore available data and determine optimal usage

Data Source Integration:

Resources can expose information from any source - databases, file systems, APIs, or external services - making previously siloed data accessible to AI systems through a standardized interface.

3.5.3. Annotations (optional)

Resources, resource templates and content blocks support optional annotations that provide hints to clients about how to use or display the resource:

- `audience`: An array indicating the intended audience(s) for this resource.
  - Valid values are `"user"` and `"assistant"`.
  - For example, `["user", "assistant"]` indicates content useful for both.
- `priority`: A number from `0.0` to `1.0` indicating the importance of this resource.
  - A value of `1` means "most important" (effectively required), while `0` means "least important" (entirely optional).
- `lastModified`: An ISO 8601 formatted timestamp indicating when the resource was last modified (e.g., `"2025-01-12T15:00:58Z"`). [14]

3.6. Resource Discovery Patterns

MCP supports two complementary discovery patterns that enable flexible data access and exploration.

Direct Resources:

- Static URIs pointing to specific, well-defined data sources
- Predictable endpoints for known information
- Example: `calendar://events/2024` retrieves all calendar events for 2024

Resource Templates:

- Dynamic URIs with parameterized patterns for flexible querying
- Enable AI to explore data relationships and hierarchies
- Support complex data navigation through parameter substitution

- Support flexible querying mechanisms across domain contexts
- Example: `travel://activities/{city}/{category}` - returns activities by city and category

[15]

3.6.1. URI Schemes

MCP defines standard URI schemes for resource identification through the following generic format: `[protocol]://[host]/[path]`

Standard Schemes:

- **`https://`**: Web-accessible resources that clients can fetch directly without server
- **`file://`**: Filesystem-like resources (may be virtual filesystems)
    - Supports XDG MIME types like `inode/directory` for non-regular files
- **`git://`**: Git version control integration

Custom Schemes:

- Custom URI schemes must comply with `RFC3986` and follow MCP guidance for proper resource identification.

[16]

> **Tip**
>
> Use custom schemes rather than `https://` when the server must mediate access.

3.6.2. Resource Template Patterns

Template resources enable dynamic, parameterized data access

Template Structure:

```
{
    "uriTemplate": "file://database/tables/{table}/rows/{file_name}",
    "name": "table-row-access",
    "title": "Database Table Row Access",
    "description": "Access specific rows from any database table",
    "mimeType": "application/json"
}
```

3.7. Document Resource Template Example

Different resource types can be accessed by their unique URI identifier

```
{
    "uriTemplate": "file://projects/{project_id}/files/{file_path}",
    "name": "project-documents",
    "title": "Project Document Access",
    "description": "Access project documentation and files by path",
    "mimeType": "text/plain"
}

{
    "uriTemplate": "file://reports/{department}/{file_path}",
    "name": "department-reports",
    "title": "Department Analytics",
    "description": "Get analytics reports filtered by department and time",
    "mimeType": "application/json"
}
```

3.8. Customer Database Template Example

Reflect: Review the templates below and consider what elements you observe.

```
{
    "uriTemplate": "customers://orders/{customer_id}/{status}",
    "name": "customer-orders",
    "title": "Customer Order History",
    "description": "Retrieve customer orders filtered by status (pending, completed,
cancelled)",
    "mimeType": "application/json"
}

{
    "uriTemplate": "customers://support/{customer_id}/tickets",
    "name": "support-tickets",
    "title": "Customer Support Tickets",
    "description": "Access customer support history and active tickets",
    "mimeType": "application/json"
}
```

3.9. Parameter Completion for Customer Support

Dynamic resources support intelligent parameter completion to help discover valid customer identifiers and service options:

Status Code Completion:

- Typing "pen" for `customers://orders/{customer_id}/{status}` might suggest "pending" or "pending_payment"
- Typing "esc" for `customers://support/{customer_id}/tickets` might suggest "escalated" or "escalation_pending"

The system intelligently suggests valid parameter values without requiring exact knowledge of customer identifiers or internal codes, making data discovery more intuitive for AI agents.

3.10. Resources User Interaction Model

Resources offer application-driven flexibility for retrieving, processing, and presenting contextual data.

Common Interaction Patterns:

- **Tree/List Views**: Browse resources using familiar folder-like hierarchical structures
- **Search and Filter**: Find specific resources through query-based discovery
- **Automatic Context**: Include context or smart suggestions based on heuristics or AI selection
- **Manual Selection**: Select individual or bulk resources through user-controlled interfaces

3.10.1. Implementation Flexibility

The MCP protocol provides interface freedom, enabling applications to implement resource discovery through:

- **Resource Pickers**: Preview-enabled selection interfaces
- **Smart Suggestions**: Context-aware recommendations based on current conversation
- **Bulk Operations**: Multi-resource selection and inclusion capabilities
- **Native Integration**: Seamless connection with existing file browsers and data explorers

> ⓘ Note
>
> This design philosophy ensures that resource access patterns can be optimized for specific application needs while maintaining consistency across the MCP ecosystem.

3.11. Resource Capabilities

MCP Servers that support resources MUST declare the resources capability.

MCP resource capabilities also include two optional features:

- `subscribe`: Enables clients to receive notifications of updates / changes to individual resources
  - *e.g. Customer status changes*
- `listChanged`: Allows clients to receive notifications when new resources become available or existing resources are removed from the server
  - *e.g. Additional customers are added to data*

```
{
  "capabilities": {
    "resources": {
      "subscribe": true,
      "listChanged": true
    }
  }
}
```

3.12. Resource Discovery Deep Dive

Operation	Method	Direction	Parameters	Response	Purpose
List Resources	`resources/list`	Client → Server	`cursor` (optional)	Array of resource descriptors with metadata	Discover available resources with pagination
Read Resource	`resources/read`	Client → Server	`uri` (required)	Resource content with metadata and data	Retrieve specific resource contents
List Templates	`resources/templates/list`	Client → Server	None	Array of URI templates with parameters	Discover parameterized resource patterns
Subscribe to Resource	`resources/subscribe`	Client → Server	`uri` (required)	Subscription confirmation	Monitor specific resource for changes
List Changed Notification	`notifications/resources/list_changed`	Server → Client	None	Automatic notification	Alert when available resources change
Resource Updated Notification	`notifications/resources/updated`	Server → Client	`uri`, metadata	Resource change details	Push updates for subscribed resources

3.13. Discovery Workflow

The complete resource discovery process:

1. **List**: Client requests available resources with `resources/list`
2. **Explore**: Client examines templated resource metadata `resources/templates/list`
3. **Read**: Client accesses specific resources using `resources/read`
4. **Subscribe**: Client monitors changes with `resources/subscribe`
5. **Update**: Client receives real-time updates via notifications (individual resources updated or resources list changed)

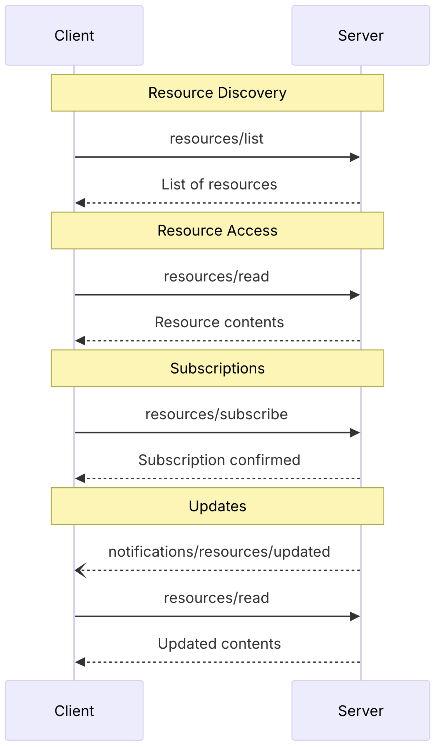


Figure 8. MCP Resource Discovery and Access Flow

3.13.1. Real-Time Resource Subscriptions

Enable AI to receive live data updates:

Subscription Workflow:

1. **Subscribe**: `resources/subscribe` to specific resource URI
2. **Monitor**: Server watches for data changes

3. **Notify**: Push updates to subscribed clients
4. **Unsubscribe**: Clean disconnection when done

Use Cases:

- **Stock prices**: Real-time financial data
- **System monitoring**: Live metrics and alerts
- **Chat systems**: New message notifications
- **IoT data**: Sensor readings and device status

3.14. MCP Resources with Fast MCP

Here's an example Python snippet implementing resource support:

```python
app = Server("example-server")

   @app.list_resources()
   async def list_resources() -> list[types.Resource]:
       return [
           types.Resource(
               uri="file:///logs/app.log",
               name="Application Logs",
               mimeType="text/plain"
           )
       ]

   @app.read_resource()
```

```
    async def read_resource(uri: AnyUrl) -> str:
        if str(uri) == "file:///logs/app.log":
            log_contents = await read_log_file()
            return log_contents
        raise ValueError("Resource not found")
```

3.14.1. Starting Server

```
# Start server
async with stdio_server() as streams:
    await app.run(
        streams[0],
        streams[1],
        app.create_initialization_options()
    )
```

3.15. Best Practices

Follow these best practices when building MCP resources:

Resource Design:

- Use descriptive, intuitive URIs that clearly indicate data source and type
- Include helpful descriptions to guide AI understanding
- Set appropriate MIME types whenever possible
  - _e.g., `text/plain`, `application/json`, `text/markdown`
- Implement resource templates for parameterized access patterns

Dynamic Content:

- Use subscriptions for frequently changing data to enable real-time updates
- Consider pagination for large datasets to manage performance and token usage
- Monitor and optimize resource access patterns for efficiency
- Cache resource contents when appropriate to reduce load times and API costs

[17]

3.16. Security Considerations

Implement comprehensive security measures when exposing resources to protect sensitive data and prevent unauthorized access.

- **Validation**: Validate all Resource URIs and MIME Types
- **Authentication**: Use secure tokens (JWT, OAuth) for client verification
- **Access Controls**: Implement role-based access controls for resource permissions and log access attempts
- **Path validation**: Sanitize file paths to prevent directory traversal attacks
- **Data encryption**: Use TLS/SSL for all data transmission
- **Credential security**: Store credentials securely and never log secrets
- **Rate limiting**: Prevent resource abuse through request throttling
- **Error handling**: Return clear errors without exposing system details (graceful failure) [17]

3.17. Roots

In a complex environment with thousands of potential resources, it’s crucial to guide the AI’s attention to what is currently relevant. MCP addresses this through `Roots`.

- **Roots** define the boundaries of where servers can operate within the filesystem
  - Helps servers understand which directories and files they have access to.
- Roots are a client primitive and are a mechanism for clients to communicate filesystem access boundaries to servers.
  - Roots are a coordination mechanism to communicate intended scope
- Servers can request the list of roots from supporting clients and receive notifications when that list changes.

3.17.1. Roots: Purpose and Function

The primary purpose of roots is to narrow the context for data retrieval.

By focusing the retrieval process on a smaller, more relevant set of resources, the system can:

- **Improve Relevance**: The information retrieved is more likely to be pertinent to the task.
  - e.g. switching between open projects in an IDE
- **Reduce Noise**: Irrelevant files and data are filtered out, preventing them from cluttering the context provided to the LLM.
- **Enhance Performance**: The search space for the retrieval part of RAG is significantly smaller, leading to faster and more efficient searches.
- **Provide Context Awareness**: Roots help the AI understand the boundaries of the current project or workspace.

> ⓘ Important

> The documentation makes a critical point: roots are a coordination mechanism, not a security boundary. Client is still responsible for enforcing security and access control

3.17.2. Root Structure

Roots are exclusively filesystem paths and always use the `file://` URI scheme.

```json
{
  "uri": "file:///Users/agent/travel-planning",
  "name": "Travel Planning Workspace"
}
```

3.18. Roots Message Flow

- Servers can call `roots/list` to retrieve list of roots from clients
- Clients can notify servers via `notifications/roots/list_changed`
  - Client must support `listChanged` and is responsible for notifying server

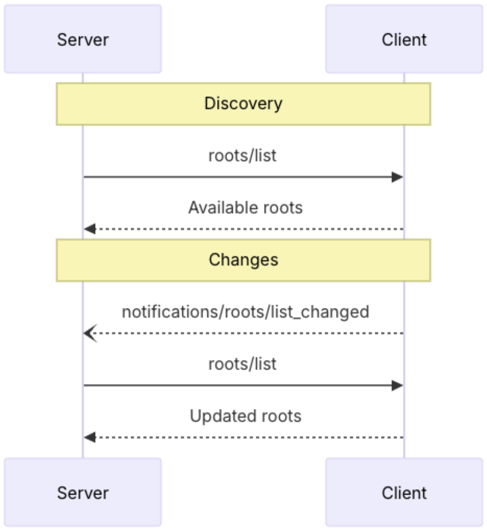


Figure 9. MCP Roots Message Flow

3.18.1. Roots Capability

Roots must be declared as a capability by clients during initialization

```
{
  "capabilities": {
    "roots": {
      "listChanged": true
    }
  }
}
```

Roots (from clients) help servers understand project boundaries, workspace organization, and accessible directories.

The roots list can be updated dynamically as users work with different projects or folders, with servers receiving notifications through roots/list_changed when boundaries change.

> ⓘ Note
>
> Remember that Roots are a Client primitive and part of client specification, which allows customized resource scoping from Host/Client.

3.19. Roots Implementation Guidelines

Client Responsibilities:

- **Consent**: Prompt users for consent before exposing roots to servers
- **User Interface**: Provide clear user interfaces for root management
- **Validation**: Validate root accessibility before exposing
- **Monitoring**: Monitor for root changes and notify servers

Server Responsibilities:

- **Capability Check**: Check for roots capability before usage
- **Change Handling**: Handle root list changes gracefully
- **Scope Adherence**: Respect root boundaries in all operations
- **Caching**: Cache root information appropriately for performance

3.20. Roots Security Considerations

Client Security Requirements:

- **Permissions:** Only expose roots with appropriate user permissions
- **Path Validation**: Validate all root URIs to prevent directory traversal attacks
- **Access controls**: Implement proper authentication, access controls, and authorization mechanisms
- **Monitoring**: Continuously monitor root accessibility and updates/changes

Server Security Guidelines:

- **Graceful degradation**: Handle cases where roots become unavailable
- **Boundary respect**: Respect root boundaries during all operations
- **Path validation**: Validate all file paths against provided roots
- **Error handling**: Return secure errors without exposing system details

> ⓘ Note
>
> Since Roots are a Client primitive, the Security guidelines for Clients are a MUST while those for Servers are recommendations

3.20.1. What We've Learned

✓ **Resources Definition**: Read-only data access pattern

✓ **URI Design**: Creating intuitive resource paths

✓ **Discovery**: Template-based resource exploration

✓ **Real-time Updates**: Notifications and subscriptions

✓ **Roots**: Context-focused data access

✓ **Implementation**: Best practices and security guidelines for resources and roots

Building Patterns with Prompt Templates

MODULE 4

After this chapter, you will be able to:

- ✓ Explain the structure and components of MCP prompt templates
- ✓ Describe how MCP prompt discovery protocols enable client applications
- ✓ Describe common use case patterns for prompt workflows
- ✓ Implement basic MCP prompts with customizable arguments using FastMCP

- ✓ Construct multi-step workflow prompts that embed MCP resources
- ✓ Analyze prompt template designs for effectiveness
- ✓ Integrate MCP prompts into client applications

4.1. The Knowledge Encoding Problem

Challenge: Organizations struggle with inconsistent AI interactions

- **Ad-hoc prompting** leads to unpredictable results
- **Expert knowledge** trapped in individual silos (teams, individuals, etc.)
- **Limited workflow standardization** across teams and tools

Solution: MCP Prompts provide structured, reusable templates for LLM interactions

4.2. Prompts: The UNDERSTAND Layer

In the context of MCP servers, **Prompts** specifically refer to Prompt Templates that provide pre-defined instructions and workflows for LLMs.

The Challenge: Ad-hoc AI interactions can lead to unpredictable results due to the non-deterministic nature of LLMs. [18]

The Solution: Pre-defined workflows and templates reduce the burden of LLM subjectivity and response variability

- By providing specific tasks and domain knowledge, prompt templates can help provide a more structured and repeatable approach at scale.

Goal: Transforming raw user intent and organizational knowledge into structured, context-rich prompts that AI systems can reliably interpret.

> Tip
>
> MCP Prompts act like a shared recipe book - encode the expertise once, use it everywhere.

4.3. Why Use MCP Prompts?

Think of MCP prompts as **smart shortcuts** for AI conversations.

- Instead of typing the same instructions over and over, you create reusable templates to plug into different contexts
- **Reduce Repetitive Work** - Transform time-consuming tasks (code reviews, reports, documentation) into one-click automations
- **Encode Expert Knowledge** - Capture organizational expertise in reusable templates instead of losing it in individual silos
- **Consistent Results** - Replace unpredictable ad-hoc prompting with structured, validated workflows
- **Workflow Intelligence** - Combine scripting flexibility with AI reasoning for complex processes
- **Scale Best Practices** - Share proven approaches across teams rather than reinventing
- **Reduce Context Switching** - Automate predictable patterns so humans focus on creative, strategic work

4.4. Prompt Templates vs Direct Queries

Balance scale comparing direct queries vs prompt templates
Figure 10. Direct Queries vs Prompt Templates

4.4.1. MCP Prompt Specification

The Model Context Protocol (MCP) provides a standardized way for servers to expose prompt templates (structured messages and instructions) to clients. [19]

- Prompts in MCP follow a structured schema with specific components that define how they're discovered, invoked, and executed.
- The protocol uses JSON-RPC for communication between clients and servers

Core Components:

- **Prompt Templates**: Pre-defined instruction patterns with dynamic parameters
- **Discovery Protocol**: Standardized way for MCP clients to find available prompts
- **Parameter System**: Type-safe inputs that customize prompt execution
- **Resource Integration**: Ability to embed contextual data from MCP resources

4.5. How Prompts Work

Prompts allow MCP server authors to provide reusable templates for a domain

Key Characteristics:

- **User-controlled execution** - Require explicit invocation rather than automatic triggering
- **Context-aware processing** - Reference available resources and tools to create comprehensive workflows

- **Parameter completion** - Support argument discovery to help users find valid input values
- **Template-based structure** - Define consistent interaction patterns across applications

> ⓘ Note
>
> MCP transforms prompts from ad-hoc text into **discoverable, parameterized APIs** that clients can invoke programmatically.

4.6. User Interaction Model

Prompts are **user-controlled**, requiring explicit invocation rather than automatic execution.

Core Design Principles:

- **Easy Discovery** - Users can quickly find available prompts
- **Clear Descriptions** - Each prompt explains its purpose and functionality
- **Natural Input** - Argument collection with built-in validation
- **Template Transparency** - Users can see the underlying prompt structure

> Tip
>
> The key is making prompts **discoverable and accessible** while maintaining consistency with your application’s existing user experience.

4.6.1. Prompt Discovery Interfaces

Prompts are designed to show up in your AI application’s interface so you can easily find and use them. [19]

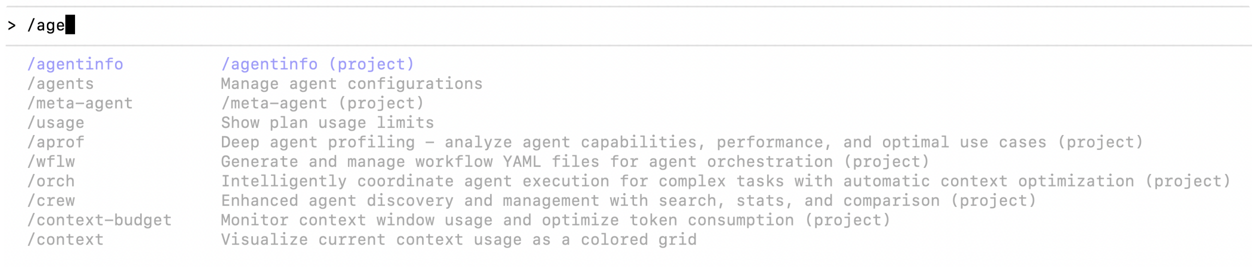
How You’ll See Them:

- Slash commands (like typing `/explain-code`)
- Quick action buttons — UI elements for frequently used prompts
- Context menu options (e.g. right-click menu for relevant prompts)
- Command palette (search bar or keyboard shortcuts)

> **Tip**
>
> It’s like having a playbook of pre-made AI commands you can choose from instead of reinventing the workflow each time.

4.7. Example: Slash Commands in Claude Code

In the example below each of the slash commands (`/command`) maps to a specific `.md` file containing prompt template instructions.



```
> /age

  /agentinfo          /agentinfo (project)
  /agents             Manage agent configurations
  /meta-agent         /meta-agent (project)
  /usage              Show plan usage limits
  /aprof              Deep agent profiling - analyze agent capabilities, performance, and optimal use cases (project)
  /wflw               Generate and manage workflow YAML files for agent orchestration (project)
  /orch               Intelligently coordinate agent execution for complex tasks with automatic context optimization (project)
  /crew               Enhanced agent discovery and management with search, stats, and comparison (project)
  /context-budget     Monitor context window usage and optimize token consumption (project)
  /context            Visualize current context usage as a colored grid
```

Figure 11. Slash Commands in Claude Code - MCP Prompts in Action

4.8. Example Prompt Template: Technical research

```
---
name: research
allowed-tools: webfetch, websearch
description: Conduct comprehensive technical research on a <topic>
---

Create a git commit with message: $ARGUMENTS
**Purpose**: Conduct comprehensive technical research and generate detailed
documentation in project docs/ folders. This command performs deep analysis of
codebases, APIs, systems, and technologies, then synthesizes findings into structured,
actionable research documents.

**Parameters:**

* **topic** (string, required): The research topic or question to investigate
** Can be a technology, system, API, architecture, or comparison
** Be specific for better results

**Usage Examples:**
----
/research <topic> [options]
/research "<detailed-research-request>"
/research --codebase <path>
/research --api <api-name>
/research --compare "<option1> vs <option2>"
----
```

4.8.1. Core Protocol Operations

Method	Purpose	Returns
`prompts/list`	Discover available prompts on server	Array of prompt descriptors with metadata
`prompts/get`	Retrieve specific prompt with parameters	Complete prompt definition with arguments

+

> ⓘ Note

> Like resources, prompts follow the MCP discovery pattern - list available options, then retrieve specific implementations.

4.9. MCP Prompt Protocol

Server Capabilities Declaration

- Servers supporting prompts MUST declare the capability explicitly
  - Capability declaration is required in order for clients to access prompts functionality from servers.

```
{
  "capabilities": {
    "prompts": {
      "listChanged": true
    }
  }
}
```

Core Operations: List prompts, get specific prompts, handle changes

4.10. MCP Prompt Message Flow

1. **Discovery** - Discover the available prompts on server
2. **Prompts** - Retrieve prompts stored on server
3. **Changes** - Reflect updated changes to prompts on server

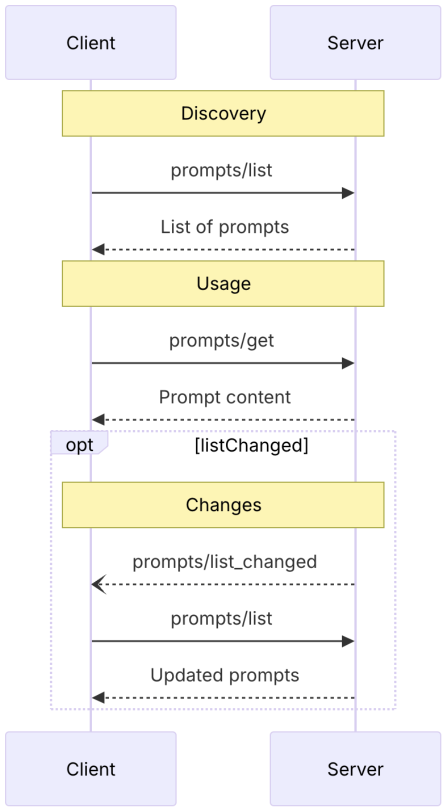


Figure 12. MCP Prompt Message Flow

4.11. Protocol Messages

Listing Available Prompts

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "prompts/list",
  "params": {
    "cursor": "optional-cursor-value"
  }
}
```

ⓘ Note

MCP Clients should **support pagination** for large prompt libraries

4.12. Prompt Response Structure

Generated Prompt with Messages

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "description": "Code review prompt",
    "messages": [{
      "role": "user",
      "content": {
        "type": "text",
        "text": "Please review this Python code:\ndef hello():\n    print('world')"
      }
    }]
  }
}
```

4.13. Prompt Definitions

Prompt definitions in MCP protocol involve the following basic elements:

- `name`: Unique identifier for the prompt
- `title`: Optional human-readable name for display
- `description`: Optional explanation of prompt purpose

- `arguments`: Optional list of parameters for customization

These elements reflect the lookup information that will be requested by clients with `prompts/list`.

> ⓘ Note
>
> Though some elements are "optional", we strongly encourage recommend providing them when possible for optimal use of MCP prompts

4.14. Requesting with `prompts/list` (Client to Server)

A client may send the following message to a server in order to

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "prompts/list",
  "params": {
    "cursor": "optional-cursor-value"
  }
}
```

4.15. Response to `prompts/list` (Server to Client)

A server may respond with the following type of message.

> ⓘ Note
>
> Notice the elements of prompt definition in the fields of the message below and how the prompt is exposed as a structured object.

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "prompts": [
      {
        "name": "code_review",
        "title": "Request Code Review",
        "description": "Asks the LLM to analyze code quality and suggest
improvements",
        "arguments": [
          {
            "name": "code",
            "description": "The code to review",
            "required": true
          }
        ]
      }
    ],
    "nextCursor": "next-page-cursor"
  }
}
```

[19]

4.16. Prompt Messages

Prompts definitions are stored as JSON-RPC

Messages inside of a prompt can contain these fields:

- `roles`: Speaker indicated as either "user" or "assistant"
- `content`: Text content, image content, audio content, embedded resources

All content types in prompt messages support optional annotations for metadata about audience, priority, and modification times.

4.16.1. Text Content

The snippet below will sit inside of the `content` field inside `messages` to specify a text style content.

```
{
  "type": "text",
  "text": "The text content of the message"
}
```

4.16.2. Image Content

Here is an example of an image type content

```
{
  "type": "image",
  "data": "base64-encoded-image-data",
  "mimeType": "image/png"
}
```

> ⓘ Note
>
> The image data MUST be base64-encoded and include a valid MIME type. This enables multi-modal interactions where visual context is important. (Source: MCP Documentation)

4.17. Audio Message

Audio content allows including audio information in messages

```
{
  "type": "audio",
  "data": "base64-encoded-audio-data",
  "mimeType": "audio/wav"
}
```

> ⓘ Note
>
> The audio data MUST be base64-encoded and include a valid MIME type. This enables multi-modal interactions where audio context is important.

4.18. Embedded Resources

Resources are like attachments you can include with your prompts. Instead of just sending instructions, you can bundle actual files, documentation, or data.

- Embedded resources enable prompts to seamlessly incorporate server-managed content like documentation, code samples, or other reference materials directly into the conversation flow. Resources can contain either text or binary (blob) data and MUST include:
- A valid resource URI
- The appropriate MIME type
- Either text content or base64-encoded blob data

Tip

Think of it like attaching your favorite recipes when asking for a meal plan or including your company's style guide when asking for writing help

4.18.1. Embedded Resource Example

```json
{
  "type": "resource",
  "resource": {
    "uri": "resource://example",
    "name": "example",
    "title": "My Example Resource",
    "mimeType": "text/plain",
    "text": "Resource content"
  }
}
```

4.19. Retrieving A Specific Prompt

While `prompts/list` will provide a list of all available prompts, in order to actually leverage one of the prompts we will need to use `prompts/get`

The request from client to server looks like the below:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "prompts/get",
  "params": {
    "name": "code_review",
    "arguments": {
      "code": "def hello():\n    print('world')"
    }
  }
}
```

4.20. `prompts/get` Response

Response from server to client in retrieving the "Code Review prompt"

> **ⓘ Note**
>
> Notice the message role of "user" and content type of text with specific prompt encoded in JSON-RPC message.

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "description": "Code review prompt",
    "messages": [
      {
        "role": "user",
        "content": {
          "type": "text",
          "text": "Please review this Python code:\ndef hello():\n    print('world')"
        }
      }
    ]
  }
}
```

4.21. Argument Properties

Argument name properties

- Unique identifier within the prompt
- Used for parameter passing
- Must be valid identifier string

Argument Description

- Explains the argument’s purpose
- Helps users understand what to provide
- Should be clear and concise

Mandatory Flag

- Boolean flag indicating if argument is mandatory
- `true` : Must be provided by client
- `false` : Optional, may have defaults

4.21.1. Multiple Arguments Example

```
{
  "name": "plan-vacation",
  "description": "Guide through vacation planning process",
  "arguments": [
    {
      "name": "destination",
      "description": "Destination city or country",
      "required": true
    },
    {
      "name": "budget",
      "description": "Budget in USD",
      "required": false
    },
    {
      "name": "duration",
      "description": "Number of days",
      "required": true
    },
    {
      "name": "interests",
      "description": "Activities of interest",
      "required": false
    }
  ]
}
```

4.22. Implementation Considerations

- **Validate prompt arguments** before processing
- **Handle pagination** for large prompt lists
- **Respect capability negotiation** during initialization

Security: Validate all inputs/outputs (client-side) to prevent injection attacks

4.23. Implementing MCP Prompts with FastMCP

FastMCP provides a Pythonic, decorator-based approach to implementing MCP prompts:

1. **Implementation**: Use `@mcp.prompt()` decorator with async functions
2. **Arguments**: Define via Python type hints and docstrings
3. **Customization**: Use parameters to create dynamic, context-aware prompts
4. **Registration**: Automatic via decorators, with schema auto-generation (for MCP protocol)

5. **Validation**: Automatic type checking + custom validation logic

4.24. Example: FastMCP Implementation

`fastmcp` library allows easy declaration of prompt workflows

```python
from fastmcp import FastMCP

# Create a named server
mcp = FastMCP("Code Review")

@mcp.prompt()
def review_code(code: str) -> str:
    """Request a code review"""
    return f"Please review this code:\n\n{code}"

if __name__ == "__main__":
    mcp.run(transport="stdio") # Local stdio transport
```

> ⓘ Note
>
> FastMCP relies on type hints and docstrings to generate primitives definitions.

4.25. Multi-message Prompt Example

```python
from fastmcp import FastMCP
from mcp.types import UserMessage, AssistantMessage

mcp = FastMCP("Debug Assistant")

@mcp.prompt()
def debug_error(error: str) -> list:
    """Create a debugging conversation prompt"""
    return [
        UserMessage("I'm seeing this error:"),
        UserMessage(error),
        AssistantMessage("I'll help debug that. Let me analyze the error.")
    ]
```

4.26. Prompts with Optional Parameters (Arguments)

```python
mcp = FastMCP("Code Analysis with Context")

@mcp.prompt()
def analyze_with_context(code: str, context: str = "", focus_area: str = "") -> str:
    """Analyze code with optional context and focus area"""
    base_prompt = f"Please analyze this code:\n\n{code}"

    if context:
        base_prompt += f"\n\nContext: {context}"

    if focus_area:
        base_prompt += f"\n\nFocus on: {focus_area}"

    return base_prompt
```

4.26.1. Database Schema Example (Embedding Resource)

Transform vague queries into precise operations:

```python
@mcp.prompt()
def sql_query(question: str,
              include_schema: bool = True):
  if include_schema:
    schema = fetch_resource(
      "postgres://db/schema"
    )
    return f"""
      Given this schema:
      {schema}

      Generate SQL for: {question}
    """
```

4.26.2. Prompt Strategy

Table 1. Strategic Prompt Elements

Strategy	Elements
Context Setting	**Role definition** — *"You are a senior security engineer"* **Task clarity** — *"Identify vulnerabilities in code"* **Success criteria** — *"Find OWASP Top 10 issues"*
Input Integration	**Dynamic variables** — `{code}`, `{language}`, `{version}` **Resource URIs** — `file:///schemas/api.json` **User parameters** — *Custom focus areas, thresholds*
Output Specification	**Format requirements** — JSON, Markdown, or structured text **Structure guidelines** — Sections, hierarchy, organization **Quality metrics** — Confidence scores, severity levels
Validation Rules	**Completeness checks** — *All sections present* **Length constraints** — *Min/max token limits* **Content filtering** — *Remove sensitive data*

4.27. Prompts as Multi-Step Workflows

Transform ad-hoc AI conversations across an organization into structured, reusable workflows

Evolution of AI Interactions:

- **Basic Query**: "Create a meal plan" → Generic results
- **Basic Template**: "Create a {cuisine} meal plan" → Customizable output
- **Context-Aware**: "Create a {cuisine} meal plan based on {my_recipe_book}" → Embedded Data for higher-quality results

Impact:

✅ **Consistency**: Same quality results across all users

✅ **Knowledge Capture**: Expert practices encoded permanently

✅ **Efficiency**: time savings on repetitive tasks

✅ **Scale**: Best practices shared organization-wide

4.27.1. Prompt Caching

Avoid regenerating identical prompts to LLM systems (e.g. customer service chatbot)

Caching Strategy Benefits:

- **Performance improvement**: Eliminate expensive prompt regeneration
- **Consistent results**: Same parameters always produce identical prompts
- **Resource optimization**: Reduce CPU usage for complex prompt logic
- **Scalability**: Handle high-frequency requests efficiently

When to Use Caching:

- Complex prompt generation with expensive operations
- Frequently repeated parameter combinations
- High-traffic applications with similar requests
- Resource-intensive template processing

4.28. Enabling Multi-Step Workflows

MCP prompts can orchestrate complex, multi-step processes by combining structured instructions with dynamic resource integration.

Multi-Step Workflow Components:

1. **Parameter Collection**: Developer selects a prompt with parameters
   - e.g., "review-code" with language="python", focus="security"
2. **Resource Assembly**: Server returns both instructional text AND resource references (code files, schemas, documentation)
3. **Context Processing**: Client decides how to handle resources - Applications might choose to select relevant files using embeddings or pattern matching, or pass the entire codebase context directly to the model
4. **AI Execution**: AI receives the code context and generates a detailed review response based on specified review protocol defined in "review-code" prompt workflow

4.28.1. Benefits of Multi-Step workflows

Workflow Benefits:

- **Consistency**: Same quality results across all users
- **Scalability**: Handle complex tasks without manual orchestration
- **Intelligence**: Combine scripting logic with AI reasoning
- **Efficiency**: Automate multi-step processes end-to-end

Tip

> Multi-step workflows transform one-off AI conversations into repeatable organizational processes.

4.29. Multi-Server Workflow Example

Real-world workflows often involve multiple MCP servers working together to accomplish complex tasks that no single server could handle alone.

Scenario: Automated Security Assessment

A comprehensive security review that combines code analysis, vulnerability scanning, and compliance checking across multiple specialized servers.

1. **Static Code Analysis** (Code Server)
   - Get security analysis prompt template, check vulnerabilities
2. **Threat Intelligence** (Security Database Server)
   - Fetch current CVE data and vulnerability patterns
3. **Compliance Validation** (Compliance Server)
   - Apply regulatory compliance prompts
4. **Policy Enforcement** (Documentation Server)
   - Retrieve company security policies
5. **Comprehensive Report Generation**
   - Synthesize findings from all servers

   - Generate executive summary with risk scores

4.30. Multi-Server pattern: Specialized Server Delegation

Different servers handle specific aspects of a workflow, allowing for specialized capabilities and clear separation of concerns.

Database Server:

- **Tools**: `query_database`, `update_records`
- **Resources**: `schema`, `connection_info`
- **Prompts**: `sql-query-builder`

File System Server:

- **Tools**: `read_file`, `write_file`, `list_directory`
- **Resources**: `file_contents`, `directory_structure`
- **Prompts**: `file-analysis`

GitHub Server:

- **Tools**: `create_pr`, `review_code`, `search_repos`
- **Resources**: `repository_info`, `issue_data`
- **Prompts**: `code-review-workflow`

> ⓘ Note
>
> Each server specializes in its domain while contributing to the overall workflow through standardized MCP interfaces.

4.31. Conditional Workflows

Like a **medical diagnosis tree** that asks different questions based on symptoms, we can use user context to adapt focus dynamically and trigger different prompt workflows based on current status of a project/task.

```
Code Metrics Analysis
├─ Complexity > 10?
│  └─ Focus: Refactoring strategies
├─ Duplication > 20%?
│  └─ Focus: DRY principles
├─ Coverage < 80%?
│  └─ Focus: Test generation
└─ All good?
   └─ Focus: Performance tuning
```

4.32. Best Practices for Multi-Server Coordination

1. **Clear Server Boundaries**: Each server handles a specific domain
2. **Unified Prompt Interface**: Prompts abstract server coordination complexity
3. **Error Handling**: Handle individual server failures gracefully
4. **Resource Sharing**: Pass context between servers efficiently
5. **Security Isolation**: Respect permissions and access controls per server

4.33. Security Consideratinos

MCP clients and servers must handle sensitive data appropriately.

- Primary security responsibility and access controls are configured through host

- Clients should implement rate limiting and validate all message content.
- Design for human-in-the-loop framework to ensure that server-initiated AI interactions cannot compromise security or access sensitive data without explicit user consent.
- Check OWASP Top 10 for latest risks related to LLM applications
- Follow OWASP Top 10 for Large Language Model Applications guidance [20]

Anti-patterns to avoid:

- Requesting sensitive data (passwords, API keys) in prompts
- Insufficient validation of user inputs
- No rate limiting or abuse prevention
- Bypassing user consent mechanisms
- Exposing sensitive information in error messages

4.33.1. Prompt Injection Prevention

Multi-Layer Defense Strategy

Like a **medieval castle** with multiple defensive walls:

1. **Outer Wall: Pattern Detection**
   Block known injection phrases instantly

2. **Middle Wall: Input Sanitization**
   Escape dangerous characters and sequences

3. **Inner Wall: Boundary Enforcement**
   Isolate user input from instructions

4. **Keep: Output Validation**
   Verify response contains no injections

⚠ Warning

82% of AI systems are vulnerable to prompt injection attacks. A single successful injection can expose sensitive data, bypass security controls, or hijack system behavior. Security must be built-in, not bolted-on.

4.33.2. Input Validation

Ensure prompt parameters are valid and safe.

Validation Aspect	Description / Example
Type safety	Ensure parameters match expected types
Range validation	Enforce minimum and maximum values
Pattern matching	Verify strings conform to expected formats
Custom validation	Implement domain-specific security checks
Length constraints	Prevent excessively large or empty inputs
Allowed values	Restrict to predefined safe options
Content scanning	Check for dangerous patterns or code
Format verification	Ensure structured data meets requirements
Code analysis	Block potentially unsafe functions (exec, eval)
Language restriction	Only allow supported programming languages
Depth limiting	Prevent resource exhaustion with reasonable bounds
Error handling	Fail safely with clear error messages

4.34. Key Takeaways

MCP Prompts Enable

✅ **User-controlled** prompt selection through standardized discovery

✅ **Multi-modal content** including text, images, and audio

✅ **Resource integration** for context-aware prompt generation

✅ **Type-safe arguments** with validation and auto-completion

✅ **Production deployment** with security, error handling and notifications

4.35. Learn More

Essential Resources

- **MCP Specification**: modelcontextprotocol.io/specification/2025-06-18/server/prompts
- **FastMCP Documentation**: github.com/jlowin/fastmcp
- **Community Examples**: github.com/modelcontextprotocol/servers
- **Protocol Revision**: 2025-06-18 (current)

Enabling Tool-Use on Servers

MODULE 5

- ✓ **Create and implement** a complete MCP tool with proper validation that LLMs can successfully execute
- ✓ **Apply CRUD and workflow patterns** to solve real-world automation scenarios with appropriate error handling
- ✓ **Build production-ready tool servers** with monitoring, documentation, and performance optimization

- ✓ **Demonstrate tool discovery** by enabling AI agents to find and use your tools autonomously

5.1. The Action Gap in AI Systems

MCP's tool-use capability transforms LLMs from passive information sources into active agents that can perform real-world tasks.

Core Problem: LLMs could only answer questions, not take actions.

- Each external integration required custom implementation.

MCP Solution: Provides a universal standard for AI tools - any compliant AI can use any compliant tool.

Key Benefits: * Standardized action protocols * Interoperable tool ecosystem * Real-time system interaction * Reduced integration complexity

5.1.1. Why MCP Tools Matter

MCP tools address one of GenAI's core limitations: LLMs can analyze and advise but cannot act on their recommendations.

Key Benefits:

- **Real-time data access**: Tools connect AI to live systems
  - Resources can also access real-time data, while tools can perform actions or interact with this data (e.g. update data record)
- **Action capability**: Transforms AI from advisor to agent that can execute tasks
  - e.g. `createCalendarEvent` or `sendEmail`
- **Ecosystem growth**: Standardized protocol enables tool reuse across AI applications

- **Safe execution**: Structured schemas and approval gates ensure reliable, controlled operations
    - *e.g. tools are not allowed to perform database row deletions without human approval*

Result: AI moves from generating suggestions to performing actual work - booking meetings, updating databases, and orchestrating workflows across systems.

5.2. MCP Tool Use Cases

The versatility of the MCP tool-use framework allows for an incredibly broad range of applications, from simple utilities to complex, multi-system workflows.

Use cases can range among:

- Personal Productivity
- Real-time Database Management
- Software Development and Devops
- … many more!

5.2.1. Use Case: Personal Productivity and Local Automation

Filesystem Server: MCP Server example for local file operations (custom-defined)

- **Tools**: `findFile`, `writeFile`, `createDirectory`, `listDirectory`, `searchFiles`
- **Resource**: `listFiles`, `readFile`

- **Use Case**: "Summarize all written (.md, .doc, .txt) files in ~/Documents/ProjectX/research and save as summary.md on Desktop"
    - AI uses tools to read files, process content, and write results locally

Calendar & Email Server: Personal productivity automation * **Tools**: `createCalendarEvent`, `findAvailableTime`, `sendEmail`

- **Resources**: `readCalendar`
- **Use Case**: "Find a 30-minute slot next week to meet with Jane and send her an invite"
    - AI uses tools to check availability, book time slot, and send notification

5.2.2. Use Case: Realtime API Access

Weather Data Server: Real-time weather information and alerts

- **Tools**: `get_alerts`, `get_forecast`
- **Resources**: `current_conditions`
- **Use Case**: "I'm traveling to Austin tomorrow - any weather alerts?"
    - AI calls `get_alerts(state='TX')` and `get_forecast(latitude=30.26, longitude=-97.74)`

Database Analytics Server: Live business intelligence and reporting * **Tools**: `queryDatabase`, `getSchema`, `executeAnalysis`

- **Resources**: `tableMetadata`, `queryHistory`
- **Use Case**: "What were our top 5 selling products in Europe last quarter vs same quarter last year?"
    - AI formulates SQL queries, executes via `queryDatabase` tool, and analyzes results
    - Transforms raw data into business insights automatically

5.2.3. Use Case: Software Development and DevOps

Development Lifecycle Integration: MCP tools integrate into development workflows as expert pair programming and DevOps assistance

GitHub Integration Server: Version control and collaboration automation * **Tools**: `createPullRequest`, `getIssueDetails`, `searchRepositories`

- **Resources**: `branchDiff`, `commitHistory`
- **Use Case**: "Review changes on my current branch, create a pull request targeting main, and link to issue #123"
  - AI analyzes git diff, generates PR with appropriate metadata, and automatically links to specified issue
  - Streamlines code review workflow with intelligent automation

5.3. Error Monitoring

Error Monitoring Server: Production issue tracking and analysis * **Tools**: `getLatestErrors`, `analyzeErrorTrends`, `createIncident`

- **Resources**: `errorSummary`, `performanceMetrics`
- **Use Case**: "Check for critical errors in production web-app since last deployment"
  - AI queries Sentry for recent critical issues, analyzes patterns, and provides actionable insights
  - Enables proactive incident response and faster resolution times

5.3.1. Tool Discovery

For a client application and its underlying LLM to use tools, they must first discover what tools an MCP server offers.

1. **Initialization**: After a client connects and completes the initialize handshake, it checks the server’s declared capabilities. If the `tools` capability is present, the client knows it can proceed with tool discovery.

2. `tools/list` **Request**: The client sends a `tools/list` JSON-RPC request (shown below) to the server.
    - This request is simple and typically has no parameters, though it can support pagination for large number of tools

```
{
    "jsonrpc": "2.0",
    "id": 2,
    "method": "tools/list"
}
```

5.3.2. `tools/list` Response

```
{
    "jsonrpc": "2.0",
    "id": 2,
    "result": {
        "tools": [
            {
                "name": "weather_current",
                "title": "Current Weather",
                "description": "Get the current weather for a given location.",
                "inputSchema": { ... }
            },
            {
                "name": "send_email",
                "title": "Send Email",
                "description": "Sends an email to a specified recipient.",
                "inputSchema": { ... }
            }
        ]
    }
}
```

> ⓘ Note
>
> The Server may respond to client with something like the above JSON-RPC message.

5.3.3. Tool Definition

Each object in the `tools` array provides the complete metadata for one tool:

`name`: Unique identifier for the tool (used programmatically)

`title`: Human-readable name displayed in interfaces

`description`: Clear explanation of what the tool does and when to use it

`inputSchema`: JSON Schema defining required and optional parameters

```
{
  name: string;           // Unique identifier for the tool
  description?: string;   // Human-readable description
  inputSchema: {          // JSON Schema for the tool's parameters
    type: "object",
    properties: { ... }  // Tool-specific parameters
  }
}
```

5.3.4. Example Tool Pattern: Shell Command

```
{
  name: "execute_command",
  description: "Run a shell command",
  inputSchema: {
    type: "object",
    properties: {
      command: { type: "string" },
      args: { type: "array", items: { type: "string" } }
    }
  }
}
```

5.3.5. Tool Example: API Integration

```
{
  name: "github_create_issue",
  description: "Create a GitHub issue",
  inputSchema: {
    type: "object",
    properties: {
      title: { type: "string" },
      body: { type: "string" },
      labels: { type: "array", items: { type: "string" } }
    }
  }
}
```

5.4. Dynamic Tool Availability and Notifications

5.4.1. Dynamic Tool Discovery

Tool availability can change based on user permissions, system state, or runtime conditions. MCP handles this through real-time notifications.

Why Dynamic Tools Matter:

- User permissions change (admin vs. regular user)
- System state affects available operations
- Feature flags enable/disable functionality
- Resource availability impacts tool access

5.4.1.1. Notification-Based Updates

1. Server Declaration During initialization, servers can declare dynamic tool capabilities:

```
{
    "capabilities": {
        "tools": {
            "listChanged": true
        }
    }
}
```

5.4.1.2. Change Notifications and Client Refresh

2. Change Notification When tools change, the server sends a notification:

```
{
    "jsonrpc": "2.0",
    "method": "notifications/tools/list_changed"
}
```

3. Client Refresh The client automatically requests an updated tool list:

```
{
    "jsonrpc": "2.0",
    "id": 3,
    "method": "tools/list"
}
```

> Tip
>
> **Efficiency Benefit**: Event-driven updates eliminate the need for polling, ensuring AI always has current tool capabilities.

5.5. MCP Tool Testing

5.6. MCP Tool Testing

The `MCP Inspector` is a useful developer tool for testing and validating MCP tools during development.

Key Testing Features:

- **Tool Discovery**: Automatically lists all available tools, validating the `tools/list` endpoint
- **Schema Validation**: Displays `inputSchema` and descriptions to verify LLM-readable metadata
- **Interactive Testing**: Provides forms to manually test tool execution without writing client code
- **Result Verification**: Shows raw JSON responses to validate output format and content

5.6.1. Testing Workflow

The MCP Inspector enables a systematic, iterative approach to building reliable tools through comprehensive testing.

1. **Connect Inspector** to your MCP server
   - Launch your MCP server using the `npx @modelcontextprotocol/inspector <command>` wrapper.
2. **Navigate to Tools tab** to see discovered tools
   - Verify tools are listed correctly and review descriptions and schemas
3. **Review metadata** for clarity and completeness
4. **Execute test cases** using the interactive forms
   - Test ideal inputs for succesful execution
   - Test edge cases to test limitations
5. ***Iterate and refine** MCP server code and configurations based on results and inspection

💡 Tip

Use the Inspector to catch tool definition issues before AI integration.

5.7. Error Handling with Tools

Proper error handling is a critical part of building robust and reliable MCP tools.

ⓘ Important

When a tool call fails, MCP distinguishes between: Protocol errors (JSON-RPC level) and Tool execution errors (within the tool itself)

- **Protocol errors**: useful when the tool call itself is invalid
- **Tool Execution errors**: when the tool runs but fails
  - MCP handles tool errors through structured responses, not transport-level failures.
  - Even when a tool fails, the JSON-RPC response itself is successful. The error information is contained within the response content.

Error Response Structure:

- **Success Case**: `isError` is omitted or set to `false`
- **Failure Case**: `isError` is set to `true` and the `text` field contains a human-readable error message

5.7.1. Example Error Response

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "result": {
        "content": [
            {
                "type": "text",
                "text": "Deployment failed: Invalid version format. Expected format:
v1.2.3",
                "isError": true
            }
        ]
    }
}
```

> **Tip**
>
> **Best Practice**: Make error messages specific and actionable for both LLMs and human users.

5.8. Client-Side Handling

When receiving tool responses, clients must check the `isError` flag to determine success or failure.

Error Handling Options:

- **Display to user**: Show error message directly
- **Return to LLM**: Allow AI to retry with corrected parameters
- **Log for debugging**: Track failures for system improvement

This standardized error handling enables resilient AI systems that recover gracefully from tool execution failures.

5.8.1. Tools with Fast MCP

```python
# See: chapter04_crud_operations.py
@mcp.tool()
async def create_record(
    collection: str,
    data: dict
) -> dict:
    record_id = generate_uuid()
    await db.insert(collection, data)
    return {"id": record_id}
```

5.8.2. Tool Example

```
# See: chapter04_tool_definition.py
@mcp.tool()
async def deploy_application(
    params: DeploymentParams
) -> dict:
    """Deploy to environment."""
    return {
        "status": "deployed",
        "environment": params.environment
    }
```

5.8.3. Tool Example: Weather

```
@mcp.tool()
async def get_alerts(state: str) -> str:
    """Get weather alerts for a US state.

    Args:
        state: Two-letter US state code (e.g. CA, NY)
    """
    url = f"{NWS_API_BASE}/alerts/active/area/{state}"
    data = await make_nws_request(url)

    if not data or "features" not in data:
        return "Unable to fetch alerts or no alerts found."

    if not data["features"]:
        return "No active alerts for this state."

    alerts = [format_alert(feature) for feature in data["features"]]
    return "\n---\n".join(alerts)
```

5.8.4. Development Best Practices

- **Comprehensive Testing**: Thoroughly validate both success scenarios and failure cases using MCP Inspector
- **Quality Documentation**: Well-crafted description fields enable AI to understand when and how to use your tools effectively

- **Production Readiness**: Implement monitoring, rate limiting, and audit logging for reliable production deployment

> ⓘ Important
>
> **Remember**: Tools bridge the gap between AI advice and AI action—make them robust, well-documented, and secure.

5.8.5. Implementation Workflow: Error Handling

```python
def execute_tool(name, arguments):
    try:
        # Validate input first
        if not validate_input(arguments):
            return error_response("Invalid input parameters")

        # Execute tool logic
        result = perform_operation(arguments)

        return success_response(result)

    except AuthenticationError:
        return error_response("Authentication failed: Invalid credentials")

    except RateLimitError as e:
        return error_response(f"Rate limit exceeded. Try again in {e.retry_after}
seconds")

    except NetworkError:
        return error_response("Network error: Unable to connect to external service")

    except Exception as e:
        # Log the full error internally
        log_error(e)
        # Return user-friendly message
        return error_response("An unexpected error occurred. Please try again later")
```

5.8.6. Tool Testing Checklist

Test Category	Requirements
Definition & Schema	✅ Valid JSON format ✅ Required fields present ✅ Input/output schemas match behavior
Execution Testing	✅ Happy path works correctly ✅ Invalid inputs return proper errors ✅ Edge cases handled gracefully
Documentation	✅ Clear descriptions for LLM use ✅ Accurate annotations ✅ Examples provided
Performance	✅ Completes within timeout ✅ Reasonable response times ✅ Progress updates for long operations

5.9. Essential Tool Development Principles

1. **Clear naming**: Use descriptive names that explain what the tool does
2. **Comprehensive documentation**: Write detailed descriptions with examples showing proper usage
3. **Input validation**: Define strict JSON Schema with proper type checking
4. **Graceful error handling**: Implement comprehensive error handling and validation
5. **Progress reporting**: Show status updates for operations that take time
6. **Focused scope**: Keep each tool atomic - one clear purpose per tool
7. **Structured outputs**: Document exactly what your tool returns

8. **Time management**: Implement timeouts to prevent hanging operations
9. **Resource protection**: Use rate limiting to protect system resources
10. **Complete logging**: Track usage for debugging, monitoring, and improvement

5.10. Key Takeaways

Essential principles for building effective MCP tools that enable AI to take reliable actions in production environments.

- **Tools = Actions**: While resources are for reading data, tools are for taking action on systems and data
- **Clear Definitions**: Invest time writing comprehensive descriptions and schemas—they directly impact how effectively AI uses your tools
- **Output Schemas**: Leverage the new 2025 feature to provide structure and consistency to your tool results
- **Error Handling**: Distinguish between protocol errors (malformed requests) and execution errors (tool operation failures)
- **Annotations**: Use hints to help AI optimize performance, but never rely on them for security boundaries
- **Feedback Loops**: Implement mechanisms for continuous improvement based on user interactions and outcomes

MCP Sampling: Bidirectional AI Integration

MODULE 6

- ✓ Explain how MCP sampling inverts traditional client-server patterns
- ✓ Configure bidirectional sampling between MCP servers and AI clients
- ✓ Implement multi-step reasoning workflows using FastMCP and TypeScript
- ✓ Analyze security implications of bidirectional AI access
- ✓ Design production-ready sampling systems with human oversight

6.1. The Traditional AI Tool Problem

Current Approach:

- Every "smart" tool needs its own LLM integration
- Each tool developer chooses a specific AI model
- Users must trust multiple third-party AI services
- Fragmented ecosystem with vendor lock-in

The Problem:

- 🔒 **Vendor Lock-in**: Tools tied to specific AI providers
- 💰 **Cost Multiplication**: Every tool needs its own AI budget
- 🔐 **Security Concerns**: Data sent to multiple AI services
- 🛠️ **Development Overhead**: Every developer reinvents AI integration

Challenge: How do we enable AI-powered tools without these limitations?

6.2. MCP Sampling enables Two-Way Communication

MCP sampling allows MCP servers to request for LLM completions from clients, enabling bi-directional communication between LLMs and server primitives [3]:

LLM sampling enables tools to:

- **Leverage AI capabilities**: Use the client’s LLM for text generation and analysis
- **Offload complex reasoning**: Let the LLM handle tasks requiring natural language understanding
- **Generate dynamic content**: Create responses, summaries, or transformations based on data
- **Maintain context**: Use the same LLM instance that the user is already interacting with

6.3. Why Sampling Matters

Sampling is a mechanism that allows the MCP ecosystem to scale intelligently.

Key Benefits:

- **Ecosystem Interoperability**: Servers work with any MCP client regardless of underlying model (GPT, Claude, Gemini) — avoids vendor lock-in

Separation of concerns: Servers focus on their domain expertise (data access and actions), while clients manage the user interface and AI reasoning.

- **Enhanced Intelligence**: Beyond raw data, servers provide AI-powered analysis, recommendations, and decision-making
  - e.g. Financial servers provide market sentiment analysis, not just stock data.
- **Security & Trust**: User data processed by trusted client LLM, not third-party server's LLM
- **Context Efficiency**: Self-contained sampling requests optimize context window usage

6.4. Sampling Participants

Three Essential Components:

- **Host**: Container and coordinator (manages security and oversight)
- **User**: Human-in-the-loop approval of AI calls (where appropriate)
- **Client**: Protocol negotiation (handles LLM communication and orchestrates sampling)
- **Server**: Manages server primitives (tools and resources) and requests sampling

[21]

Security: Host provides approval layer for all AI interactions

6.5. How Sampling Works

Sampling Request/Response Process:

1. **Server** initiates sampling request
2. **Client** surfaces the request to user for approval
3. **User** approves/modifies request
4. **Client** forwards approved call to LLM
5. **LLM** returns completion (generation)
6. **Client** reviews LLM response and presents to Host
7. **Host** approves response
8. **Client** forwards to server

6.5.1. Sampling Message Flow

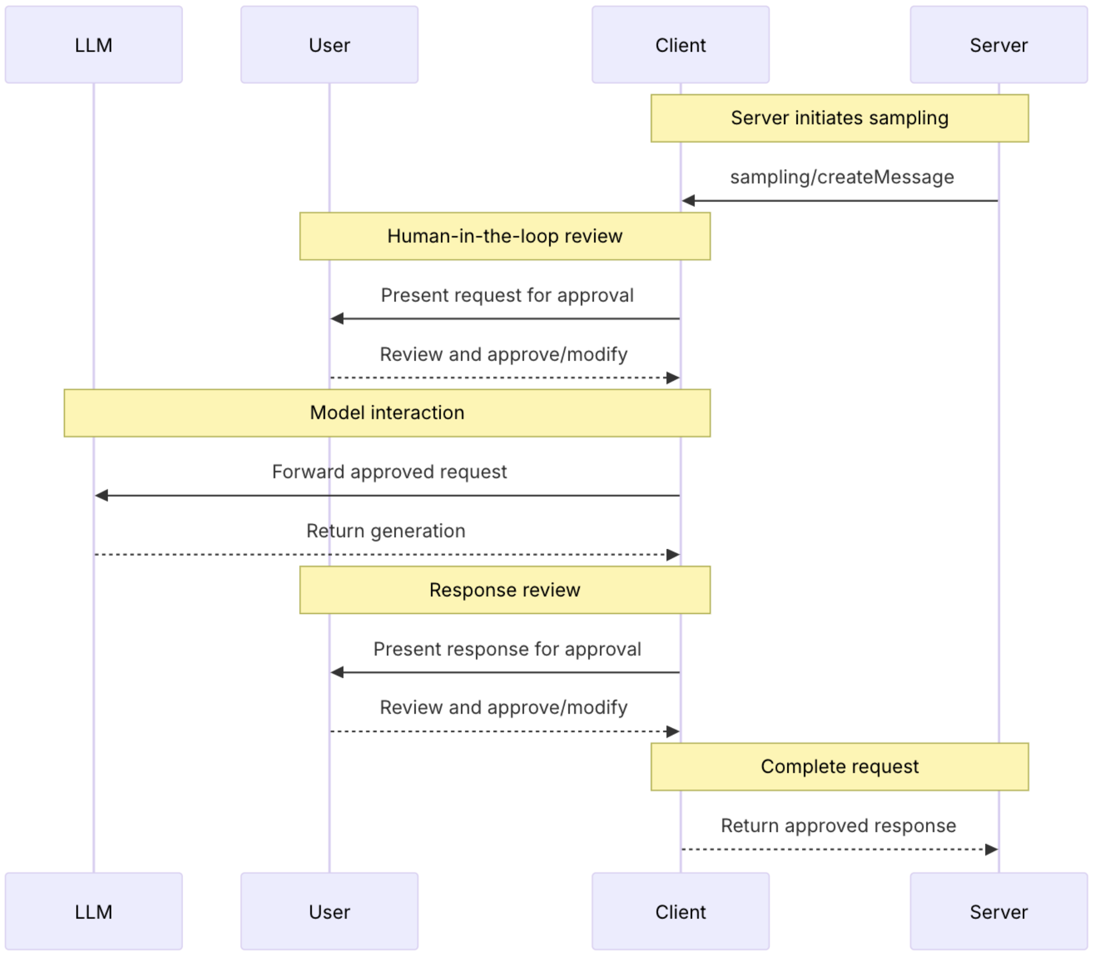


Figure 13. MCP Sampling Message Flow

6.6. Sampling Request Structure

Standard Sampling Request Template sent by an MCP Server:

```
{
  "method": "sampling/createMessage",
  "params": {
    "messages": [{
      "role": "user",
      "content": {"type": "text", "text": "..."}
    }],
    "maxTokens": 1000,
    "temperature": 0.7
  }
}
```

6.7. Sampling Request example

The snippet below shows an example of **parameters** passed in the sampling request

```
{
  "messages": [
    {
      "role": "user",
      "content": "Analyze these flight options and recommend the best choice:\n" +
               "[47 flights with prices, times, airlines, and layovers]\n" +
               "User preferences: morning departure, max 1 layover"
    }
  ],
  "modelPreferences": {
    "hints": [{
      "name": "claude-3-5-sonnet"  // Suggested model
    }],
    "costPriority": 0.3,      // Less concerned about API cost
    "speedPriority": 0.2,     // Can wait for thorough analysis
    "intelligencePriority": 0.9  // Need complex trade-off evaluation
  },
  "systemPrompt": "You are a travel expert helping users find the best flights based
on their preferences",
  "maxTokens": 1500
}
```

6.8. Capability priorities

Priorities help select models based on key characteristics of cost, speed, and model sophistication

ⓘ Note

Servers communicate their requirements using three normalized priority values (range 0.0-1.0):

- `costPriority`: Preference for cost efficiency - higher values favor less expensive models
- `speedPriority`: Preference for response speed - higher values favor faster inference models
- `intelligencePriority`: Preference for model capability - higher values favor more capable models

Trade-off Strategy: Clients use these priorities to select the optimal model from available options, balancing competing requirements based on the server's specified needs.

[22]

6.9. Creating Sampling Responses (Client-Side)

Upon receiving a sampling/complete request, the client's main job is to manage the LLM interaction and return the result. The response sent back to the server contains the LLM-generated content, structured within the MCP format.

```
{
  "jsonrpc": "2.0",
  "id": "request-id-from-server",
  "result": {
    "content": [
      {
        "type": "text",
        "text": "Based on your preferences, I recommend Flight BA2490..."
      }
    ]
  }
}
```

6.9.1. Model Hints

Model Hints provide specific model guidance

- **Flexible Matching**: Hints use substring matching against available model names
- **Priority Ordering**: Multiple hints are evaluated in preference order
- **Provider Flexibility**: Clients can map model hints to equivalent models across different providers
- **Client Authority**: Hints are suggestions only - clients retain final model selection control

```
{
  "hints": [
    { "name": "claude-3-sonnet" }, // Prefer Sonnet-class models
    { "name": "claude" } // Fall back to any Claude model
  ],
  "costPriority": 0.3, // Cost is less important
  "speedPriority": 0.8, // Speed is very important
  "intelligencePriority": 0.5 // Moderate capability needs
}
```

6.10. Human-in-the-Loop

MCP sampling provides built-in human oversight mechanisms as a core design feature:

- **Approval Controls**: Clients require explicit user consent for sampling requests, especially from third-party servers
- **Transparency Features**: Users see exact prompts, data being sent, suggested models, and token limits before approval
- **Response Review**: Clients can display LLM responses to users for final review before forwarding to servers
- **Configuration Options**:
  - Auto-approve trusted servers
  - Require approval for all requests
  - Set model preferences
  - Enable sensitive data redaction
- **Trust Management**: Clear indication of server identity and request purpose builds user confidence

6.11. Sampling with FastMCP: Client Side

ⓘ Note

Clients must implement sampling handlers to process requests. If the client doesn't support sampling, calls to ctx.sample() will fail

```
from fastmcp import Client
from fastmcp.client.sampling import (
    SamplingMessage,
    SamplingParams,
    RequestContext,
)

async def sampling_handler(
    messages: list[SamplingMessage],
    params: SamplingParams,
    context: RequestContext
) -> str:
    # Your LLM integration logic here
    # Extract text from messages and generate a response
    return "Generated response based on the messages"

client = Client(
    "my_mcp_server.py",
    sampling_handler=sampling_handler,
)
```

6.12. FastMCP Sampling (Server Side) - Text Generation

```
from fastmcp import FastMCP, Context
mcp = FastMCP("SamplingDemo")

@mcp.tool
async def generate_summary(content: str, ctx: Context) -> str:
    """Generate a summary of the provided content."""
    prompt = f"Please provide a concise summary of the following content:
\n\n{content}"

    response = await ctx.sample(prompt)
    return response.text
```

> ⓘ Note
>
> `ctx.sample` defines a sampling context based on the guiding `prompt` and awaits the response from the LLM

6.13. Fast MCP Sampling with System Prompts

```
@mcp.tool
async def generate_code_example(concept: str, ctx: Context) -> str:
    """Generate a Python code example for a given concept."""
    response = await ctx.sample(
        messages=f"Write a simple Python code example demonstrating '{concept}'.",
        system_prompt="You are an expert Python programmer. Provide concise, working
code examples without explanations.",
        temperature=0.7,
        max_tokens=300
    )

    code_example = response.text
    return f"```python\n{code_example}\n```"
```

6.14. System Prompts and Specialization

Role-Based AI Analysis:

```
prompts = {
    "technical": "You are a technical expert. Analyze for bugs.",
    "legal": "You are a legal analyst. Check compliance.",
    "security": "You are a security expert. Find vulnerabilities."
}

analysis = await ctx.sample(
    prompt=document_content,
    system_prompt=prompts[analysis_type],
    max_tokens=500,
    temperature=0.3
)
```

6.15. Sampling for Technical Analysis (model preference)

```
@mcp.tool
async def technical_analysis(data: str, ctx: Context) -> str:
    """Perform technical analysis with a reasoning-focused model."""
    response = await ctx.sample(
        messages=f"Analyze this technical data and provide insights: {data}",
        model_preferences=["claude-3-opus", "gpt-4"],  # Prefer reasoning models
        temperature=0.2,  # Low randomness for consistency
        max_tokens=800
    )

    return response.text
```

6.16. Sampling for Sentiment Analysis

```python
from fastmcp import FastMCP, Context

mcp = FastMCP("SamplingDemo")

@mcp.tool
async def analyze_sentiment(text: str, ctx: Context) -> dict:
    """Analyze the sentiment of text using the client's LLM."""
    prompt =
f"""Analyze the sentiment of the following text as positive, negative, or neutral.
    Just output a single word - 'positive', 'negative', or 'neutral'.

    Text to analyze: {text}"""

    # Request LLM analysis
    response = await ctx.sample(prompt)

    # Process the LLM's response
    sentiment = response.text.strip().lower()

    # Map to standard sentiment values
    if "positive" in sentiment:
        sentiment = "positive"
    elif "negative" in sentiment:
        sentiment = "negative"
    else:
        sentiment = "neutral"

    return {"text": text, "sentiment": sentiment}
```

6.17. Sampling: Complex Message Structures

```python
from fastmcp.client.sampling import SamplingMessage

@mcp.tool
async def multi_turn_analysis(user_query: str, context_data: str, ctx: Context) ->
str:
    """Perform analysis using multi-turn conversation structure."""
    messages = [
        SamplingMessage(role="user", content=f"I have this data: {context_data}"),
        SamplingMessage(role="assistant",
content="I can see your data. What would you like me to analyze?"),
        SamplingMessage(role="user", content=user_query)
    ]

    response = await ctx.sample(
        messages=messages,
        system_prompt="You are a data analyst. Provide detailed insights based on the
conversation context.",
        temperature=0.3
    )

    return response.text
```

6.18. Adaptive Workflows

AI-Driven Strategy Selection:

```
@mcp.tool
async def adaptive_processor(task: str, ctx: Context):
    # AI determines best approach
    strategy = await ctx.sample(
        f"Choose best approach for: {task}\n"
        "Options: analysis, summary, translation, coding",
        temperature=0.2
    )

    # Route based on AI decision
    if "analysis" in strategy.text.lower():
        return await ctx.call_tool("deep_analyze", {"data": task})
    elif "summary" in strategy.text.lower():
        return await ctx.call_tool("summarize", {"text": task})
    elif "coding" in strategy.text.lower():
        return await ctx.call_tool("generate_code", {"spec": task})
    else:
        return await ctx.sample(f"Process: {task}")
```

6.19. Learn More

Essential Resources:

- **Official Documentation**: modelcontextprotocol.io

- **FastMCP Framework**: github.com/jlowin/fastmcp
- **Research Paper**: arXiv:2509.09734 (MCP Technical Specification)
- **Community Hub**: github.com/anthropics/mcp
- **Inspector Tool**: @modelcontextprotocol/inspector

Examples: 4,774+ community servers for reference implementations

Agentic AI and Multi-Agent Orchestration

MODULE 7

- ✓ Define agentic AI and contrast it with traditional, request/response systems
- ✓ Build multi-agent solutions using orchestration and role-based frameworks
- ✓ Configure agent communication (MCP, A2A) and resource/tool discovery
- ✓ Apply reasoning patterns (CoT, ReAct, LATS) to decompose and solve complex tasks

- ✓ Design production-ready architectures with security, testing, and rollout strategies

7.1. What is Agentic AI?

Agentic AI is a means of using individual "workers" or instances of AI that act individually or in collaboration to fulfill *goal-directed adaptive behavior.*

Every agent (or group of agents) generally follows a basic cycle:

```
1. GOAL: What needs to be accomplished?
   ↓
2. PLAN: How can I accomplish it?
   ↓
3. EXECUTE: Do the planned actions
   ↓
4. EVALUATE: Did it work?
   ↓
5. ADAPT: If not, adjust plan
   ↓
(Repeat until goal achieved)
```

7.2. What Makes an App "Agentic"?

An agentic application has the following characteristics:

- **Autonomy** - Makes decisions without constant human input
- **Goal-Oriented** - Works toward completing a stated objective
- **Tool Use** - Can interact with external systems via tools
- **Context Awareness** - Maintains and uses relevant context
- **Adaptability** - Adjusts approach based on results
- **Collaboration** - Can work with other agents or humans

7.3. MCP Architecture Components

Before building an agent, understand that agentic behavior emerges from three interacting MCP components:

- **MCP Host** - Houses the core agent logic and reasoning loop
  - Primary LLM orchestrator and user interface
  - Examples: Claude Desktop, VS Code with MCP extension
  - Breaks down user goals into executable plans
- **MCP Client** - Manages communication between Host and Servers
  - Protocol-level JSON-RPC handling
  - One client per external service connection
  - Acts as the "nervous system" connecting brain to capabilities
- **MCP Server** - Provides specialized tools and data access
  - Exposes specific capabilities (filesystem, database, APIs)
  - Examples: filesystem, database, API servers, research tools
  - Multiple servers can run simultaneously

7.3.1. From Tools to Agents

1. **Basic Tool Server**
   "Here are my tools. Call them when needed."
   (Static API endpoints)

2. **Smart Tool Server**
   "Here are my tools. I'll validate inputs and guide usage."
   (Self-documenting with validation)

3. **Agentic Application**
   "Tell me your goal. I'll plan and execute using available tools/resources."
   (Goal-oriented reasoning with tools and data)

Key difference: Agentic applications add an LLM reasoning layer that transforms goals into coordinated tool usage.

7.3.2. Why This Matters

Traditional Programming	Agentic Approach
You must know exact function names	You describe what you want
You must provide exact parameters	Agent interprets your intent
You must handle every edge case	Agent handles coordination
You must coordinate multiple steps	Agent adapts to situations

7.4. Understanding Agentic Applications

Agentic applications follow a **Sense** → **Think** → **Act** loop to achieve goals. MCP provides standardized primitives for each phase:

Sense (Gathering Context)

- Use `resources/list` to discover available information (files, databases, APIs)
- Use `resources/read` to access specific content needed for decision-making
- Build situational awareness before planning actions

Think (Planning and Reasoning)

- LLM processes gathered context to formulate execution plans
- Complex reasoning uses MCP's Sampling primitive for sub-tasks
- Plans typically output as sequences of tool calls

Act (Executing the Plan)

- Execute plans through MCP tool calls (`writeFile`, `sendEmail`, `queryDatabase`)
- Use Elicitation primitive to request user clarification when needed
- Maintain feedback loop to adjust actions based on results

7.4.1. Simple Agentic Example

```
result = await agent.accomplish("Find all files larger than 1MB on my system")
```

How the agent works:

1. **Discovers** the `list_files` tool from connected MCP server
2. **Calls** the tool with "/" directory parameter
3. **Filters** results for files larger than 1MB
4. **Formats** the answer in human-readable form
5. **Returns** the complete result to the user

7.5. Orchestration Patterns and Framework Comparison

7.5.1. Sequential Orchestration

Agents process tasks in defined order, each building on previous outputs

```
Data Collector

↓

Analyzer

↓
```

```
Report Generator

↓

Reviewer

↓

Publisher
```

7.5.2. Parallel Orchestration

Agents process tasks in defined order, each building on previous outputs

Multiple agents work simultaneously on independent subtasks

```
           ┌→ Market Analyst ─┐
Query ────┼→ Tech Analyst   ──┼→ Synthesizer → Report
           └→ Risk Analyst   ──┘
```

7.5.3. Hierarchical Orchestration

Manager agents coordinate teams of specialized workers

```
Project Manager
├── Development Team Lead
│   ├── Frontend Agent
│   ├── Backend Agent
│   └── Database Agent
└── QA Team Lead
    ├── Test Generator
    └── Test Executor
```

7.5.4. Framework Performance Comparison

Framework	**Architecture**	**Strengths**	**Best Use Cases**	**Performance**
LangGraph	Graph-based	Fine-grained control, visual debugging, checkpoints	Complex workflows with branching logic	Lowest latency
CrewAI	Role-based	Intuitive setup, team dynamics, autonomous delegation	Collaborative tasks requiring role separation	Optimal for multi-agent
AutoGen	Conversational	Natural interaction, dynamic composition, flexibility	Research, prototyping, human-in-loop	Good for dynamic scenarios
Google ADK	Modular	Production-ready, evaluation tools, enterprise features	Enterprise deployment at scale	Balanced performance
Semantic Kernel	Plugin-based	Enterprise integration, Microsoft ecosystem	Microsoft-centric environments	Scalable architecture

7.6. Understanding Where The Agent Will "Live"

Before building an agent, we need to understand where it will run.

In MCP architecture, agents "live" in three possible places:

1. Inside an MCP Client (Most Common)
2. Agent inside of MCP Server
3. Standalone Application

Pattern	Best For	Example
Agent in Client	Interactive tools, IDE integration	Claude Desktop, Cursor
Agent as Server	Reusable agent services, microservices	Research agent, code reviewer
Standalone	Complete applications, automation	CI/CD bot, monitoring agent

7.7. Integrating Tools, LLMs, and Resources

The true power of agentic applications emerges from seamlessly integrating LLM reasoning with sensing (Resources), thinking (Prompts) and acting (Tools). MCP primitives work together to create this synergy.

Building an agentic application with MCP follows a clear, structured process. Let’s outline the steps to create a simple agent that can organize files.

1. Step 1: Define the Goal and Capabilities
2. Step 2: Build or Connect the MCP Server
3. Step 3: Build the Host Application (The Agent’s Core Logic)
4. Step 4: Connect the Host to the Server

7.7.1. Step 1: Define the Goal and Capabilities

- **Goal**: The agent will take a high-level command like "Organize my downloaded documents from this quarter" and execute it.
- **Capabilities Needed**: It needs to be able to list files, read file contents (to determine their topic), create directories, and move files.
  - Based on the allowed tools (filesystem access, bash/unix capabilities, etc), we can plan to use the tools to interact with the file system.

7.7.2. Step 2: Build or Connect the MCP Server

- Implement an MCP server that exposes the necessary tools: `listDirectory`, `readFile`, `createDirectory`, and `moveFile`.
- The official MCP Filesystem Server is a perfect example to use or adapt.
- This server provides the agent’s "skills" for file management.

7.7.3. Step 3: Build the Host Application (The Agent’s Core Logic)

- This is the main program that runs the **Sense** → **Think** → **Act** loop
- Contains the LLM reasoning engine and orchestration logic
- Manages the conversation flow and user interactions
- The Host will orchestrate the necessary number of clients to connect to each of the MCP servers to perform the task at hand (e.g. filesystem manipulations)

7.7.4. Step 4: Connect the Host to the Server

- The Host application uses an MCP Client library to establish the connection to the Filesystem Server
- This links the agent's "brain" (the Host) to its "skills" (the Server)
- Enables dynamic tool discovery and execution

7.8. Basic Agent Logic

```
class Agent:
    async def act(self, observation):
        context = self.memory.recall()
        action = await self.model.reason(
            observation, context)
        return action
```

7.8.1. Pattern 1: Planning Agents

Use planning agents for complex tasks, multi-step workflows, and tasks requiring coordination.

```
from fastmcp import FastMCP, Context

mcp = FastMCP("Planning Agent")

@mcp.tool()
async def create_plan(goal: str, ctx: Context) -> dict:
    """
    Agent creates a multi-step plan to achieve a goal
    """
    await ctx.info(f"Planning how to achieve: {goal}")

    # Use LLM to create plan
    plan = await ctx.sample(
        messages=[{
            "role": "user",
            "content": f"""Create a detailed plan to accomplish this goal: {goal}

            Return a JSON plan with these fields:
            - steps: Array of step objects
            - dependencies: Which steps depend on others
            - estimated_time: Time estimate for each step
            """
        }],
        max_tokens=2000
    )

    return plan
```

7.8.2. Execute Step in Plan

```
@mcp.tool()
async def execute_step(step_id: int, plan: dict, ctx: Context) -> dict:
    """
    Execute a single step from the plan
    """
    step = plan["steps"][step_id]
    await ctx.info(f"Executing step {step_id}: {step['description']}")

    # Execute the step (may call other tools)
    result = await ctx.sample(
        messages=[{
            "role": "user",
            "content": f"Execute this step: {step['description']}"
        }],
        max_tokens=1000
    )

    return {
        "step_id": step_id,
        "status": "completed",
        "result": result
    }
```

7.9. Agent Communication and Orchestration Protocols

As agentic applications grow in complexity, multiple agents often need to collaborate to achieve goals. This requires standardized protocols for agent communication and orchestration.

Orchestration enables specialized agents to collaborate on complex tasks.

- Task decomposition - breaking down complex task into subtasks
- Resource allocation - assigning specialized agents to different components of the work
- Agent coordination - managing dependencies and handoffs between agents
- Result synthesis - combining outputs into final deliverable

7.10. Modular Orchestration Design

When using multiple agents, it is generally a good practice to delineate functions of individual agents

```
# Single-purpose agents
class ValidationAgent:
    async def validate(data): ...
        ### Validation Agent logic

# Orchestrator composes them
class Orchestrator:
    async def process(source):
        valid = await validate(source)
        result = await transform(valid)
        ### Orchestrator Agent Logic
```

7.11. Agent Implementation: File Organizer

```python
def run_file_organizer_agent(user_goal):
    # SENSE: Discover capabilities from the connected Filesystem Server
    tools = mcp_client.tools.list()

    # THINK: Formulate an initial plan with the LLM
    initial_prompt = f"The user wants to: '{user_goal}'. The available tools are: {to
ols}. Create a step-by-step plan to achieve this."
    plan = llm.create_plan(initial_prompt)

    # ACT: Execute the plan
    while not plan.is_complete():
        next_action = plan.get_next_action()

        if next_action.type == 'tool_call':
            # Get user approval for file operations
            if user_approves(f"Agent wants to: {next_action.name}
({next_action.args}). Approve?"):
                result = mcp_client.tools.call(next_action.name, next_action.args)
                # SENSE again: Feed the result back to the LLM to update the plan
                plan = llm.update_plan(plan, result)
            else:
                print("Action denied by user. Aborting.")
                break
        elif next_action.type == 'elicit_information':
            # Ask user for clarification
            user_response = ask_user(next_action.question)
            plan = llm.update_plan(plan, user_response)
        elif next_action.type == 'final_response':
            print(f"Agent finished: {next_action.message}")
            plan.mark_complete()
```

7.12. Testing and Debugging Agentic Applications

Debugging and testing in agentic systems is more complex than a traditional application because it involves multiple, independent

components and non-deterministic LLM behavior.

A systematic approach is required, involving:

- **Component-level testing** to test servers, hosts, clients individually
- **Integration testing** to test connection and protocol logic between components
- **Deployment and End-to-end testing**:
- **Debugging** of each component and connection
- **Continuous Monitoring** after individual components and protocol have been connected

7.12.1. Component-Level Testing

- **Test the Server**: Before integrating, test the MCP server in isolation.
  - The MCP Inspector is the essential tool for this. Its "Tools" tab allows you to manually call your tools with custom inputs and inspect the raw output.
  - This validates that your server’s core capabilities are working correctly before introducing the complexity of an LLM.
- **Test the Host Logic**: Unit test your agent’s core loop.
  - Mock the MCP client with an LLM API.
  - Provide a fake tool call request from the LLM and verify that your Host correctly calls the mock MCP client.
  - Provide a fake tool result from the client and verify the Host correctly updates the LLM.

7.13. Integration and End-to-End Testing

Monitor the Protocol Layer

- **Use MCP Inspector**: Launch your server with the Inspector to monitor the live stream of JSON-RPC messages between your Host/Client and the Server.

- **Monitor message flow**: View what tools/call requests the agent is making and what results the server is returning.

Log the Reasoning Layer

- **Record LLM interactions**: In your Host application, log all interactions with the LLM.
- **Track prompts and plans**: Record the exact prompts you are sending (including the context from tools and resources) and the exact plans the LLM is generating.
- **Debug reasoning issues**: Often, unexpected agent behavior is due to a misunderstanding in the prompt or the LLM's plan, not a bug in the tool itself.

> Tip
>
> Common issues arise at the seams between components: malformed `inputSchema` causing bad arguments, unexpected tool results confusing the LLM's next step, or silent tool failures that the Host doesn't report back to the LLM.

7.14. Testing Approach

- Unit test individual agents for logic, error handling, and edge cases
- Integration test multi-agent workflows and orchestration patterns
- Load test at scale to measure concurrency, latency, and resource usage

> Tip
>
> Use mocks for external APIs in unit tests, simulate real-world scenarios in integration tests, and automate load tests for continuous performance monitoring.

7.15. Security Considerations

- **Authentication and authorization**
  - Ensures only trusted users and agents can access or perform actions, preventing unauthorized access.
- **Rate limiting per agent**
  - Protects the system from abuse or denial-of-service attacks by limiting how often each agent can make requests.
- **Sandboxed execution**
  - Isolates agent code to prevent it from affecting the host system or accessing sensitive resources.
- **Output validation**
  - Prevents malicious or malformed data from being processed or returned, reducing security vulnerabilities.

Caution

Always enforce least privilege for tools and carefully validate tool outputs before acting on them.

7.16. Security Pattern

```python
async def execute_task(agent_id, task, creds):
    agent = await authenticate(agent_id, creds)
    if not can_execute(agent, task):
        raise PermissionError()
    return await sandbox_execute(agent, task)
```

7.17. Future Directions

MCP enables increasingly powerful agentic applications:

- **Proactive Agents**: MCP’s notification system lets agents monitor resources and respond to changes automatically. When an error log updates, an agent can analyze the issue and suggest fixes without being asked.
- **Creative and Physical Tasks**: As MCP expands to tools like Figma and Blender, agents will handle complex design work and even control physical devices like 3D printers.
- **Ecosystem Approach**: Users will connect multiple MCP servers to create custom "super-agents" with capabilities from many sources - the true "USB-C for AI" vision.

7.18. Deployment Patterns

Gradual Rollout Strategy:

- **Shadow Mode**: Agents observe and log actions while humans remain in full control
  - e.g. Agent logs scheduling recommendations, but staff schedules appointments manually.
- **Assisted Mode**: Agents suggest actions or next steps, but humans make final decisions
  - e.g. Agent proposes draft emails or responses, user reviews and sends.
- **Supervised Autonomy**: Agents execute actions with human oversight and approval checkpoints
  - e.g. Agent processes invoices, but supervisor must approve before payment.
- **Full Autonomy**: Agents operate independently with minimal or no human intervention
  - e.g. Agent automatically routes support tickets and triggers workflows end-to-end.

7.19. Challenges

- **Hallucination**
  - Agents may generate plausible but incorrect information
  - Mitigations: verifier agents, retrieval augmentation, human-in-the-loop review
- **Context limits**
  - LLMs have finite context windows, restricting long-term reasoning
  - Mitigations: hierarchical memory, chunking, retrieval strategies
- **Error cascades**
  - Mistakes by one agent can propagate through workflows
  - Mitigations: circuit breakers, retries, fallback mechanisms
- **Security**
  - Agents may access or expose sensitive data
  - Mitigations: least privilege, sandboxing, output validation

7.20. Future Evolution

- **Causal reasoning and world models**
  - Enable agents to understand cause-and-effect, improving planning, prediction, and reliability.
- **Multi-modal intelligence integration**
  - Allows agents to process and reason across text, vision, audio, and sensor data for richer, more accurate decisions.
- **Distributed federated memory**
  - Supports scalable, privacy-preserving knowledge sharing across agents and organizations.
- **Emergent collective intelligence**
  - Enables agent swarms to solve complex problems collaboratively, increasing adaptability and robustness.

Details

Next-Generation Agent Capabilities (2025-2028):

- **Causal Reasoning and World Models**
  - Agents will develop sophisticated understanding of cause-and-effect relationships
  - Implementation of internal world models for better prediction and planning
  - Integration with causal inference frameworks for robust decision-making
  - Timeline: Initial implementations by 2026, mainstream adoption by 2028
- **Multi-Modal Intelligence Integration**
  - Seamless integration of vision, audio, text, and sensor data processing
  - Unified multi-modal reasoning for complex real-world environments
  - Enhanced human-agent interaction through natural multi-modal interfaces
  - Timeline: Advanced prototypes in 2025, production systems by 2027
- **Distributed and Federated Memory Systems**
  - Scalable memory architectures supporting massive multi-agent deployments
  - Privacy-preserving federated learning across organizational boundaries
  - Blockchain-based knowledge provenance and verification systems
  - Timeline: Research phase through 2026, early implementations by 2028
- **Neuromorphic and Edge Computing Integration**
  - Ultra-low-power agent deployments on neuromorphic hardware
  - Real-time processing capabilities for autonomous vehicles and robotics
  - Distributed edge intelligence with local decision-making autonomy
  - Timeline: Hardware availability 2025-2026, agent integration by 2027-2028

7.21. Key Takeaways

- Agentic AI shifts from answers to actions

- Multi-agent orchestration enables complex workflows
- Standard protocols ensure interoperability
- Production success requires careful architecture

7.22. Learn More

- MCP Documentation: modelcontextprotocol.io [3]
- Framework benchmarks: arXiv:2308.03688 [23]
- Enterprise case studies and patterns
- Implementation best practices

7.23. References

1. Anthropic. "Introducing the Model Context Protocol." November 2024.
2. Google. "Agent-to-Agent Protocol Specification." Cloud NEXT, 2025.
3. LangChain. "LangGraph: Graph-Based Agent Orchestration." Documentation, 2025.
4. H. Rick. "MCP the USB-C for AI." Medium, April 2025.
5. Yao et al. "ReAct: Synergizing Reasoning and Acting." ICLR, 2023.
6. Hou, et al. "Model Context Protocol (MCP)." arXiv:2503.23278, 2025.
7. "Language Agent Tree Search: 2x Performance over ReAct." arXiv, 2025.
8. "AgentBench: Evaluating LLMs as Agents." arXiv:2308.03688, 2025.
9. "WebArena: Real-World Web Agent Testing." Zhou et al., 2025.
10. "Clinical Multi-Agent Systems: Real-World Impact Study." Journal of Medical AI, 2025.

11. Multi-Agent Systems. "Debate Patterns for Consensus." 2025.

12. CrewAI. "Performance Benchmarks for Team-Based AI." 2025.

---

1. Guo, Zikang, et al. "MCP-AgentBench: Evaluating Real-World Language Agent Performance with MCP-Mediated Tools." arXiv preprint arXiv:2509.09734 (2025).
2. L. Edwin, "Model Context Protocol (MCP): Solution to AI integration bottlenecks." https://addepto.com/blog/model-context-protocol-mcp-solution-to-ai-integration-bottlenecks/, May 2025.
3. Anthropic. "Introducing the Model Context Protocol." https://www.anthropic.com/news/model-context-protocol, November 2024.
4. H. Rick, "MCP the USB-C for AI." https://medium.com/@richardhightower/how-the-model-context-protocol-is-revolutionizing-ai-integration-48926ce5d823, April 2025.
5. MCP Community Registry. "MCP Servers Directory." December 2024.
6. Enterprise AI Report. "MCP Adoption in Large Organizations." 2024.
7. Performance Study. "MCP Caching Efficiency Analysis." 2024.
8. The 4 most critical aspects of model context protocol (MCP) for developers building AI-native architectures . https://www.techradar.com/pro/the-4-most-critical-aspects-of-model-context-protocol-mcp-for-developers-building-ai-native-architectures
9. Model Context Protocol. "Client Concepts." https://modelcontextprotocol.io/docs/learn/client-concepts
10. Model Context Protocol. "Architecture." https://modelcontextprotocol.io/docs/learn/architecture
11. JSON-RPC 2.0 Specification. https://www.jsonrpc.org/specification
12. Model Context Protocol (MCP) Documentation - Server Concepts. https://modelcontextprotocol.io/docs/learn/server-concepts
13. MIME types - MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/MIME_types
14. Model Context Protocol - Resource Specifications. https://modelcontextprotocol.io/specification/2025-06-18/server/resources
15. MCP Server Documentation Resources. https://modelcontextprotocol.io/docs/learn/server-concepts

16. RFC 3986: Uniform Resource Identifier (URI): Generic Syntax. https://datatracker.ietf.org/doc/html/rfc3986
17. Model Context Protocol - Resources Concepts. https://modelcontextprotocol.info/docs/concepts/resources/
18. Atil, B., Aykent, S., Chittams, A., Fu, L., Passonneau, R. J., Radcliffe, et al. (2024). Non-Determinism of "Deterministic" LLM Settings. https://arxiv.org/html/2408.04667v5
19. Model Context Protocol Specification. "Server Features - Prompts." https://modelcontextprotocol.io/specification/2025-06-18/server/prompts
20. OWASP Top 10 for Large Language Model Applications. https://owasp.org/www-project-top-10-for-large-language-model-applications
21. Hou, et al. "Model Context Protocol (MCP)." arXiv:2503.23278, 2025.
22. Model Context Protocol. "Client Sampling - Model Hints." June 18, 2025. https://modelcontextprotocol.io/specification/2025-06-18/client/sampling
23. AgentBench: Evaluating LLMs as Agents. arXiv:2308.03688, 2025.