Claude MCP 완벽 가이드

Model Context Protocol의 모든 것

개념부터 고급 개발까지 단계별 학습서

목차

기초

1. 소개

인공지능 기술의 발전과 함께 대형 언어 모델(LLM)은 현대 소프트웨어 개발 및 사용자 경험의 핵심 요소로 자리 잡았습니다. 그러나 이러한 모델들의 능력이 발전함에도 불구하고, 외부 데이터 소스와의 연결 없이는 그 잠재력을 완전히 발휘하기 어렵습니다. 이런 문제를 해결하기 위해 Anthropic은 Model Context Protocol(MCP)이라는 혁신적인 오픈 프로토콜을 개발했습니다.

> MCP는 AI 모델과 외부 시스템 간의 표준화된 통신 방법을 제공하는 프로토콜입니다. 이는 마치 컴퓨터 하드웨어의 USB-C 포트와 같은 역할을 합니다. USB-C가 다양한 주변장치와 액세서리를 연결하는 표준화된 방식을 제공하듯, MCP는 AI 모델이 다양한 데이터 소스와 도구에 연결하는 표준화된 방식을 제공합니다.

이 전자책에서는 Claude MCP의 기본 개념부터 시작해 실제 개발 및 구현 방법까지 단계별로 알아볼 것입니다. 기초 개념부터 시작하여 MCP 서버와 클라이언트 개발, 실행 및 디버깅, 그리고 고급 개발 기법까지 포괄적으로 다룰 예정입니다.

먼저 MCP가 무엇인지, 왜 중요한지, 그리고 어떤 문제를 해결하는지 살펴보겠습니다.

MCP의 배경

MCP의 필요성

대형 언어 모델(LLM)은 놀라운 능력을 보여주지만, 몇 가지 중요한 제약이 있습니다:

- **최신 정보에 대한 접근 제한**: 지식 단절(Knowledge Cutoff)
- **사용자의 특정 데이터에 대한 접근 부재**
- **외부 시스템과의 상호작용 능력 부족**
- **시스템마다 다른 통합 방식**으로 인한 개발 복잡성

MCP는 이러한 제약을 해결하기 위해 설계되었습니다. 이 프로토콜을 통해 AI 모델은:

- 실시간 데이터에 접근할 수 있습니다
- 사용자 특정 문서, 파일, 데이터베이스를 검색할 수 있습니다
- 외부 API 및 서비스와 상호작용할 수 있습니다
- 표준화된 방식으로 다양한 도구를 사용할 수 있습니다

MCP와 API의 차이점

MCP는 일반적인 API와는 다른 접근 방식을 취합니다:

MCP와 일반 API 비교

특성	일반 API	MCP
통신 방향	주로 단방향(요청-응답)	양방향 통신
데이터 접근	특정 엔드포인트로 제한	다양한 리소스와 도구에 대한 접근
컨텍스트 인식	제한적	풍부한 컨텍스트 공유 가능
확장성	서비스별 확장	표준화된 방식으로 확장

참고: MCP는 API를 대체하는 것이 아니라, AI 모델과 데이터 소스 간의 더 풍부한 상호작용을 가능하게 하는 상위 프로토콜입니다.

기초

2. 기본 개념

Model Context Protocol(MCP)은 복잡한 개념처럼 보일 수 있지만, 그 핵심은 간단합니다: AI 모델과 외부 시스템 간의 표준화된 통신 방법을 제공하는 것입니다. 이 섹션에서는 MCP의 기본 개념과 구성 요소에 대해 알아보겠습니다.

2.1. MCP 아키텍처

MCP는 기본적으로 클라이언트-서버 아키텍처를 따릅니다:

- **MCP 호스트(Host)**: Claude 데스크톱, IDE, 또는 AI 도구와 같은 프로그램으로, MCP를 통해 데이터에 접근하고자 합니다.
- **MCP 클라이언트(Client)**: 호스트 애플리케이션 내부에서 동작하며, 서버와 1:1 연결을 유지합니다.
- **MCP 서버(Server)**: 표준화된 프로토콜을 통해 특정 기능을 노출하는 경량 프로그램입니다.
- **로컬 데이터 소스**: 컴퓨터의 파일, 데이터베이스, 서비스 등 MCP 서버가 안전하게 접근할 수 있는 리소스입니다.
- **원격 서비스**: 인터넷을 통해 사용 가능한 외부 시스템(API 등)으로, MCP 서버가 연결할 수 있습니다.

이러한 구성 요소들의 관계를 다이어그램으로 표현하면 다음과 같습니다:

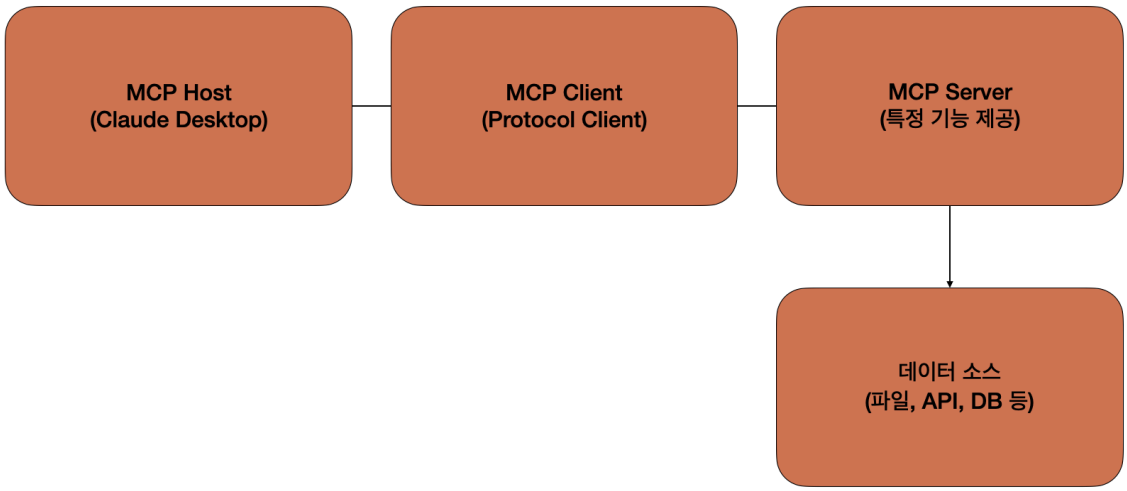

그림 1: MCP 아키텍처 구성 요소

2.2. 프로토콜 세부사항

MCP는 구조화된 메시지를 교환하는 JSON-RPC 2.0 기반 프로토콜입니다. 주요 프로토콜 요소는 다음과 같습니다:

메시지 유형

- **요청(Request)**: 상대방으로부터 응답을 기대하는 메시지
- **결과(Result)**: 요청에 대한 성공적인 응답
- **오류(Error)**: 요청이 실패했음을 나타내는 응답
- **알림(Notification)**: 응답을 기대하지 않는 일방적인 메시지

통신 계층

MCP는 여러 통신 방식을 지원합니다:

- **표준 입출력(Stdio) 통신**: 로컬 프로세스 간 통신에 적합
- **HTTP와 SSE(Server-Sent Events) 통신**: 서버에서 클라이언트로의 메시지는 SSE, 클라이언트에서 서버로는 HTTP POST 사용

연결 생명주기

1. **초기화**: 클라이언트가 프로토콜 버전과 기능을 포함한 initialize 요청을 전송
2. **서버 응답**: 서버가 자신의 프로토콜 버전과 기능으로 응답
3. **확인**: 클라이언트가 initialized 알림을 전송하여 확인
4. **메시지 교환**: 일반적인 요청-응답 패턴으로 통신
5. **종료**: 연결 종료 (clean shutdown, 연결 해제, 오류 상태 등)

초기화 메시지 예시:

```
// 초기화 요청 예시
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-03-26",
```

```
    "capabilities": {
      "supportsProgressReporting": true
    }
  }
}

// 초기화 응답 예시
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "protocolVersion": "2025-03-26",
    "capabilities": {
      "supportsResourceCancellation": true
    }
  }
}
```

2.3. 활용 사례

MCP는 다양한 상황에서 AI 모델의 기능을 확장하는 데 활용될 수 있습니다. 주요 활용 사례는 다음과 같습니다:

파일 시스템 접근

Claude가 로컬 파일 시스템에 접근하여 파일을 읽고, 쓰고, 검색할 수 있습니다. 이를 통해:

- 대용량 문서 분석
- 코드 파일 검사 및 수정
- 데이터 파일 처리

데이터베이스 연결

MCP를 통해 Claude가 데이터베이스에 쿼리를 실행하고 결과를 분석할 수 있습니다:

- 데이터 분석 및 시각화 준비
- 데이터베이스 스키마 이해 및 최적화
- 복잡한 쿼리 생성 및 디버깅

API 통합

Claude가 외부 API와 상호작용하여 실시간 데이터를 가져오거나 작업을 수행할 수 있습니다:

- 날씨 데이터 조회
- 주식 시장 정보 분석
- 번역 서비스 활용
- 이메일 전송 또는 메시지 작성

개발 환경 통합

IDE와 통합하여 코드 작성, 디버깅, 테스트를 지원합니다:

- 코드 베이스 탐색 및 이해
- 테스트 케이스 자동 생성
- 코드 리팩토링 제안
- 버그 진단 및 수정

> **팁:** MCP의 가장 강력한 측면 중 하나는 여러 서버를 연결하여 복합적인 작업을 수행할 수 있다는 점입니다. 예를 들어, 데이터베이스 서버와 파일 시스템 서버를 함께 사용하여 데이터를 추출하고 파일로 저장하는 작업을 수행할 수 있습니다.

기초

3. 시작하기

MCP를 시작하기 전에 필요한 환경 설정과 기본 요구사항에 대해 알아보겠습니다. 이 섹션에서는 MCP를 사용하기 위한 준비 과정부터 첫 번째 MCP 연결까지의 과정을 다룹니다.

3.1. 환경 설정

MCP를 사용하기 위해서는 다음과 같은 기본 요구사항이 필요합니다:

시스템 요구사항

- **운영체제**: Windows, macOS, 또는 Linux
- **Python**: Python 3.9 이상 (Python 서버 개발 시)
- **Node.js**: Node.js 16.x 이상 (JavaScript/TypeScript 서버 개발 시)

- **Java/Kotlin**: JDK 11 이상 (Java/Kotlin 서버 개발 시)
- **.NET**: .NET 6.0 이상 (C# 서버 개발 시)

개발 도구

- **텍스트 에디터 또는 IDE**: VS Code, IntelliJ IDEA, PyCharm 등
- **터미널**: 명령줄 작업을 위한 터미널 프로그램
- **Git**: 소스 코드 관리 및 MCP 리포지토리 클론을 위한 Git

MCP 호스트

MCP 서버를 테스트하려면 MCP를 지원하는 호스트 애플리케이션이 필요합니다. 가장 일반적인 옵션은:

- **Claude 데스크톱**: MCP 서버와 연결하여 테스트할 수 있는 가장 쉬운 방법
- **커스텀 MCP 클라이언트**: 직접 만든 MCP 클라이언트 (이 전자책의 클라이언트 개발 섹션에서 다룸)

3.2. 설치 방법

MCP를 사용하기 위해 필요한 도구와 라이브러리를 설치하는 방법을 알아보겠습니다.

Claude 데스크톱 설치

Claude 데스크톱은 MCP 서버와 연결하여 테스트할 수 있는 가장 간편한 방법입니다:

1. Claude 데스크톱 다운로드 페이지에서 최신 버전을 다운로드합니다.
2. 다운로드한 파일을 실행하여 설치를 완료합니다.
3. 설치가 완료되면 Claude 데스크톱을 실행합니다.

MCP SDK 설치

MCP 서버나 클라이언트를 개발하려면 해당 언어의 MCP SDK를 설치해야 합니다:

Python SDK

```
pip install mcp-sdk
# 또는
uv pip install mcp-sdk
```

TypeScript/JavaScript SDK

```
npm install @modelcontextprotocol/typescript-sdk
# 또는
yarn add @modelcontextprotocol/typescript-sdk
```

Java SDK

```
// build.gradle에 다음 의존성 추가
dependencies {
    implementation 'io.modelcontextprotocol:mcp-server:1.0.0'
}
```

C# SDK

```
dotnet add package ModelContextProtocol.Server
```

개발 환경 구성

Python을 사용한 MCP 개발 환경 설정 예시:

```
# 프로젝트 디렉토리 생성
mkdir my-mcp-server
cd my-mcp-server

# 가상 환경 생성 (uv 사용)
uv venv

# 의존성 설치
uv pip install mcp-sdk

# pyproject.toml 파일 생성
cat > pyproject.toml << EOF
[build-system]
requires = ["setuptools>=42", "wheel"]
build-backend = "setuptools.build_meta"

[project]
name = "my-mcp-server"
version = "0.1.0"
description = "My MCP Server"
requires-python = ">=3.9"
dependencies = [
    "mcp-sdk",
```

```
]
EOF
```

3.3. 첫 단계

이제 MCP를 시작하기 위한 첫 번째 단계를 알아보겠습니다.

Claude 데스크톱에서 MCP 서버 구성

Claude 데스크톱에서 MCP 서버를 사용하려면 구성 파일을 설정해야 합니다:

1. 텍스트 에디터에서 Claude 데스크톱 구성 파일을 엽니다:
   - `macOS: ~/Library/Application Support/Claude/claude_desktop_config.json`
   - `Windows: %APPDATA%\Claude\claude_desktop_config.json`
2. 파일이 없다면 새로 생성하고 다음과 같이 작성합니다:

```
{
  "mcpServers": {
    "my-server": {
      "command": "python /path/to/your/server.py"
    }
  }
}
```

이 구성은 Claude 데스크톱에 "my-server"라는 이름의 MCP 서버가 있으며, 해당 서버를 실행하기 위해 `python /path/to/your/server.py` 명령을 사용한다고 알려줍니다.

미리 만들어진 MCP 서버 사용하기

처음부터 직접 MCP 서버를 개발하지 않고, 이미 만들어진 서버를 사용할 수도 있습니다:

1. MCP 서버 리포지토리에서 원하는 서버를 클론합니다.
2. README.md 파일의 지시에 따라 서버를 설정합니다.
3. Claude 데스크톱 구성 파일을 업데이트하여 서버를 추가합니다.

MCP 서버 테스트

설정한 MCP 서버가 제대로 작동하는지 확인하는 방법:

1. Claude 데스크톱을 (재)시작합니다.
2. 채팅 인터페이스 오른쪽 상단에 도구(망치) 아이콘이 표시되는지 확인합니다.
3. 도구 아이콘을 클릭하여 연결된 MCP 서버와 사용 가능한 도구 목록을 확인합니다.
4. Claude에게 서버의 기능을 사용하도록 요청합니다.

> ⚠ **주의:**
> MCP 서버가 Claude 데스크톱에 표시되지 않는 경우, 다음을 확인하세요:
>
> - 구성 파일이 올바른 위치에 있고 JSON 형식이 올바른지
> - 서버 경로가 정확한지
> - 필요한 의존성이 모두 설치되어 있는지
> - Claude 데스크톱을 재시작했는지

활용

4. MCP 서버 개발

이 섹션에서는 직접 MCP 서버를 개발하는 방법에 대해 배웁니다. 기본적인 서버 구조부터 시작하여 도구 구현, 서버 통신 처리, 그리고 고급 기능까지 단계별로 알아보겠습니다.

4.1. 서버 기초

MCP 서버는 클라이언트에게 특정 기능을 제공하는 프로그램입니다. 서버의 기본 구조와 생명주기에 대해 알아보겠습니다.

서버 구조

기본적인 MCP 서버는 다음과 같은 요소로 구성됩니다:

- **서버 인스턴스**: MCP 프로토콜을 처리하는 핵심 객체
- **도구 등록**: 클라이언트에게 제공하는 기능 정의
- **도구 실행기**: 요청된 도구를 실행하는 핸들러

- **메인 실행 로직**: 서버를 시작하고 관리하는 코드

기본 Python 서버 예제

다음은 간단한 날씨 정보를 제공하는 MCP 서버의 기본 구조입니다:

```
import asyncio
import json
from typing import Dict, Any, List

from mcp_sdk import Server
from mcp_sdk.types import ToolDefinition

# 서버 인스턴스 생성
server = Server()

# 도구 리스트 등록
@server.list_tools
async def list_tools() -> List[ToolDefinition]:
    return [
        ToolDefinition(
            name="get-weather",
            description="현재 날씨 정보를 가져옵니다",
            input_schema={
                "type": "object",
                "properties": {
                    "location": {
                        "type": "string",
                        "description": "날씨를 조회할 위치"
                    }
                },
                "required": ["location"]
            }
        )
    ]

# 도구 실행 핸들러
@server.execute_tool
async def execute_tool(tool_name: str, params: Dict[str, Any]) ->
Any:
    if tool_name == "get-weather":
        location = params.get("location", "")
        # 실제로는 여기서 API 호출 등이 필요
        weather_data = {
            "location": location,
            "temperature": 23,
            "condition": "맑음",
            "humidity": 60
```

```
        }
        return weather_data
    else:
        raise ValueError(f"알 수 없는 도구: {tool_name}")

# 메인 함수
async def main():
    await server.serve()

if __name__ == "__main__":
    asyncio.run(main())
```

서버 생명주기

MCP 서버의 생명주기는 다음과 같습니다:

1. **초기화**: 서버 객체 생성 및 도구 등록
2. **연결 설정**: 클라이언트의 연결 요청 수락
3. **핸드셰이크**: 프로토콜 버전 및 기능 교환
4. **도구 목록 제공**: 클라이언트 요청 시 등록된 도구 목록 제공
5. **도구 실행**: 클라이언트의 도구 실행 요청 처리
6. **연결 종료**: 클라이언트 연결 종료 또는 서버 종료 시 리소스 정리

4.2. 도구 구현

MCP 서버의 핵심 기능은 도구(Tool)입니다. 도구는 클라이언트가 사용할 수 있는 기능을 정의합니다.

도구 정의 구조

도구 정의는 다음 요소로 구성됩니다:

- **이름(name)**: 도구의 고유 식별자
- **설명(description)**: 도구의 기능 설명
- **입력 스키마(input_schema)**: JSON Schema 형식의 입력 파라미터 정의
- **출력 스키마(output_schema, 선택사항)**: 도구의 출력 구조 정의

효과적인 도구 설계

효과적인 MCP 도구를 설계하기 위한 가이드라인:

- **명확한 이름과 설명**: 도구의 이름과 설명은 LLM이 이해하기 쉽게 명확하게 작성
- **단일 책임 원칙**: 각 도구는 하나의 명확한 기능에 집중
- **견고한 입력 검증**: 상세한 스키마로 입력 유효성 검사
- **유용한 오류 메시지**: 문제 발생 시 명확한 오류 메시지 제공
- **적절한 응답 형식**: LLM이 처리하기 쉬운 구조화된 응답 제공

다양한 도구 유형 예제

파일 시스템 접근 도구

```
ToolDefinition(
    name="read-file",
    description="지정된 경로의 파일 내용을 읽습니다",
    input_schema={
        "type": "object",
        "properties": {
            "path": {
                "type": "string",
                "description": "읽을 파일의 경로"
            }
        },
        "required": ["path"]
    }
)

# 도구 실행 예시
if tool_name == "read-file":
    path = params.get("path", "")
    try:
        with open(path, "r", encoding="utf-8") as f:
            content = f.read()
        return {"content": content}
    except Exception as e:
        return {"error": str(e)}
```

API 호출 도구

```
ToolDefinition(
    name="search-web",
    description="웹에서 정보를 검색합니다",
    input_schema={
        "type": "object",
```

```
        "properties": {
            "query": {
                "type": "string",
                "description": "검색할 쿼리"
            },
            "limit": {
                "type": "integer",
                "description": "최대 결과 수",
                "default": 5
            }
        },
        "required": ["query"]
    }
)

# 도구 실행 예시
if tool_name == "search-web":
    import requests

    query = params.get("query", "")
    limit = params.get("limit", 5)

    try:
        response = requests.get(
            "https://api.example.com/search",
            params={"q": query, "limit": limit}
        )
        return response.json()
    except Exception as e:
        return {"error": str(e)}
```

데이터베이스 쿼리 도구

```
ToolDefinition(
    name="query-database",
    description="SQL 쿼리를 실행하고 결과를 반환합니다",
    input_schema={
        "type": "object",
        "properties": {
            "query": {
                "type": "string",
                "description": "실행할 SQL 쿼리"
            },
            "params": {
                "type": "array",
                "description": "쿼리 파라미터",
                "default": []
```

```
            }
        },
        "required": ["query"]
    }
)

# 도구 실행 예시
if tool_name == "query-database":
    import sqlite3

    query = params.get("query", "")
    query_params = params.get("params", [])

    try:
        conn = sqlite3.connect("database.db")
        cursor = conn.cursor()
        cursor.execute(query, query_params)

        if query.strip().upper().startswith("SELECT"):
            columns = [desc[0] for desc in cursor.description]
            rows = cursor.fetchall()
            result = {
                "columns": columns,
                "rows": rows
            }
        else:
            conn.commit()
            result = {"affected_rows": cursor.rowcount}

        conn.close()
        return result
    except Exception as e:
        return {"error": str(e)}
```

4.3. 서버 통신

MCP 서버는 클라이언트와 다양한 방식으로 통신합니다. 이 섹션에서는 MCP 통신의 기본 원리와 다양한 통신 방식에 대해 알아보겠습니다.

통신 방식

MCP는 두 가지 주요 통신 방식을 지원합니다:

- **표준 입출력(Stdio) 통신**

- 로컬 프로세스 간 통신에 적합
- Claude 데스크톱과 같은 로컬 애플리케이션에서 주로 사용
- JSON-RPC 메시지를 표준 입출력 스트림으로 전송

- **HTTP와 SSE 통신**
  - 클라이언트에서 서버로의 요청은 HTTP POST
  - 서버에서 클라이언트로의 응답은 Server-Sent Events(SSE)
  - 원격 서비스와의 통신에 적합

비동기 처리

MCP 서버는 비동기 방식으로 작동합니다. 이를 통해 여러 클라이언트 요청을 효율적으로 처리할 수 있습니다.

```
# Python의 asyncio를 활용한 비동기 도구 실행 예시
@server.execute_tool
async def execute_tool(tool_name: str, params: Dict[str, Any]) ->
Any:
    if tool_name == "long-running-task":
        # 비동기 작업 시뮬레이션
        for i in range(10):
            # 진행 상황 보고
            yield {"progress": i * 10, "message": f"진행 중... {i *
10}%"}
            await asyncio.sleep(1)  # 비동기 대기

        # 최종 결과 반환
        return {"status": "complete", "result": "작업이 완료되었습니다"}
```

오류 처리

MCP 서버는 다양한 오류 상황을 처리해야 합니다:

- **클라이언트 오류**: 잘못된 요청, 잘못된 파라미터 등
- **서버 오류**: 내부 처리 오류, 리소스 접근 실패 등
- **통신 오류**: 연결 끊김, 타임아웃 등

```
@server.execute_tool
async def execute_tool(tool_name: str, params: Dict[str, Any]) ->
Any:
    try:
        if tool_name == "process-data":
            # 파라미터 검증
            if "data" not in params:
                return {"error": "파라미터 'data'가, 필요합니다", "code":
"MISSING_PARAMETER"}
```

```
            # 데이터 처리 로직
            data = params["data"]
            result = process_data(data)
            return {"success": True, "result": result}
        else:
            return {"error": f"알 수 없는 도구: {tool_name}", "code":
"UNKNOWN_TOOL"}
    except FileNotFoundError as e:
        return {"error": f"파일을 찾을 수 없습니다: {str(e)}", "code":
"FILE_NOT_FOUND"}
    except PermissionError as e:
        return {"error": f"권한이 없습니다: {str(e)}", "code":
"PERMISSION_DENIED"}
    except Exception as e:
        # 예상치 못한 오류 처리
        print(f"오류 발생: {str(e)}", file=sys.stderr)
        return {"error": "내부 서버 오류", "code": "INTERNAL_ERROR"}
```

4.4. 고급 서버 기능

기본적인 MCP 서버 구현을 넘어, 더 강력한 기능들을 알아보겠습니다.

진행 상황 보고

오래 걸리는 작업의 경우, 클라이언트에게 진행 상황을 지속적으로 보고할 수 있습니다:

```
@server.execute_tool
async def execute_tool(tool_name: str, params: Dict[str, Any]) ->
Any:
    if tool_name == "process-large-file":
        path = params.get("path", "")

        try:
            # 파일 크기 확인
            file_size = os.path.getsize(path)
            processed = 0

            with open(path, "r", encoding="utf-8") as f:
                # 청크 단위로 처리
                while chunk := f.read(4096):
                    # 청크 처리 로직
                    process_chunk(chunk)
```

```
                    # 진행 상황 업데이트
                    processed += len(chunk)
                    progress_percent = min(100, int((processed /
file_size) * 100))

                    # 진행 상황 보고 (generator를 통한 중간 결과 반환)
                    yield {
                        "progress": progress_percent,
                        "processed_bytes": processed,
                        "total_bytes": file_size
                    }

                    # 실제 처리에 약간의 시간이 걸린다고 가정
                    await asyncio.sleep(0.1)

            # 최종 결과 반환
            return {"status": "complete", "processed_bytes":
processed}
        except Exception as e:
            return {"error": str(e)}
```

자원 관리

MCP 서버가 외부 자원을 효율적으로 관리하는 방법:

```
class DatabasePool:
    def __init__(self, connection_string, max_connections=10):
        self.connection_string = connection_string
        self.max_connections = max_connections
        self.available_connections = []
        self.used_connections = set()
        self._lock = asyncio.Lock()

    async def get_connection(self):
        async with self._lock:
            if not self.available_connections:
                if len(self.used_connections) >=
self.max_connections:
                    # 연결 풀이 가득 찬 경우 대기
                    await asyncio.sleep(0.1)
                    return await self.get_connection()

                # 새 연결 생성
                conn = await self._create_connection()
            else:
                # 사용 가능한 연결 재사용
                conn = self.available_connections.pop()
```

```
            self.used_connections.add(conn)
            return conn

    async def release_connection(self, conn):
        async with self._lock:
            self.used_connections.remove(conn)
            self.available_connections.append(conn)

    async def _create_connection(self):
        # 실제 DB 연결 생성 로직
        import aiosqlite
        return await aiosqlite.connect(self.connection_string)

    async def close_all(self):
        async with self._lock:
            for conn in self.available_connections +
list(self.used_connections):
                await conn.close()
            self.available_connections = []
            self.used_connections = set()

# 서버에서 사용
db_pool = DatabasePool("database.db")

@server.execute_tool
async def execute_tool(tool_name: str, params: Dict[str, Any]) ->
Any:
    if tool_name == "query-database":
        query = params.get("query", "")

        # 커넥션 획득
        conn = await db_pool.get_connection()
        try:
            cursor = await conn.cursor()
            await cursor.execute(query)
            rows = await cursor.fetchall()
            return {"rows": rows}
        finally:
            # 항상 커넥션 반환
            await db_pool.release_connection(conn)

# 서버 종료 시 리소스 정리
async def cleanup():
    await db_pool.close_all()

server.on_shutdown(cleanup)
```

보안 고려사항

MCP 서버 개발 시 고려해야 할 보안 사항:

- **입력 검증**: 모든 클라이언트 입력에 대한 철저한 검증
- **경로 조작 방지**: 파일 경로 등의 입력에서 디렉토리 순회 공격 방지
- **리소스 제한**: CPU, 메모리, 디스크 사용량 등에 대한 제한
- **권한 관리**: 최소 권한 원칙 적용
- **기밀 정보 보호**: 암호, API 키 등의 노출 방지

```
# 파일 경로 검증 예시
def is_safe_path(base_path, user_path):
    # 절대 경로로 변환
    abs_path = os.path.abspath(os.path.join(base_path, user_path))

    # base_path 내부에 있는지 확인
    return abs_path.startswith(os.path.abspath(base_path))

@server.execute_tool
async def execute_tool(tool_name: str, params: Dict[str, Any]) ->
Any:
    if tool_name == "read-file":
        path = params.get("path", "")

        # 안전한 기본 디렉토리 설정
        base_dir = "/safe/read/directory"

        # 경로 검증
        if not is_safe_path(base_dir, path):
            return {"error": "접근이 허용되지 않은 경로입니다", "code":
"INVALID_PATH"}

        # 실제 파일 읽기
        try:
            full_path = os.path.join(base_dir, path)
            with open(full_path, "r", encoding="utf-8") as f:
                content = f.read()
            return {"content": content}
        except Exception as e:
            return {"error": str(e)}
```

팁:
MCP 서버 개발 시 항상 보안을 최우선으로 고려하세요. 특히 파일 시스템, 데이터베이스, 네트워크 리소스와 같은 시스템 자원에 접근하는 도구를 구현할 때는 더욱 주의해야 합니다.

활용

5. MCP 클라이언트 개발

MCP 클라이언트는 LLM을 기반으로 한 애플리케이션이 MCP 서버에 연결하여 그 기능을 활용할 수 있게 해주는 프로그램입니다. 이 섹션에서는 MCP 클라이언트의 개발 방법에 대해 알아보겠습니다.

5.1. 클라이언트 기초

MCP 클라이언트의 기본 구조와 작동 원리에 대해 알아보겠습니다.

클라이언트 구조

기본적인 MCP 클라이언트는 다음과 같은 요소로 구성됩니다:

- **클라이언트 인스턴스**: MCP 프로토콜을 처리하는 핵심 객체
- **세션 관리**: 서버와의 연결 세션 관리
- **도구 목록 조회**: 서버에서 제공하는 도구 목록 조회
- **도구 실행 요청**: 서버에 도구 실행 요청 전송
- **결과 처리**: 서버의 응답 처리

기본 Python 클라이언트 예제

다음은 간단한 MCP 클라이언트의 기본 구조입니다:

```
import asyncio
import os
from typing import Dict, Any, List, Optional

from dotenv import load_dotenv
from anthropic import AsyncAnthropic
from mcp_sdk import Client
```

```
from mcp_sdk.types import ToolDefinition

# 환경 변수 로드
load_dotenv()

class MCPClient:
    def __init__(self):
        # Anthropic API 클라이언트 초기화
        self.anthropic =
AsyncAnthropic(api_key=os.getenv("ANTHROPIC_API_KEY"))
        # 대화 컨텍스트 관리
        self.messages = []
        # 출력을 저장할 버퍼
        self.response_buffer = []

    async def connect_to_server(self, server_path: str) -> bool:
        """MCP 서버에 연결합니다."""
        try:
            # MCP 클라이언트 생성 및 서버 연결
            self.client = Client()
            await self.client.connect(server_path)

            # 사용 가능한 도구 목록 조회
            self.tools = await self.client.list_tools()
            print(f"연결 성공: {len(self.tools)}개의 도구를 사용할 수 있습니
다.")

            # 도구 정보 출력
            for tool in self.tools:
                print(f"- {tool.name}: {tool.description}")

            return True
        except Exception as e:
            print(f"서버 연결 실패: {str(e)}")
            return False

    async def process_query(self, query: str) -> str:
        """사용자 쿼리를 처리하고 결과를 반환합니다."""
        # 메시지 추가
        self.messages.append({"role": "user", "content": query})

        # 응답 버퍼 초기화
        self.response_buffer = []

        # 도구 정의 준비
        tool_definitions = [
            {
                "name": tool.name,
                "description": tool.description,
                "input_schema": tool.input_schema
```

```
            }
            for tool in self.tools
        ]

        # Claude에 요청 전송
        response = await self.anthropic.messages.create(
            model="claude-3-5-sonnet-20240620",
            messages=self.messages,
            max_tokens=1024,
            tools=tool_definitions
        )

        # 응답 처리
        message = response.content[0].text

        # 도구 호출이 있는 경우
        if response.tool_use:
            for tool_use in response.tool_use:
                tool_name = tool_use.name
                tool_params = tool_use.input

                print(f"도구 호출: {tool_name}")

                # 도구 실행
                try:
                    result = await
self.client.execute_tool(tool_name, tool_params)
                    print(f"도구 실행 결과: {result}")

                    # 도구 실행 결과를 Claude에 전달
                    followup = await self.anthropic.messages.create(
                        model="claude-3-5-sonnet-20240620",
                        messages=self.messages + [
                            {"role": "assistant", "content":
message},
                            {
                                "role": "tool",
                                "name": tool_name,
                                "content": str(result)
                            }
                        ],
                        max_tokens=1024
                    )

                    # 최종 응답 업데이트
                    message = followup.content[0].text
                except Exception as e:
                    print(f"도구 실행 오류: {str(e)}")

        # 응답 메시지 추가
```

```
        self.messages.append({"role": "assistant", "content":
message})

        return message

    async def chat_loop(self):
        """대화형 인터페이스를 제공합니다."""
        print("MCP 클라이언트가 준비되었습니다. '종료'를 입력하면 종료됩니다.")

        while True:
            query = input("\n> ")

            if query.lower() in ["종료", "exit", "quit"]:
                break

            response = await self.process_query(query)
            print(f"\n{response}")

    async def cleanup(self):
        """리소스를 정리합니다."""
        if hasattr(self, 'client'):
            await self.client.disconnect()

async def main():
    client = MCPClient()

    # 서버 경로 (Python 스크립트 또는 디렉토리)
    server_path = "path/to/your/server.py"

    try:
        if await client.connect_to_server(server_path):
            await client.chat_loop()
    finally:
        await client.cleanup()

if __name__ == "__main__":
    asyncio.run(main())
```

클라이언트 생명주기

MCP 클라이언트의 생명주기는 다음과 같습니다:

1. **초기화**: 클라이언트 객체 생성 및 API 클라이언트 설정
2. **서버 연결**: MCP 서버에 연결
3. **핸드셰이크**: 프로토콜 버전 및 기능 교환
4. **도구 목록 조회**: 서버에서 제공하는 도구 목록 조회

5. **대화 처리**: 사용자 쿼리 처리 및 도구 실행
6. **연결 종료**: 서버 연결 종료 및 리소스 정리

5.2. Claude와 통합하기

MCP 클라이언트를 Claude와 통합하는 방법에 대해 알아보겠습니다.

Claude API 활용

Anthropic의 Claude API를 사용하여 MCP 클라이언트를 구현하는 방법:

```
from anthropic import AsyncAnthropic

# Anthropic API 클라이언트 초기화
anthropic = AsyncAnthropic(api_key="your-api-key")

# 도구 정의 준비
tool_definitions = [
    {
        "name": "get-weather",
        "description": "현재 날씨 정보를 가져옵니다",
        "input_schema": {
            "type": "object",
            "properties": {
                "location": {
                    "type": "string",
                    "description": "날씨를 조회할 위치"
                }
            },
            "required": ["location"]
        }
    }
]

# Claude에 요청 전송
async def get_claude_response(messages, tools):
    response = await anthropic.messages.create(
        model="claude-3-5-sonnet-20240620",
        messages=messages,
        max_tokens=1024,
        tools=tools
    )

    return response
```

```
# 도구 사용 처리
async def handle_tool_use(response, client):
    message = response.content[0].text

    # 도구 호출이 있는 경우
    if response.tool_use:
        for tool_use in response.tool_use:
            tool_name = tool_use.name
            tool_params = tool_use.input

            print(f"도구 호출: {tool_name}")

            # 도구 실행
            try:
                result = await client.execute_tool(tool_name,
tool_params)
                print(f"도구 실행 결과: {result}")

                # 도구 실행 결과를 Claude에 전달
                followup = await anthropic.messages.create(
                    model="claude-3-5-sonnet-20240620",
                    messages=messages + [
                        {"role": "assistant", "content": message},
                        {
                            "role": "tool",
                            "name": tool_name,
                            "content": str(result)
                        }
                    ],
                    max_tokens=1024
                )

                # 최종 응답 업데이트
                message = followup.content[0].text
            except Exception as e:
                print(f"도구 실행 오류: {str(e)}")

    return message
```

도구 호출 처리

Claude의 도구 호출을 처리하는 과정:

1. **도구 정의 제공**: Claude에게 사용 가능한 도구 목록 제공
2. **사용자 쿼리 처리**: Claude가 사용자 질문 분석
3. **도구 선택**: Claude가 적절한 도구 선택
4. **도구 호출**: Claude가 선택한 도구와 파라미터 반환

5. **도구 실행**: 클라이언트가 MCP 서버에 도구 실행 요청
6. **결과 처리**: 서버의 응답을 Claude에게 전달
7. **최종 응답**: Claude가 도구 실행 결과를 바탕으로 최종 응답 생성

다중 도구 호출 처리

Claude가 여러 도구를 순차적으로 호출하는 경우의 처리 방법:

```
async def process_query_with_multiple_tools(query, messages,
tools, client):
    # 메시지 추가
    messages.append({"role": "user", "content": query})

    # Claude에 초기 요청 전송
    response = await anthropic.messages.create(
        model="claude-3-5-sonnet-20240620",
        messages=messages,
        max_tokens=1024,
        tools=tools
    )

    message = response.content[0].text
    messages.append({"role": "assistant", "content": message})

    # 도구 호출 처리
    tool_calls_remain = True
    max_tool_calls = 5  # 안전장치: 최대 도구 호출 수 제한
    call_count = 0

    while tool_calls_remain and call_count < max_tool_calls:
        call_count += 1
        tool_calls_remain = False

        if response.tool_use:
            for tool_use in response.tool_use:
                tool_name = tool_use.name
                tool_params = tool_use.input

                print(f"도구 호출 {call_count}: {tool_name}")

                try:
                    # 도구 실행
                    result = await client.execute_tool(tool_name,
tool_params)

                    # 도구 실행 결과를 메시지에 추가
```

```
                    messages.append({
                        "role": "tool",
                        "name": tool_name,
                        "content": str(result)
                    })

                    # Claude에 후속 요청 전송
                    response = await anthropic.messages.create(
                        model="claude-3-5-sonnet-20240620",
                        messages=messages,
                        max_tokens=1024,
                        tools=tools
                    )

                    message = response.content[0].text
                    messages.append({"role": "assistant",
"content": message})

                    # 추가 도구 호출이 있는지 확인
                    if response.tool_use:
                        tool_calls_remain = True

                    break  # 한 번에 하나의 도구 호출만 처리
                except Exception as e:
                    print(f"도구 실행 오류: {str(e)}")
                    messages.append({
                        "role": "tool",
                        "name": tool_name,
                        "content": f"오류: {str(e)}"
                    })

    # 최종 응답
    return messages[-1]["content"]
```

5.3. 클라이언트 통신

MCP 클라이언트의 통신 방식과 효율적인 서버 통신 관리에 대해 알아보겠습니다.

통신 모드

MCP 클라이언트는 서버와 두 가지 주요 통신 방식을 사용합니다:

- **로컬 프로세스 통신 (Stdio)**
  - 로컬 서버 실행 및 표준 입출력을 통한 통신

- 개발 및 테스트에 적합
- Claude 데스크톱에서 주로 사용하는 방식

- **HTTP/SSE 통신**
  - 클라이언트에서 서버로는 HTTP POST 요청
  - 서버에서 클라이언트로는 Server-Sent Events (SSE)
  - 원격 서버와의 통신에 적합

연결 관리

효율적인 서버 연결 관리를 위한 전략:

```
class ConnectionManager:
    def __init__(self):
        self.clients = {}
        self._lock = asyncio.Lock()

    async def get_client(self, server_path: str):
        """서버 경로에 대한 클라이언트를 반환합니다."""
        async with self._lock:
            if server_path not in self.clients:
                # 새 클라이언트 생성 및 연결
                client = Client()
                try:
                    await client.connect(server_path)
                    self.clients[server_path] = client
                except Exception as e:
                    print(f"서버 연결 실패: {str(e)}")
                    raise

            return self.clients[server_path]

    async def disconnect_all(self):
        """모든 클라이언트 연결을 종료합니다."""
        async with self._lock:
            for server_path, client in self.clients.items():
                try:
                    await client.disconnect()
                except Exception as e:
                    print(f"연결 종료 오류 ({server_path}):
{str(e)}")

            self.clients = {}

# 사용 예시
conn_manager = ConnectionManager()
```

```
async def process_with_server(server_path, tool_name, params):
    client = await conn_manager.get_client(server_path)
    return await client.execute_tool(tool_name, params)

# 종료 시 정리
async def cleanup():
    await conn_manager.disconnect_all()
```

오류 처리

클라이언트 통신 과정에서 발생할 수 있는 다양한 오류 상황 처리:

```
async def execute_tool_with_retry(client, tool_name, params,
max_retries=3, retry_delay=1.0):
    """재시도 로직이 포함된 도구 실행 함수"""
    retries = 0
    last_error = None

    while retries < max_retries:
        try:
            return await client.execute_tool(tool_name, params)
        except ConnectionError as e:
            last_error = e
            print(f"연결 오류, 재시도 중...
({retries+1}/{max_retries})")

            # 재연결 시도
            try:
                await client.reconnect()
            except Exception as reconnect_error:
                print(f"재연결 실패: {str(reconnect_error)}")

            retries += 1
            await asyncio.sleep(retry_delay * retries)  # 지수 백오
프
        except TimeoutError as e:
            last_error = e
            print(f"타임아웃, 재시도 중...
({retries+1}/{max_retries})")
            retries += 1
            await asyncio.sleep(retry_delay)
        except Exception as e:
            # 그 외 예외는 즉시 전파
            print(f"도구 실행 오류: {str(e)}")
            raise

    # 최대 재시도 횟수 초과
```

```
    raise last_error or RuntimeError(f"최대 재시도 횟수
({max_retries})를 초과했습니다.")
```

5.4. 고급 클라이언트 기능

MCP 클라이언트의 고급 기능과 최적화 방법에 대해 알아보겠습니다.

비동기 작업 처리

오래 걸리는 작업을 비동기적으로 처리하는 방법:

```
class AsyncTaskManager:
    def __init__(self):
        self.tasks = {}
        self._task_id_counter = 0
        self._lock = asyncio.Lock()

    async def start_task(self, coroutine):
        """비동기 작업을 시작하고 작업 ID를 반환합니다."""
        async with self._lock:
            task_id = str(self._task_id_counter)
            self._task_id_counter += 1

            # 작업 시작
            task = asyncio.create_task(coroutine)
            self.tasks[task_id] = {
                "task": task,
                "status": "running",
                "result": None,
                "error": None,
                "start_time": asyncio.get_event_loop().time()
            }

            # 완료 콜백 설정
            task.add_done_callback(lambda t:
self._task_done_callback(task_id, t))

            return task_id

    def _task_done_callback(self, task_id, task):
        """작업 완료 시 호출되는 콜백"""
        try:
            result = task.result()
            self.tasks[task_id]["status"] = "completed"
```

```
            self.tasks[task_id]["result"] = result
        except Exception as e:
            self.tasks[task_id]["status"] = "failed"
            self.tasks[task_id]["error"] = str(e)

    async def get_task_status(self, task_id):
        """작업 상태를 조회합니다."""
        if task_id not in self.tasks:
            return {"error": "작업을 찾을 수 없습니다"}

        task_info = self.tasks[task_id].copy()
        # 작업 객체 자체는 제외
        del task_info["task"]

        # 실행 시간 계산
        current_time = asyncio.get_event_loop().time()
        task_info["elapsed_time"] = current_time -
task_info["start_time"]

        return task_info

    async def cancel_task(self, task_id):
        """작업을 취소합니다."""
        if task_id not in self.tasks:
            return {"error": "작업을 찾을 수 없습니다"}

        if self.tasks[task_id]["status"] == "running":
            self.tasks[task_id]["task"].cancel()
            self.tasks[task_id]["status"] = "cancelled"

        return {"status": self.tasks[task_id]["status"]}

# 사용 예시
task_manager = AsyncTaskManager()

@server.execute_tool
async def execute_tool(tool_name: str, params: Dict[str, Any]) ->
Any:
    if tool_name == "start-long-task":
        # 오래 걸리는 작업 시작
        async def long_running_task():
            # 실제 작업 로직
            await asyncio.sleep(30)  # 예: 30초 소요
            return {"result": "작업 완료"}

        task_id = await
task_manager.start_task(long_running_task())
        return {"task_id": task_id, "status": "started"}

    elif tool_name == "get-task-status":
```

```
        # 작업 상태 조회
        task_id = params.get("task_id")
        return await task_manager.get_task_status(task_id)

    elif tool_name == "cancel-task":
        # 작업 취소
        task_id = params.get("task_id")
        return await task_manager.cancel_task(task_id)
```

캐싱과 성능 최적화

반복적인 작업의 성능을 최적화하기 위한 캐싱 전략:

```
class ResultCache:
    def __init__(self, max_size=100, ttl=3600):
        self.cache = {}
        self.max_size = max_size
        self.ttl = ttl  # 초 단위 유효 시간
        self._access_times = {}
        self._lock = asyncio.Lock()

    async def get(self, key):
        """캐시에서 값을 조회합니다."""
        async with self._lock:
            if key not in self.cache:
                return None

            # 만료 확인
            current_time = time.time()
            if current_time - self._access_times[key]["created"] >
self.ttl:
                # 만료된 항목 삭제
                del self.cache[key]
                del self._access_times[key]
                return None

            # 접근 시간 업데이트
            self._access_times[key]["accessed"] = current_time
            return self.cache[key]

    async def set(self, key, value):
        """캐시에 값을 저장합니다."""
        async with self._lock:
            # 캐시 크기 제한 확인
            if len(self.cache) >= self.max_size and key not in
self.cache:
                # LRU 정책: 가장 오래 접근하지 않은 항목 제거
```

```
                oldest_key = min(
                    self._access_times,
                    key=lambda k: self._access_times[k]
["accessed"]
                )
                del self.cache[oldest_key]
                del self._access_times[oldest_key]

            # 새 값 저장
            current_time = time.time()
            self.cache[key] = value
            self._access_times[key] = {
                "created": current_time,
                "accessed": current_time
            }

            return value

    async def invalidate(self, key):
        """특정 키의 캐시를 무효화합니다."""
        async with self._lock:
            if key in self.cache:
                del self.cache[key]
                del self._access_times[key]

    async def clear(self):
        """모든 캐시를 비웁니다."""
        async with self._lock:
            self.cache = {}
            self._access_times = {}

# 사용 예시
result_cache = ResultCache()

async def execute_cached_tool(client, tool_name, params):
    # 캐시 키 생성 (도구 이름 + 파라미터의 해시)
    cache_key = f"{tool_name}:{hash(frozenset(params.items()))}"

    # 캐시 확인
    cached_result = await result_cache.get(cache_key)
    if cached_result is not None:
        print(f"캐시에서 결과 사용: {tool_name}")
        return cached_result

    # 캐시에 없으면 실제 실행
    print(f"도구 실행: {tool_name}")
    result = await client.execute_tool(tool_name, params)

    # 결과 캐싱
    await result_cache.set(cache_key, result)
```

```
    return result
```

멀티 서버 통합

여러 MCP 서버를 통합하여 사용하는 방법:

```
class MultiServerClient:
    def __init__(self):
        self.servers = {}
        self.tool_registry = {}  # 도구 이름 -> 서버 매핑

    async def add_server(self, server_name, server_path):
        """새 서버를 추가하고 도구 목록을 로드합니다."""
        client = Client()
        try:
            await client.connect(server_path)
            tools = await client.list_tools()

            # 서버 등록
            self.servers[server_name] = {
                "client": client,
                "tools": tools
            }

            # 도구 등록
            for tool in tools:
                self.tool_registry[tool.name] = {
                    "server": server_name,
                    "definition": tool
                }

            return len(tools)
        except Exception as e:
            print(f"서버 추가 실패 ({server_name}): {str(e)}")
            raise

    def get_all_tools(self):
        """모든 서버의 도구 목록을 반환합니다."""
        return [tool_info["definition"] for tool_info in
self.tool_registry.values()]

    async def execute_tool(self, tool_name, params):
        """도구를 실행합니다."""
        if tool_name not in self.tool_registry:
            raise ValueError(f"알 수 없는 도구: {tool_name}")
```

```
        server_name = self.tool_registry[tool_name]["server"]
        client = self.servers[server_name]["client"]

        return await client.execute_tool(tool_name, params)

    async def disconnect_all(self):
        """모든 서버 연결을 종료합니다."""
        for server_name, server_info in self.servers.items():
            try:
                await server_info["client"].disconnect()
            except Exception as e:
                print(f"연결 종료 오류 ({server_name}): {str(e)}")

# 사용 예시
multi_client = MultiServerClient()

# 여러 서버 추가
await multi_client.add_server("file-server",
"path/to/file_server.py")
await multi_client.add_server("weather-server",
"path/to/weather_server.py")
await multi_client.add_server("database-server",
"path/to/db_server.py")

# 모든 도구 목록 조회
all_tools = multi_client.get_all_tools()

# 도구 실행
result = await multi_client.execute_tool("get-weather",
{"location": "Seoul"})

# 정리
await multi_client.disconnect_all()
```

⚠ **주의:**
여러 서버를 통합할 때는 도구 이름 충돌에 주의해야 합니다. 서로 다른 서버에서 동일한 이름의 도구를 제공하는 경우, 나중에 추가된 서버의 도구가 이전 도구를 덮어쓸 수 있습니다.

중급

6. 실행 및 디버깅

MCP 애플리케이션을 효과적으로 테스트하고 디버깅하는 방법에 대해 알아보겠습니다. 이 섹션에서는 개발 과정에서 발생할 수 있는 문제를 해결하는 전략과 도구를 다룹니다.

6.1. 테스트

MCP 서버와 클라이언트를 효과적으로 테스트하는 방법을 알아보겠습니다.

단위 테스트

개별 도구 및 함수의 동작을 검증하기 위한 단위 테스트:

```
# test_weather_server.py
import unittest
import asyncio
import json
from unittest.mock import patch, MagicMock

# 테스트할 서버 모듈 가져오기
from weather_server import list_tools, execute_tool

class WeatherServerTest(unittest.TestCase):
    def setUp(self):
        # 테스트 설정
        self.loop = asyncio.new_event_loop()
        asyncio.set_event_loop(self.loop)

    def tearDown(self):
        # 테스트 정리
        self.loop.close()

    def test_list_tools(self):
        # list_tools 함수 테스트
        tools = self.loop.run_until_complete(list_tools())

        # 기대 결과 확인
        self.assertEqual(len(tools), 2, "도구 수가 2개여야 합니다")
        self.assertEqual(tools[0].name, "get-weather", "첫 번째 도
구 이름이 잘못되었습니다")
        self.assertEqual(tools[1].name, "get-alerts", "두 번째 도구
이름이 잘못되었습니다")

    @patch('weather_server.get_weather_data')
    def test_execute_tool_weather(self, mock_get_weather):
```

```
        # get_weather_data 모의 객체 설정
        mock_get_weather.return_value = {
            "temperature": 25,
            "condition": "맑음",
            "humidity": 60
        }

        # execute_tool 함수 테스트
        params = {"location": "Seoul"}
        result = self.loop.run_until_complete(execute_tool("get-
weather", params))

        # 모의 객체가 올바른 파라미터로 호출되었는지 확인
        mock_get_weather.assert_called_once_with("Seoul")

        # 결과 확인
        self.assertEqual(result["temperature"], 25)
        self.assertEqual(result["condition"], "맑음")
        self.assertEqual(result["humidity"], 60)

    def test_execute_tool_unknown(self):
        # 존재하지 않는 도구 호출 테스트
        with self.assertRaises(ValueError):
            self.loop.run_until_complete(execute_tool("unknown-
tool", {}))

if __name__ == '__main__':
    unittest.main()
```

통합 테스트

전체 MCP 서버와 클라이언트 간의 통합 테스트:

```
# test_integration.py
import unittest
import asyncio
import subprocess
import time
import os
from mcp_sdk import Client

class IntegrationTest(unittest.TestCase):
    def setUp(self):
        # 테스트 설정
        self.loop = asyncio.new_event_loop()
        asyncio.set_event_loop(self.loop)
```

```
        # 서버 프로세스 시작
        self.server_process = subprocess.Popen(
            ["python", "weather_server.py"],
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE
        )

        # 서버가 시작될 때까지 약간의 지연
        time.sleep(1)

        # 클라이언트 초기화
        self.client = Client()

self.loop.run_until_complete(self.client.connect("weather_server.
py"))

    def tearDown(self):
        # 테스트 정리
        self.loop.run_until_complete(self.client.disconnect())

        # 서버 프로세스 종료
        self.server_process.terminate()
        self.server_process.wait()

        self.loop.close()

    def test_tool_listing(self):
        # 도구 목록 조회 테스트
        tools =
self.loop.run_until_complete(self.client.list_tools())

        # 기대 결과 확인
        self.assertTrue(len(tools) >= 1, "최소 하나의 도구가 있어야 합니
다")
        self.assertTrue(any(tool.name == "get-weather" for tool
in tools),
                        "get-weather 도구가 목록에 있어야 합니다")

    def test_tool_execution(self):
        # 도구 실행 테스트
        result = self.loop.run_until_complete(
            self.client.execute_tool("get-weather", {"location":
"Seoul"})
        )

        # 결과 형식 확인
        self.assertIn("temperature", result, "응답에 temperature 필
드가 필요합니다")
        self.assertIn("condition", result, "응답에 condition 필드가
```

```
필요합니다")
        self.assertIn("humidity", result, "응답에 humidity 필드가 필
요합니다")

        # 값 유효성 확인
        self.assertIsInstance(result["temperature"], (int,
float))
        self.assertIsInstance(result["condition"], str)
        self.assertIsInstance(result["humidity"], (int, float))

if __name__ == '__main__':
    unittest.main()
```

자동화된 테스트 도구

MCP 테스트를 자동화하기 위한 도구와 기법:

- **MCP Inspector**: MCP 서버와 상호작용하고 도구 작동을 테스트하는 도구
- **테스트 자동화 스크립트**: 다양한 시나리오를 자동으로 테스트하는 스크립트
- **CI/CD 파이프라인 통합**: GitHub Actions, Jenkins 등과 연동하여 지속적 테스트

```
# mcp_test_runner.py
import asyncio
import os
import sys
import json
import argparse
from mcp_sdk import Client

async def run_test_scenarios(server_path, scenarios_file):
    """테스트 시나리오 파일에서 시나리오를 읽고 실행합니다."""
    # 시나리오 파일 로드
    with open(scenarios_file, 'r', encoding='utf-8') as f:
        scenarios = json.load(f)

    # 클라이언트 초기화
    client = Client()
    try:
        await client.connect(server_path)

        # 사용 가능한 도구 조회
        tools = await client.list_tools()
        print(f"사용 가능한 도구 {len(tools)}개:")
        for tool in tools:
            print(f"- {tool.name}: {tool.description}")
```

```
        # 시나리오 실행
        for i, scenario in enumerate(scenarios):
            print(f"\n[시나리오 {i+1}] {scenario.get('description',
'설명 없음')}")

            tool_name = scenario["tool"]
            params = scenario["params"]
            expected = scenario.get("expected", {})

            try:
                print(f"도구 실행: {tool_name}")
                print(f"파라미터: {params}")

                # 도구 실행
                result = await client.execute_tool(tool_name,
params)
                print(f"결과: {result}")

                # 결과 검증
                if "keys" in expected:
                    for key in expected["keys"]:
                        assert key in result, f"결과에 필요한 키가 없
습니다: {key}"

                if "values" in expected:
                    for key, value in expected["values"].items():
                        assert result.get(key) == value, f"결과 값
이 일치하지 않습니다: {key}"

                print("✅ 성공")
            except AssertionError as e:
                print(f"❌ 실패: {str(e)}")
            except Exception as e:
                print(f"❌ 오류: {str(e)}")
    finally:
        await client.disconnect()

def main():
    parser = argparse.ArgumentParser(description='MCP 서버 테스트 도
구')
    parser.add_argument('server', help='MCP 서버 경로')
    parser.add_argument('scenarios', help='시나리오 JSON 파일 경로')
    args = parser.parse_args()

    asyncio.run(run_test_scenarios(args.server, args.scenarios))

if __name__ == '__main__':
    main()
```

6.2. 디버깅 기법

MCP 애플리케이션에서 발생하는 문제를 효과적으로 디버깅하는 방법을 알아보겠습니다.

로깅

효과적인 로깅을 통한 문제 진단:

```
import logging
import sys
import os
import time

# 로거 설정
def setup_logger(name):
    # 로그 디렉토리 생성
    log_dir = "logs"
    os.makedirs(log_dir, exist_ok=True)

    # 로거 생성
    logger = logging.getLogger(name)
    logger.setLevel(logging.DEBUG)

    # 날짜별 로그 파일 설정
    timestamp = time.strftime("%Y%m%d")
    file_handler = logging.FileHandler(f"
{log_dir}/{name}_{timestamp}.log")
    file_handler.setLevel(logging.DEBUG)

    # 콘솔 출력 설정
    console_handler = logging.StreamHandler(sys.stdout)
    console_handler.setLevel(logging.INFO)

    # 포맷 설정
    formatter = logging.Formatter('%(asctime)s - %(name)s - %
(levelname)s - %(message)s')
    file_handler.setFormatter(formatter)
    console_handler.setFormatter(formatter)

    # 핸들러 추가
    logger.addHandler(file_handler)
    logger.addHandler(console_handler)

    return logger
```

```
# MCP 서버에서 사용
logger = setup_logger("mcp_server")

@server.execute_tool
async def execute_tool(tool_name: str, params: Dict[str, Any]) ->
Any:
    logger.info(f"도구 실행 요청: {tool_name}")
    logger.debug(f"도구 파라미터: {params}")

    try:
        if tool_name == "get-weather":
            location = params.get("location", "")
            logger.info(f"날씨 정보 조회: {location}")

            try:
                # API 호출
                result = get_weather_data(location)
                logger.debug(f"날씨 API 응답: {result}")
                return result
            except Exception as e:
                logger.error(f"날씨 API 호출 오류: {str(e)}",
exc_info=True)
                raise
        else:
            logger.warning(f"알 수 없는 도구: {tool_name}")
            raise ValueError(f"알 수 없는 도구: {tool_name}")
    except Exception as e:
        logger.error(f"도구 실행 오류: {str(e)}", exc_info=True)
        raise
```

프로토콜 검사

MCP 프로토콜 메시지 검사를 통한 문제 해결:

```
class ProtocolInspector:
    def __init__(self, enabled=True):
        self.enabled = enabled
        self.log_file = None
        self.request_history = []
        self.response_history = []
        self.max_history = 100

    def start_logging(self, file_path=None):
        """메시지 로깅 시작"""
        if not self.enabled:
```

```
            return

        if file_path:
            self.log_file = open(file_path, 'w', encoding='utf-
8')
            self.log(f"--- MCP 프로토콜 로깅 시작:
{time.strftime('%Y-%m-%d %H:%M:%S')} ---")

    def stop_logging(self):
        """메시지 로깅 종료"""
        if self.log_file:
            self.log(f"--- MCP 프로토콜 로깅 종료:
{time.strftime('%Y-%m-%d %H:%M:%S')} ---")
            self.log_file.close()
            self.log_file = None

    def log_request(self, request):
        """요청 메시지 로깅"""
        if not self.enabled:
            return

        self.request_history.append(request)
        if len(self.request_history) > self.max_history:
            self.request_history.pop(0)

        self.log(f">>> 요청: {json.dumps(request, indent=2,
ensure_ascii=False)}")

    def log_response(self, response):
        """응답 메시지 로깅"""
        if not self.enabled:
            return

        self.response_history.append(response)
        if len(self.response_history) > self.max_history:
            self.response_history.pop(0)

        self.log(f"<<< 응답: {json.dumps(response, indent=2,
ensure_ascii=False)}")

    def log(self, message):
        """메시지 로깅"""
        if self.log_file:
            self.log_file.write(f"{message}\n")
            self.log_file.flush()

    def get_last_exchange(self, count=1):
        """최근 교환 메시지 조회"""
        exchanges = []
        for i in range(min(count, len(self.request_history),
```

```
len(self.response_history))):
            idx = -1 - i
            exchanges.append({
                "request": self.request_history[idx],
                "response": self.response_history[idx]
            })
        return exchanges

# 클라이언트에서 사용
inspector = ProtocolInspector()
inspector.start_logging("mcp_debug.log")

# 원래 클라이언트 send/receive 메서드 대체
original_send = client.send
original_receive = client.receive

async def send_with_logging(message):
    inspector.log_request(message)
    return await original_send(message)

async def receive_with_logging():
    response = await original_receive()
    inspector.log_response(response)
    return response

client.send = send_with_logging
client.receive = receive_with_logging

# 사용 후 정리
inspector.stop_logging()
```

일반적인 문제 해결

자주 발생하는 MCP 관련 문제와 해결 방법:

문제	가능한 원인	해결 방법
서버 연결 실패	잘못된 경로, 서버 미실행, 권한 문제	• 서버 경로 확인 • 서버 프로세스 상태 확인 • 권한 설정 확인
도구 목록 불러오기 실패	잘못된 서버 구현, 초기화 오류	• list_tools 핸들러 구현 확인 • 서버 로그 확인

문제	가능한 원인	해결 방법
		• 프로토콜 메시지 검사
도구 실행 오류	잘못된 파라미터, 서버 내부 오류	• 파라미터 유효성 확인 • 서버 로그 확인 • 수동으로 도구 테스트
타임아웃	긴 작업 시간, 네트워크 문제	• 타임아웃 값 조정 • 진행 상황 보고 구현 • 작업 분할
메모리 사용량 과다	메모리 누수, 큰 데이터 처리	• 스트리밍 처리 구현 • 자원 정리 확인 • 메모리 프로파일링

6.3. 성능 최적화

MCP 서버와 클라이언트의 효율적인 운영을 위한 성능 최적화 방법에 대해 알아보겠습니다. 최적화를 통해 응답 시간을 단축하고, 리소스 사용량을 줄이며, 더 많은 사용자와 요청을 처리할 수 있습니다.

프로파일링 및 병목 식별

성능 최적화의 첫 번째 단계는 현재 성능을 측정하고 병목 지점을 식별하는 것입니다:

```
# Python에서 MCP 서버 성능 프로파일링
import cProfile
import pstats
import io
from mcp_sdk import Server

server = Server()
# ... 서버 설정 ...

def profile_server():
    # cProfile을 사용한 프로파일링
    pr = cProfile.Profile()
    pr.enable()

    # 테스트 실행 (예: 100번의 도구 호출)
    for i in range(100):
```

```
        test_execute_tool("get-weather", {"location": "Seoul"})

    pr.disable()

    # 결과 분석
    s = io.StringIO()
    ps = pstats.Stats(pr, stream=s).sort_stats('cumulative')
    ps.print_stats(20)  # 상위 20개 항목만 출력
    print(s.getvalue())

async def test_execute_tool(tool_name, params):
    # execute_tool 함수 직접 호출
    return await execute_tool(tool_name, params)

if __name__ == "__main__":
    profile_server()
```

응답 시간 최적화

MCP 서버의 응답 시간을 최적화하는 주요 방법:

비동기 처리 최적화

```
# 효율적인 비동기 처리 예시
import asyncio
from functools import lru_cache

@server.execute_tool
async def execute_tool(tool_name: str, params: Dict[str, Any]) ->
Any:
    if tool_name == "get-data":
        # 여러 비동기 작업을 병렬로 처리
        tasks = [
            fetch_main_data(params),
            fetch_additional_info(params),
            update_statistics(params)
        ]

        # 모든 작업을 동시에 실행
        results = await asyncio.gather(*tasks)

        # 결과 조합
        return {
            "main_data": results[0],
            "additional_info": results[1],
            "stats_updated": results[2]
        }
```

```
# 자주 요청되는 정보는 캐싱
@lru_cache(maxsize=100)
async def fetch_main_data(params_str):
    # params_str은 딕셔너리를 문자열로 변환한 값
    # (lru_cache가 딕셔너리를 직접 처리할 수 없으므로)
    params = json.loads(params_str)
    # ... 데이터 조회 로직 ...
    return data
```

데이터 전송량 최적화

```
# 응답 데이터 최적화
async def optimize_response(data, fields=None, max_items=None):
    """응답 데이터를 최적화합니다."""
    # 1. 필요한 필드만 선택
    if fields:
        if isinstance(data, list):
            result = [{k: item[k] for k in fields if k in item}
for item in data]
        else:
            result = {k: data[k] for k in fields if k in data}
    else:
        result = data

    # 2. 항목 수 제한
    if max_items and isinstance(result, list) and len(result) >
max_items:
        result = result[:max_items]

    return result

@server.execute_tool
async def execute_tool(tool_name: str, params: Dict[str, Any]) ->
Any:
    if tool_name == "search-items":
        # 쿼리 실행
        all_items = await search_database(params["query"])

        # 응답 최적화
        fields = params.get("fields", ["id", "name",
"description"])
        max_items = params.get("limit", 50)

        return await optimize_response(all_items, fields,
max_items)
```

메모리 사용량 최적화

MCP 서버의 메모리 사용량을 최적화하는 방법:

스트리밍 처리

```
# 대용량 데이터의 스트리밍 처리
@server.execute_tool
async def execute_tool(tool_name: str, params: Dict[str, Any]) ->
Any:
    if tool_name == "process-large-data":
        file_path = params.get("file_path")

        # 전체 파일을 메모리에 로드하지 않고 스트리밍 처리
        processed_count = 0
        result_chunks = []

        # 파일을 청크 단위로 처리
        async with aiofiles.open(file_path, 'r') as f:
            chunk_size = 1024 * 1024  # 1MB 청크
            while chunk := await f.read(chunk_size):
                # 청크 처리
                processed_data = process_chunk(chunk)
                result_chunks.append(processed_data)
                processed_count += len(chunk)

                # 진행 상황 보고
                yield {
                    "progress": {
                        "processed_bytes": processed_count,
                        "status": "processing"
                    }
                }

        # 최종 결과 반환
        return {
            "result": "".join(result_chunks),
            "processed_bytes": processed_count
        }
```

자원 할당 제한

```
# 메모리 사용량 제한 구현
class MemoryLimiter:
    def __init__(self, max_memory_mb=200):
        self.max_memory_mb = max_memory_mb
        self.current_usage = 0
        self._lock = asyncio.Lock()
```

```
    async def allocate(self, size_mb):
        """리소스 할당을 시도합니다."""
        async with self._lock:
            if self.current_usage + size_mb > self.max_memory_mb:
                raise MemoryError(f"메모리 할당 한도 초과: {size_mb}MB
요청, {self.max_memory_mb - self.current_usage}MB 가용")

            self.current_usage += size_mb
            return True

    async def release(self, size_mb):
        """리소스를 해제합니다."""
        async with self._lock:
            self.current_usage = max(0, self.current_usage -
size_mb)

# 사용 예시
memory_limiter = MemoryLimiter(max_memory_mb=500)

@server.execute_tool
async def execute_tool(tool_name: str, params: Dict[str, Any]) ->
Any:
    if tool_name == "process-image":
        image_path = params.get("image_path")

        # 이미지 크기 확인
        file_size_mb = os.path.getsize(image_path) / (1024 * 1024)
        # 처리에 필요한 예상 메모리 (원본 크기의 3배로 가정)
        required_memory = file_size_mb * 3

        try:
            # 메모리 할당 시도
            await memory_limiter.allocate(required_memory)

            try:
                # 이미지 처리 로직
                result = process_image(image_path)
                return result
            finally:
                # 처리 완료 후 메모리 해제
                await memory_limiter.release(required_memory)
        except MemoryError as e:
            return {"error": str(e), "suggestion": "더 작은 이미지로
시도하세요"}
```

확장성 및 동시성 처리

MCP 서버의 확장성과 동시성 처리를 위한 전략:

요청 제한 및 큐잉

```
# 요청 제한 및 큐잉 구현
class RequestLimiter:
    def __init__(self, max_concurrent=5, queue_size=20):
        self.semaphore = asyncio.Semaphore(max_concurrent)
        self.queue = asyncio.Queue(maxsize=queue_size)
        self._worker_task = None

    async def start_workers(self):
        """작업자 태스크를 시작합니다."""
        self._worker_task =
asyncio.create_task(self._process_queue())

    async def stop_workers(self):
        """작업자 태스크를 중지합니다."""
        if self._worker_task:
            self._worker_task.cancel()
            try:
                await self._worker_task
            except asyncio.CancelledError:
                pass

    async def _process_queue(self):
        """큐에서 작업을 처리합니다."""
        while True:
            job, future = await self.queue.get()
            try:
                async with self.semaphore:
                    result = await job()
                future.set_result(result)
            except Exception as e:
                future.set_exception(e)
            finally:
                self.queue.task_done()

    async def submit(self, job_coroutine):
        """작업을 제출하고 Future를 반환합니다."""
        future = asyncio.Future()
        try:
            await self.queue.put((job_coroutine, future))
            return await future
        except asyncio.QueueFull:
            raise RuntimeError("요청 큐가 가득 찼습니다. 나중에 다시 시도하
세요.")

# 사용 예시
request_limiter = RequestLimiter(max_concurrent=10, queue_size=50)
```

```
@server.execute_tool
async def execute_tool(tool_name: str, params: Dict[str, Any]) ->
Any:
    if tool_name == "process-data":
        # 작업 정의
        async def job():
            return await process_data_internal(params)

        # 작업 제출 및 결과 대기
        try:
            return await request_limiter.submit(job)
        except RuntimeError as e:
            return {"error": str(e), "retry_after": 5}
```

분산 처리

```
# 작업 분산 처리 구현
class WorkDistributor:
    def __init__(self, worker_count=4):
        self.worker_count = worker_count
        self.workers = []
        self.task_counter = 0

    async def start_workers(self):
        """작업자 프로세스를 시작합니다."""
        import multiprocessing

        for i in range(self.worker_count):
            # 작업자 프로세스 시작
            worker = multiprocessing.Process(
                target=run_worker_process,
                args=(i,)
            )
            worker.start()
            self.workers.append(worker)

        print(f"{self.worker_count}개의 작업자 프로세스가 시작되었습니다.")

    async def stop_workers(self):
        """작업자 프로세스를 종료합니다."""
        for worker in self.workers:
            worker.terminate()
            worker.join()
        self.workers = []
        print("모든 작업자 프로세스가 종료되었습니다.")

    async def distribute_task(self, task_data):
        """작업을 작업자에게 분배합니다."""
```

```
        # 작업 ID 생성
        task_id = f"task_{self.task_counter}"
        self.task_counter += 1

        # 작업 데이터를 작업자와 공유할 수 있는 형태로 직렬화
        serialized_task = json.dumps(task_data)

        # 작업 분배 (여기서는 Redis를 예시로 사용)
        import redis
        r = redis.Redis()
        r.lpush("task_queue", json.dumps({
            "task_id": task_id,
            "data": serialized_task
        }))

        # 작업 완료 대기
        result = None
        max_wait = 60  # 최대 60초 대기
        for _ in range(max_wait):
            result_data = r.get(f"result:{task_id}")
            if result_data:
                result = json.loads(result_data)
                r.delete(f"result:{task_id}")
                break
            await asyncio.sleep(1)

        if not result:
            return {"error": "작업 시간 초과"}

        return result

# 작업자 프로세스 실행 함수
def run_worker_process(worker_id):
    import redis
    r = redis.Redis()

    print(f"작업자 {worker_id} 시작")

    while True:
        # 작업 대기열에서 작업 가져오기
        task_data = r.brpop("task_queue", timeout=0)
        if not task_data:
            continue

        # 작업 처리
        task = json.loads(task_data[1])
        task_id = task["task_id"]
        data = json.loads(task["data"])

        try:
```

```
            # 실제 작업 처리
            result = process_task(data)

            # 결과 저장
            r.set(f"result:{task_id}", json.dumps(result))
            r.expire(f"result:{task_id}", 300)  # 5분 후 만료
        except Exception as e:
            # 오류 저장
            r.set(f"result:{task_id}", json.dumps({"error":
str(e)}))
            r.expire(f"result:{task_id}", 300)

# 사용 예시
distributor = WorkDistributor(worker_count=os.cpu_count())

@server.execute_tool
async def execute_tool(tool_name: str, params: Dict[str, Any]) ->
Any:
    if tool_name == "heavy-computation":
        # 무거운 계산 작업 분산 처리
        return await distributor.distribute_task(params)
```

성능 팁:
MCP 서버에서 가장 무거운 작업을 프로파일링하고, 이들을 비동기 작업, 캐싱, 또는 분산 처리로 최적화하는 것이 전체 성능 향상에 가장 큰 영향을 미칩니다.

고급

7. 고급 개발

이 섹션에서는 MCP의 고급 개발 주제를 다룹니다. 보안, 배포, 기업 환경 통합, 그리고 MCP 프로토콜의 확장에 대해 알아보겠습니다.

7.1. MCP 애플리케이션 보안

MCP 애플리케이션의 보안은 모든 개발 단계에서 고려해야 할 중요한 요소입니다. 이 섹션에서는 MCP 서버와 클라이언트의 보안 강화 방법을 알아보겠습니다.

입력 검증 및 위생 처리

사용자 입력과 외부 데이터는 항상 검증하고 적절히 처리해야 합니다:

```
# 입력 검증 및 위생 처리 구현
def validate_path(base_path, user_path):
    """사용자 입력 경로의 안전성을 검증합니다."""
    # 절대 경로로 변환
    abs_path =
os.path.abspath(os.path.normpath(os.path.join(base_path,
user_path)))

    # base_path 내부에 있는지 확인
    return abs_path.startswith(os.path.abspath(base_path))

def sanitize_sql_input(sql_input):
    """SQL 입력의 위생 처리를 수행합니다."""
    # SQL 인젝션 방지를 위한 간단한 처리
    # 실제로는 PreparedStatement나 ORM을 사용하는 것이 좋음
    dangerous_chars = [';', '--', '/*', '*/', 'xp_', 'UNION',
'SELECT', 'DROP', 'DELETE', 'UPDATE']
    sanitized = sql_input

    for char in dangerous_chars:
        sanitized = sanitized.replace(char, '')

    return sanitized

@server.execute_tool
async def execute_tool(tool_name: str, params: Dict[str, Any]) ->
Any:
    if tool_name == "read-file":
        # 기본 경로 설정 (안전한 기본 디렉토리)
        base_path = "/safe/read/directory"
        user_path = params.get("path", "")

        # 경로 검증
        if not validate_path(base_path, user_path):
            return {
                "error": "접근이 허용되지 않은 경로입니다",
                "code": "INVALID_PATH"
            }

        # 안전한 경로에서 파일 읽기
        try:
            full_path = os.path.join(base_path, user_path)
            with open(full_path, "r", encoding="utf-8") as f:
                content = f.read()
```

```
            return {"content": content}
        except Exception as e:
            return {"error": str(e)}

    elif tool_name == "query-database":
        # SQL 쿼리 위생 처리
        query = params.get("query", "")
        sanitized_query = sanitize_sql_input(query)

        # 직접 SQL 실행은 피하고 ORM이나 파라미터화된 쿼리 사용
        # 이 예시는 설명을 위한 것이며, 실제로는 더 안전한 방법 사용 권장
        return {"warning": "직접 SQL 쿼리는 권장되지 않습니다"}
```

인증 및 권한 관리

MCP 서버에 접근하는 클라이언트와 사용자의 인증 및 권한을 관리하는 방법:

```
# JWT 기반 인증 및 권한 관리
import jwt
import time
from functools import wraps

# 비밀 키 (실제로는 환경 변수 등에서 안전하게 관리)
SECRET_KEY = "your-secret-key"

def create_token(user_id, permissions, expires_in=3600):
    """인증 토큰을 생성합니다."""
    payload = {
        "user_id": user_id,
        "permissions": permissions,
        "exp": time.time() + expires_in
    }
    return jwt.encode(payload, SECRET_KEY, algorithm="HS256")

def verify_token(token):
    """토큰을 검증하고 페이로드를 반환합니다."""
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=
["HS256"])
        return payload
    except jwt.ExpiredSignatureError:
        raise ValueError("만료된 토큰입니다")
    except jwt.InvalidTokenError:
        raise ValueError("유효하지 않은 토큰입니다")

def require_permission(permission):
```

```
    """특정 권한이 필요한 도구 실행을 위한 데코레이터"""
    def decorator(func):
        @wraps(func)
        async def wrapper(tool_name, params):
            # 요청에서 토큰 추출
            token = params.pop("token", None)
            if not token:
                return {"error": "인증이 필요합니다", "code":
"AUTH_REQUIRED"}

            try:
                # 토큰 검증
                payload = verify_token(token)
                user_permissions = payload.get("permissions", [])

                # 권한 확인
                if permission not in user_permissions:
                    return {
                        "error": "접근 권한이 없습니다",
                        "code": "PERMISSION_DENIED"
                    }

                # 실제 함수 실행
                return await func(tool_name, params)
            except ValueError as e:
                return {"error": str(e), "code": "INVALID_TOKEN"}
        return wrapper
    return decorator

# 권한이 필요한 도구 실행
@server.execute_tool
@require_permission("file_write")
async def execute_tool(tool_name: str, params: Dict[str, Any]) ->
Any:
    if tool_name == "write-file":
        # 파일 쓰기 작업 (권한 확인 후)
        path = params.get("path", "")
        content = params.get("content", "")

        # ... 파일 쓰기 로직 ...
        return {"success": True}

    # ... 다른 도구 처리 ...
```

통신 보안

MCP 서버와 클라이언트 간의 통신 보안을 강화하는 방법:

```
# HTTPS 통신 설정 (FastAPI 예시)
from fastapi import FastAPI, Depends, HTTPException, Security
from fastapi.security import APIKeyHeader
import uvicorn
import ssl

app = FastAPI()

# API 키 인증
API_KEY_NAME = "X-API-Key"
API_KEY = "your-api-key"  # 실제로는 환경 변수 등에서 안전하게 관리

api_key_header = APIKeyHeader(name=API_KEY_NAME,
auto_error=False)

async def get_api_key(api_key: str = Security(api_key_header)):
    if api_key != API_KEY:
        raise HTTPException(
            status_code=403,
            detail="유효하지 않은 API 키입니다"
        )
    return api_key

@app.post("/execute-tool")
async def execute_tool_endpoint(
    request_data: dict,
    api_key: str = Depends(get_api_key)
):
    tool_name = request_data.get("tool_name")
    params = request_data.get("params", {})

    # MCP 서버에 도구 실행 요청 전달
    result = await execute_tool(tool_name, params)
    return result

if __name__ == "__main__":
    # SSL 컨텍스트 설정
    ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
    ssl_context.load_cert_chain('cert.pem', keyfile='key.pem')

    # HTTPS로 서버 실행
    uvicorn.run(
        app,
        host="0.0.0.0",
        port=8443,
        ssl_keyfile='key.pem',
```

```
        ssl_certfile='cert.pem'
    )
```

> ⚠ **보안 경고:**
> MCP 서버는 강력한 권한을 가질 수 있으므로, 모든 입력을 철저히 검증하고, 최소 권한 원칙을 적용하며, 중요한 작업에는 항상 인증과 권한 확인을 추가하세요.

7.2. 배포 전략

MCP 서버와 클라이언트를 다양한 환경에 배포하는 전략에 대해 알아보겠습니다.

컨테이너화

Docker를 사용하여 MCP 서버를 컨테이너화하는 방법:

```
# MCP 서버용 Dockerfile 예시
FROM python:3.9-slim

WORKDIR /app

# 필요한 패키지 설치
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# 애플리케이션 코드 복사
COPY . .

# 환경 변수 설정
ENV MCP_SERVER_PORT=8000
ENV LOG_LEVEL=info

# 포트 노출
EXPOSE 8000

# 서버 실행
CMD ["python", "server.py"]
```

Docker Compose를 사용하여 MCP 서버와 의존성을 함께 배포하는 방법:

```
# docker-compose.yml 예시
version: '3'

services:
  mcp-server:
    build: .
    ports:
      - "8000:8000"
    environment:
      - MCP_SERVER_PORT=8000
      - LOG_LEVEL=info
      - DATABASE_URL=postgresql://user:password@db:5432/mcp
    volumes:
      - ./data:/app/data
    depends_on:
      - db
    restart: unless-stopped

  db:
    image: postgres:13
    environment:
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=password
      - POSTGRES_DB=mcp
    volumes:
      - postgres_data:/var/lib/postgresql/data
    restart: unless-stopped

volumes:
  postgres_data:
```

서버리스 배포

AWS Lambda와 같은 서버리스 환경에 MCP 서버를 배포하는 방법:

```
# AWS Lambda를 위한 MCP 서버 코드 예시
import json
import asyncio
from mcp_sdk import Server

server = Server()

# 도구 등록
@server.list_tools
async def list_tools():
```

```
    # ... 도구 목록 정의 ...
    return tools

@server.execute_tool
async def execute_tool(tool_name, params):
    # ... 도구 실행 로직 ...
    return result

# Lambda 핸들러
def lambda_handler(event, context):
    # API Gateway 이벤트 처리
    body = json.loads(event.get('body', '{}'))
    method = body.get('method')

    # 비동기 함수 실행을 위한 이벤트 루프 설정
    loop = asyncio.get_event_loop()

    if method == 'list_tools':
        # 도구 목록 요청 처리
        tools = loop.run_until_complete(list_tools())
        return {
            'statusCode': 200,
            'body': json.dumps({
                'tools': [tool.__dict__ for tool in tools]
            })
        }
    elif method == 'execute_tool':
        # 도구 실행 요청 처리
        tool_name = body.get('tool_name')
        params = body.get('params', {})

        try:
            result =
loop.run_until_complete(execute_tool(tool_name, params))
            return {
                'statusCode': 200,
                'body': json.dumps({'result': result})
            }
        except Exception as e:
            return {
                'statusCode': 500,
                'body': json.dumps({'error': str(e)})
            }
    else:
        return {
            'statusCode': 400,
            'body': json.dumps({'error': 'Invalid method'})
        }
```

확장 가능한 인프라

Kubernetes를 사용하여 확장 가능한 MCP 서버 인프라를 구축하는 방법:

```
# Kubernetes 배포를 위한 MCP 서버 YAML 예시
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mcp-server
  labels:
    app: mcp-server
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mcp-server
  template:
    metadata:
      labels:
        app: mcp-server
    spec:
      containers:
      - name: mcp-server
        image: registry.example.com/mcp-server:latest
        ports:
        - containerPort: 8000
        env:
        - name: MCP_SERVER_PORT
          value: "8000"
        - name: LOG_LEVEL
          value: "info"
        - name: DATABASE_URL
          valueFrom:
            secretKeyRef:
              name: mcp-server-secrets
              key: database-url
        resources:
          limits:
            cpu: "1"
            memory: "1Gi"
          requests:
            cpu: "500m"
            memory: "512Mi"
        livenessProbe:
          httpGet:
            path: /health
            port: 8000
          initialDelaySeconds: 30
```

```
          periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /ready
            port: 8000
          initialDelaySeconds: 5
          periodSeconds: 5
---
apiVersion: v1
kind: Service
metadata:
  name: mcp-server-service
spec:
  selector:
    app: mcp-server
  ports:
  - port: 80
    targetPort: 8000
  type: LoadBalancer
```

7.3. 기업 환경 통합

MCP 애플리케이션을 기업 환경에 통합하는 방법에 대해 알아보겠습니다.

엔터프라이즈 시스템 통합

SAP와 같은 엔터프라이즈 시스템과 MCP 서버를 통합하는 예시:

```
# SAP 시스템 연동 MCP 서버 예시
import json
import requests
from mcp_sdk import Server
from mcp_sdk.types import ToolDefinition

# SAP 연결 설정
SAP_BASE_URL = "https://sap.example.com/api"
SAP_USER = "mcp_integration"
SAP_PASSWORD = "your-password"  # 실제로는 환경 변수 등에서 안전하게 관리

# SAP API 호출 함수
def call_sap_api(endpoint, method="GET", data=None):
    url = f"{SAP_BASE_URL}/{endpoint}"
    headers = {
        "Content-Type": "application/json",
```

```
        "Accept": "application/json"
    }

    response = requests.request(
        method,
        url,
        auth=(SAP_USER, SAP_PASSWORD),
        headers=headers,
        json=data
    )

    response.raise_for_status()
    return response.json()

# MCP 서버 설정
server = Server()

@server.list_tools
async def list_tools():
    return [
        ToolDefinition(
            name="get-customer",
            description="고객 정보를 조회합니다",
            input_schema={
                "type": "object",
                "properties": {
                    "customer_id": {
                        "type": "string",
                        "description": "고객 ID"
                    }
                },
                "required": ["customer_id"]
            }
        ),
        ToolDefinition(
            name="create-sales-order",
            description="판매 주문을 생성합니다",
            input_schema={
                "type": "object",
                "properties": {
                    "customer_id": {
                        "type": "string",
                        "description": "고객 ID"
                    },
                    "items": {
                        "type": "array",
                        "description": "주문 항목 목록",
                        "items": {
                            "type": "object",
                            "properties": {
```

```
                                "product_id": {"type": "string"},
                                "quantity": {"type": "number"}
                            },
                            "required": ["product_id",
"quantity"]
                        }
                    }
                },
                "required": ["customer_id", "items"]
            }
        )
    ]

@server.execute_tool
async def execute_tool(tool_name, params):
    if tool_name == "get-customer":
        customer_id = params.get("customer_id")
        try:
            # SAP 고객 정보 조회 API 호출
            customer_data =
call_sap_api(f"customers/{customer_id}")

            # 필요한 데이터 추출 및 가공
            return {
                "id": customer_data.get("id"),
                "name": customer_data.get("name"),
                "email": customer_data.get("email"),
                "address": customer_data.get("address"),
                "credit_limit": customer_data.get("creditLimit")
            }
        except Exception as e:
            return {"error": f"고객 정보 조회 실패: {str(e)}"}

    elif tool_name == "create-sales-order":
        try:
            # SAP 형식에 맞게 데이터 변환
            sap_order = {
                "customerId": params.get("customer_id"),
                "items": [
                    {
                        "materialNumber": item.get("product_id"),
                        "quantity": item.get("quantity")
                    }
                    for item in params.get("items", [])
                ]
            }

            # SAP 주문 생성 API 호출
            result = call_sap_api("sales-orders", method="POST",
data=sap_order)
```

```
            return {
                "order_id": result.get("orderNumber"),
                "status": result.get("status"),
                "total_amount": result.get("totalAmount"),
                "created_at": result.get("createdAt")
            }
        except Exception as e:
            return {"error": f"주문 생성 실패: {str(e)}"}

    return {"error": f"알 수 없는 도구: {tool_name}"}
```

데이터 거버넌스 및 감사

MCP 서버의 데이터 거버넌스 및 감사 로깅 구현 방법:

```
# 감사 로깅 및 데이터 거버넌스 기능 구현
import uuid
import datetime
import logging
from functools import wraps

# 감사 로그 설정
audit_logger = logging.getLogger("mcp.audit")
audit_logger.setLevel(logging.INFO)

# 파일 핸들러 설정
audit_file_handler = logging.FileHandler("audit.log")
audit_file_handler.setFormatter(
    logging.Formatter('%(asctime)s - %(message)s')
)
audit_logger.addHandler(audit_file_handler)

# 감사 로그 데이터베이스 저장 함수
async def save_audit_log(log_entry):
    """감사 로그를 데이터베이스에 저장합니다."""
    import asyncpg

    conn = await
asyncpg.connect("postgresql://user:password@localhost/auditdb")
    try:
        await conn.execute(
            """
            INSERT INTO audit_logs
            (timestamp, user_id, action, tool_name, parameters,
result_status, request_id)
```

```
            VALUES ($1, $2, $3, $4, $5, $6, $7)
            """,
            log_entry["timestamp"],
            log_entry["user_id"],
            log_entry["action"],
            log_entry["tool_name"],
            json.dumps(log_entry["parameters"]),
            log_entry["result_status"],
            log_entry["request_id"]
        )
    finally:
        await conn.close()

# 감사 로깅 데코레이터
def audit_log(func):
    @wraps(func)
    async def wrapper(tool_name, params):
        # 요청 ID 생성
        request_id = str(uuid.uuid4())

        # 사용자 ID 추출 (인증 시스템에서 가져와야 함)
        user_id = params.get("_user_id", "anonymous")

        # 민감 데이터 필터링된 파라미터
        filtered_params = {
            k: ("*****" if k in ["password", "credit_card",
"ssn"] else v)
            for k, v in params.items()
            if not k.startswith("_")  # 내부 파라미터 제외
        }

        # 시작 시간 기록
        start_time = datetime.datetime.now()

        # 감사 로그 시작 항목
        start_log = {
            "timestamp": start_time.isoformat(),
            "user_id": user_id,
            "action": "TOOL_EXECUTION_START",
            "tool_name": tool_name,
            "parameters": filtered_params,
            "request_id": request_id
        }

        # 시작 로그 기록
        audit_logger.info(f"REQUEST_START:
{json.dumps(start_log)}")

        try:
            # 실제 함수 실행
```

```
            result = await func(tool_name, params)

            # 성공 상태 설정
            status = "SUCCESS"

            return result
        except Exception as e:
            # 실패 상태 설정
            status = f"ERROR: {str(e)}"
            raise
        finally:
            # 종료 시간 기록
            end_time = datetime.datetime.now()
            duration_ms = int((end_time -
start_time).total_seconds() * 1000)

            # 감사 로그 종료 항목
            end_log = {
                "timestamp": end_time.isoformat(),
                "user_id": user_id,
                "action": "TOOL_EXECUTION_END",
                "tool_name": tool_name,
                "parameters": filtered_params,
                "result_status": status,
                "duration_ms": duration_ms,
                "request_id": request_id
            }

            # 종료 로그 기록
            audit_logger.info(f"REQUEST_END:
{json.dumps(end_log)}")

            # 데이터베이스에 로그 저장 (비동기)
            asyncio.create_task(save_audit_log(end_log))

    return wrapper

# MCP 서버에 감사 로깅 적용
@server.execute_tool
@audit_log
async def execute_tool(tool_name, params):
    # ... 도구 실행 로직 ...
    return result
```

SSO 및 기업 인증 통합

기업 환경의 Single Sign-On(SSO) 시스템과 MCP 서버를 통합하는 방법:

```
# Azure AD 인증 통합 예시
import msal
import requests
from fastapi import FastAPI, Depends, HTTPException
from fastapi.security import OAuth2AuthorizationCodeBearer
from mcp_sdk import Server

app = FastAPI()
mcp_server = Server()

# Azure AD 설정
AZURE_AD_TENANT_ID = "your-tenant-id"
AZURE_AD_CLIENT_ID = "your-client-id"
AZURE_AD_CLIENT_SECRET = "your-client-secret"
AZURE_AD_REDIRECT_URI = "https://your-
service.example.com/auth/callback"

# MSAL 앱 설정
msal_app = msal.ConfidentialClientApplication(
    AZURE_AD_CLIENT_ID,

authority=f"https://login.microsoftonline.com/{AZURE_AD_TENANT_ID
}",
    client_credential=AZURE_AD_CLIENT_SECRET
)

# OAuth2 스키마 설정
oauth2_scheme = OAuth2AuthorizationCodeBearer(

authorizationUrl=f"https://login.microsoftonline.com/{AZURE_AD_TE
NANT_ID}/oauth2/v2.0/authorize",

tokenUrl=f"https://login.microsoftonline.com/{AZURE_AD_TENANT_ID}
/oauth2/v2.0/token",
)

# 사용자 권한 검증
async def validate_token(token: str = Depends(oauth2_scheme)):
    # 토큰 검증
    try:
        # Microsoft Graph API를 호출하여 토큰 검증
        headers = {"Authorization": f"Bearer {token}"}
        response = requests.get(
            "https://graph.microsoft.com/v1.0/me",
            headers=headers
        )

        if response.status_code != 200:
            raise HTTPException(
```

```
                status_code=401,
                detail="유효하지 않은 인증 토큰입니다"
            )

        # 사용자 정보 반환
        user_info = response.json()
        return {
            "id": user_info.get("id"),
            "email": user_info.get("userPrincipalName"),
            "name": user_info.get("displayName")
        }
    except Exception as e:
        raise HTTPException(
            status_code=401,
            detail=f"인증 실패: {str(e)}"
        )

# 인증 라우트
@app.get("/auth/login")
async def login():
    # Azure AD 로그인 페이지로 리디렉션
    auth_url = msal_app.get_authorization_request_url(
        scopes=["User.Read"],
        redirect_uri=AZURE_AD_REDIRECT_URI
    )
    return {"auth_url": auth_url}

@app.get("/auth/callback")
async def auth_callback(code: str):
    # 인증 코드로 토큰 획득
    result = msal_app.acquire_token_by_authorization_code(
        code,
        scopes=["User.Read"],
        redirect_uri=AZURE_AD_REDIRECT_URI
    )

    if "error" in result:
        raise HTTPException(
            status_code=401,
            detail=f"인증 실패: {result.get('error_description')}"
        )

    # 토큰 반환
    return {"token": result.get("access_token")}

# MCP 도구 실행 엔드포인트
@app.post("/execute-tool")
async def execute_tool_endpoint(
    request_data: dict,
    user: dict = Depends(validate_token)
```

```
):
    # 도구 이름과 파라미터 추출
    tool_name = request_data.get("tool_name")
    params = request_data.get("params", {})

    # 인증된 사용자 정보 추가
    params["_user_id"] = user.get("id")
    params["_user_email"] = user.get("email")

    # MCP 서버에 도구 실행 요청 전달
    result = await mcp_server.execute_tool_handler(tool_name,
params)
    return result
```

7.4. MCP 프로토콜 확장

MCP 프로토콜을 확장하여 더 복잡한 시나리오를 지원하는 방법에 대해 알아보겠습니다.

커스텀 프로토콜 확장

표준 MCP 프로토콜을 넘어 추가 기능을 제공하는 확장 구현:

```
# 프로토콜 확장 예시: 구독 기능 추가
from mcp_sdk import Server
from mcp_sdk.types import NotificationMessage
import asyncio
import uuid

class ExtendedMCPServer(Server):
    def __init__(self):
        super().__init__()
        self.subscriptions = {}
        self.subscription_tasks = {}

    async def handle_custom_request(self, method, params):
        """커스텀 요청 처리"""
        if method == "subscribe":
            return await self._handle_subscribe(params)
        elif method == "unsubscribe":
            return await self._handle_unsubscribe(params)
        else:
            # 알 수 없는 메서드는 부모 클래스로 전달
            return await super().handle_custom_request(method,
params)
```

```
    async def _handle_subscribe(self, params):
        """구독 요청 처리"""
        event_type = params.get("event_type")
        filters = params.get("filters", {})

        if not event_type:
            return {"error": "event_type이 필요합니다"}

        # 구독 ID 생성
        subscription_id = str(uuid.uuid4())

        # 구독 정보 저장
        self.subscriptions[subscription_id] = {
            "event_type": event_type,
            "filters": filters
        }

        # 구독 태스크 시작 (예: 주기적인 이벤트 검사)
        task =
asyncio.create_task(self._subscription_task(subscription_id))
        self.subscription_tasks[subscription_id] = task

        return {"subscription_id": subscription_id}

    async def _handle_unsubscribe(self, params):
        """구독 취소 요청 처리"""
        subscription_id = params.get("subscription_id")

        if not subscription_id or subscription_id not in
self.subscriptions:
            return {"error": "유효하지 않은 subscription_id입니다"}

        # 구독 태스크 취소
        if subscription_id in self.subscription_tasks:
            self.subscription_tasks[subscription_id].cancel()
            del self.subscription_tasks[subscription_id]

        # 구독 정보 제거
        del self.subscriptions[subscription_id]

        return {"success": True}

    async def _subscription_task(self, subscription_id):
        """구독 이벤트 모니터링 및 알림 전송"""
        try:
            subscription = self.subscriptions[subscription_id]
            event_type = subscription["event_type"]
            filters = subscription["filters"]
```

```
            while True:
                # 이벤트 검사 로직 (예: 데이터베이스 폴링, 파일 변경 감지 등)
                events = await self._check_events(event_type,
filters)

                # 이벤트가 있으면 알림 전송
                for event in events:
                    await self.send_notification(
                        NotificationMessage(
                            method="event",
                            params={
                                "subscription_id":
subscription_id,
                                "event_type": event_type,
                                "data": event
                            }
                        )
                    )

                # 주기적인 대기
                await asyncio.sleep(5)
        except asyncio.CancelledError:
            # 구독 취소 시 정리 작업
            pass
        except Exception as e:
            print(f"구독 태스크 오류: {str(e)}")

    async def _check_events(self, event_type, filters):
        """이벤트 검사 로직"""
        # 실제 구현은 이벤트 유형에 따라 다름
        # 예시: 데이터베이스 변경 감지
        if event_type == "database_change":
            table = filters.get("table")
            last_id = filters.get("last_id", 0)

            # 데이터베이스 변경 사항 조회
            # 실제로는 데이터베이스 연결 및 쿼리 실행
            changes = [
                {"id": 1, "table": "users", "action": "insert",
"data": {}},
                {"id": 2, "table": "orders", "action": "update",
"data": {}}
            ]

            # 필터 적용
            filtered_changes = [
                change for change in changes
                if (not table or change["table"] == table) and
change["id"] > last_id
            ]
```

```
            return filtered_changes

        # 예시: 파일 시스템 변경 감지
        elif event_type == "file_change":
            path = filters.get("path", "")

            # 파일 시스템 변경 사항 조회
            # 실제로는 파일 시스템 모니터링 로직
            changes = [
                {"path": "/data/file1.txt", "action": "modified",
"timestamp": "..."},
                {"path": "/data/file2.txt", "action": "created",
"timestamp": "..."}
            ]

            # 필터 적용
            filtered_changes = [
                change for change in changes
                if not path or change["path"].startswith(path)
            ]

            return filtered_changes

        return []
```

실시간 컬래버레이션 지원

여러 사용자가 동시에 작업할 수 있는 실시간 협업 기능 구현:

```
# 실시간 협업 지원 MCP 서버 예시
from mcp_sdk import Server
import asyncio
import json

class CollaborativeMCPServer(Server):
    def __init__(self):
        super().__init__()
        self.document_sessions = {}
        self.user_sessions = {}

    async def handle_custom_request(self, method, params):
        """커스텀 요청 처리"""
        if method == "join_document":
            return await self._handle_join_document(params)
        elif method == "leave_document":
```

```
            return await self._handle_leave_document(params)
        elif method == "update_document":
            return await self._handle_update_document(params)
        else:
            # 알 수 없는 메서드는 부모 클래스로 전달
            return await super().handle_custom_request(method,
params)

    async def _handle_join_document(self, params):
        """문서 세션 참여 처리"""
        document_id = params.get("document_id")
        user_id = params.get("user_id")

        if not document_id or not user_id:
            return {"error": "document_id와 user_id가 필요합니다"}

        # 문서 세션 초기화 (없는 경우)
        if document_id not in self.document_sessions:
            self.document_sessions[document_id] = {
                "users": set(),
                "content": "",
                "version": 0
            }

        # 사용자를 세션에 추가
        self.document_sessions[document_id]["users"].add(user_id)

        # 사용자 세션 정보 업데이트
        if user_id not in self.user_sessions:
            self.user_sessions[user_id] = set()
        self.user_sessions[user_id].add(document_id)

        # 다른 사용자에게 참여 알림
        await self._notify_document_users(
            document_id,
            {
                "type": "user_joined",
                "user_id": user_id,
                "document_id": document_id
            },
            exclude_user=user_id
        )

        # 현재 문서 상태 반환
        return {
            "content": self.document_sessions[document_id]
["content"],
            "version": self.document_sessions[document_id]
["version"],
            "users": list(self.document_sessions[document_id]
```

```
["users"])
        }

    async def _handle_leave_document(self, params):
        """문서 세션 나가기 처리"""
        document_id = params.get("document_id")
        user_id = params.get("user_id")

        if not document_id or not user_id:
            return {"error": "document_id와 user_id가 필요합니다"}

        # 문서 세션에서 사용자 제거
        if document_id in self.document_sessions:
            if user_id in self.document_sessions[document_id]
["users"]:
                self.document_sessions[document_id]
["users"].remove(user_id)

            # 빈 세션 정리
            if not self.document_sessions[document_id]["users"]:
                del self.document_sessions[document_id]

        # 사용자 세션 정보 업데이트
        if user_id in self.user_sessions:
            if document_id in self.user_sessions[user_id]:
                self.user_sessions[user_id].remove(document_id)

            # 빈 세션 정리
            if not self.user_sessions[user_id]:
                del self.user_sessions[user_id]

        # 다른 사용자에게 나가기 알림
        await self._notify_document_users(
            document_id,
            {
                "type": "user_left",
                "user_id": user_id,
                "document_id": document_id
            }
        )

        return {"success": True}

    async def _handle_update_document(self, params):
        """문서 업데이트 처리"""
        document_id = params.get("document_id")
        user_id = params.get("user_id")
        content = params.get("content")
        client_version = params.get("version")
```

```
        if not document_id or not user_id or content is None or
client_version is None:
            return {"error": "필수 파라미터가 누락되었습니다"}

        # 문서 세션 확인
        if document_id not in self.document_sessions:
            return {"error": "존재하지 않는 문서 세션입니다"}

        # 버전 충돌 확인
        server_version = self.document_sessions[document_id]
["version"]
        if client_version != server_version:
            return {
                "error": "버전 충돌",
                "server_version": server_version,
                "current_content":
self.document_sessions[document_id]["content"]
            }

        # 문서 업데이트
        self.document_sessions[document_id]["content"] = content
        self.document_sessions[document_id]["version"] += 1
        new_version = self.document_sessions[document_id]
["version"]

        # 다른 사용자에게 업데이트 알림
        await self._notify_document_users(
            document_id,
            {
                "type": "document_updated",
                "user_id": user_id,
                "document_id": document_id,
                "content": content,
                "version": new_version
            },
            exclude_user=user_id
        )

        return {
            "success": True,
            "version": new_version
        }

    async def _notify_document_users(self, document_id, message,
exclude_user=None):
        """문서 세션의 사용자들에게 알림 전송"""
        if document_id not in self.document_sessions:
            return

        for user_id in self.document_sessions[document_id]
```

```
["users"]:
            if exclude_user and user_id == exclude_user:
                continue

            # 실제 구현에서는 사용자별 클라이언트 연결을 통해 알림 전송
            # 여기서는 단순화를 위해 서버에서 알림 로그만 출력
            print(f"알림 전송 - 사용자: {user_id}, 메시지:
{json.dumps(message)}")

            # 실제 구현에서는 클라이언트에 알림 전송
            # await self.send_notification_to_user(user_id,
message)
```

AI 모델 간 연동 지원

여러 AI 모델을 연결하고 활용하는 MCP 서버 구현:

```
# AI 모델 간 연동을 위한 MCP 서버 확장
from mcp_sdk import Server
import anthropic
import openai
import os

class AIModelConnectorServer(Server):
    def __init__(self):
        super().__init__()
        # Anthropic 클라이언트 초기화
        self.anthropic_client = anthropic.Anthropic(
            api_key=os.environ.get("ANTHROPIC_API_KEY")
        )

        # OpenAI 클라이언트 초기화
        self.openai_client = openai.OpenAI(
            api_key=os.environ.get("OPENAI_API_KEY")
        )

    @server.list_tools
    async def list_tools(self):
        """사용 가능한 도구 목록 반환"""
        return [
            ToolDefinition(
                name="anthropic-generate",
                description="Anthropic Claude를 사용하여 텍스트 생성",
                input_schema={
                    "type": "object",
                    "properties": {
```

```
                    "prompt": {
                        "type": "string",
                        "description": "생성할 텍스트의 프롬프트"
                    },
                    "max_tokens": {
                        "type": "integer",
                        "description": "생성할 최대 토큰 수",
                        "default": 1000
                    },
                    "model": {
                        "type": "string",
                        "description": "사용할 Claude 모델",
                        "default": "claude-3-opus-20240229"
                    }
                },
                "required": ["prompt"]
            }
        ),
        ToolDefinition(
            name="openai-generate",
            description="OpenAI GPT를 사용하여 텍스트 생성",
            input_schema={
                "type": "object",
                "properties": {
                    "prompt": {
                        "type": "string",
                        "description": "생성할 텍스트의 프롬프트"
                    },
                    "max_tokens": {
                        "type": "integer",
                        "description": "생성할 최대 토큰 수",
                        "default": 1000
                    },
                    "model": {
                        "type": "string",
                        "description": "사용할 GPT 모델",
                        "default": "gpt-4"
                    }
                },
                "required": ["prompt"]
            }
        ),
        ToolDefinition(
            name="openai-vision",
            description="OpenAI Vision을 사용하여 이미지 분석",
            input_schema={
                "type": "object",
                "properties": {
                    "image_url": {
                        "type": "string",
```

```
                            "description": "분석할 이미지 URL"
                        },
                        "prompt": {
                            "type": "string",
                            "description": "이미지 분석을 위한 프롬프
트",
                            "default": "이 이미지에 무엇이 보이나요?"
                        }
                    },
                    "required": ["image_url"]
                }
            ),
            ToolDefinition(
                name="model-compare",
                description="여러 AI 모델의 응답을 비교",
                input_schema={
                    "type": "object",
                    "properties": {
                        "prompt": {
                            "type": "string",
                            "description": "모델에게 제시할 프롬프트"
                        },
                        "models": {
                            "type": "array",
                            "description": "비교할 모델 목록",
                            "items": {
                                "type": "string",
                                "enum": ["claude-3-opus",
"claude-3-sonnet", "gpt-4", "gpt-3.5-turbo"]
                            },
                            "default": ["claude-3-opus", "gpt-4"]
                        }
                    },
                    "required": ["prompt"]
                }
            )
        ]

    @server.execute_tool
    async def execute_tool(self, tool_name, params):
        """도구 실행"""
        if tool_name == "anthropic-generate":
            prompt = params.get("prompt", "")
            max_tokens = params.get("max_tokens", 1000)
            model = params.get("model", "claude-3-opus-20240229")

            try:
                # Claude API 호출
                message = self.anthropic_client.messages.create(
                    model=model,
```

```
                    max_tokens=max_tokens,
                    messages=[
                        {"role": "user", "content": prompt}
                    ]
                )

                return {
                    "text": message.content[0].text,
                    "model": model,
                    "tokens": {
                        "input": message.usage.input_tokens,
                        "output": message.usage.output_tokens
                    }
                }
            except Exception as e:
                return {"error": f"Claude API 오류: {str(e)}"}

        elif tool_name == "openai-generate":
            prompt = params.get("prompt", "")
            max_tokens = params.get("max_tokens", 1000)
            model = params.get("model", "gpt-4")

            try:
                # OpenAI API 호출
                completion =
self.openai_client.chat.completions.create(
                    model=model,
                    messages=[
                        {"role": "user", "content": prompt}
                    ],
                    max_tokens=max_tokens
                )

                return {
                    "text":
completion.choices[0].message.content,
                    "model": model,
                    "tokens": {
                        "total": completion.usage.total_tokens
                    }
                }
            except Exception as e:
                return {"error": f"OpenAI API 오류: {str(e)}"}

        elif tool_name == "openai-vision":
            image_url = params.get("image_url", "")
            prompt = params.get("prompt", "이 이미지에 무엇이 보이나
요?")

            try:
```

```
                # OpenAI Vision API 호출
                response =
self.openai_client.chat.completions.create(
                    model="gpt-4-vision-preview",
                    messages=[
                        {
                            "role": "user",
                            "content": [
                                {"type": "text", "text": prompt},
                                {
                                    "type": "image_url",
                                    "image_url": {"url":
image_url}
                                }
                            ]
                        }
                    ],
                    max_tokens=1000
                )

                return {
                    "analysis":
response.choices[0].message.content,
                    "model": "gpt-4-vision-preview",
                    "tokens": {
                        "total": response.usage.total_tokens
                    }
                }
            except Exception as e:
                return {"error": f"OpenAI Vision API 오류:
{str(e)}"}

        elif tool_name == "model-compare":
            prompt = params.get("prompt", "")
            models = params.get("models", ["claude-3-opus", "gpt-
4"])

            results = {}
            errors = {}

            # 모든 모델에 대해 병렬로 요청 처리
            async def fetch_model_response(model):
                try:
                    if model.startswith("claude"):
                        # Claude 모델 사용
                        actual_model = model + "-20240229" # 실제
모델 이름 형식으로 변환
                        message =
self.anthropic_client.messages.create(
                            model=actual_model,
```

```
                            max_tokens=1000,
                            messages=[
                                {"role": "user", "content":
prompt}
                            ]
                        )
                        return {
                            "text": message.content[0].text,
                            "tokens": {
                                "input":
message.usage.input_tokens,
                                "output":
message.usage.output_tokens
                            }
                        }
                    else:
                        # OpenAI 모델 사용
                        completion =
self.openai_client.chat.completions.create(
                            model=model,
                            messages=[
                                {"role": "user", "content":
prompt}
                            ],
                            max_tokens=1000
                        )
                        return {
                            "text":
completion.choices[0].message.content,
                            "tokens": {
                                "total":
completion.usage.total_tokens
                            }
                        }
                except Exception as e:
                    return {"error": str(e)}

            # 모든 모델에 대해 요청 실행
            tasks = []
            for model in models:
                tasks.append(fetch_model_response(model))

            model_responses = await asyncio.gather(*tasks)

            # 결과 정리
            for model, response in zip(models, model_responses):
                if "error" in response:
                    errors[model] = response["error"]
                else:
                    results[model] = response
```

```
        return {
            "prompt": prompt,
            "results": results,
            "errors": errors
        }

    return {"error": f"알 수 없는 도구: {tool_name}"}
```

고급

8. 사례 연구

이 섹션에서는 실제 환경에서 MCP를 활용한 사례와 그로부터 얻은 교훈을 살펴봅니다.

8.1. 실제 활용 사례

개발 환경 통합 사례

사례: 소프트웨어 개발 지원 시스템
기업: 엔터프라이즈 소프트웨어 개발 회사
도전 과제: 대규모 코드베이스 이해, 코드 변경 리뷰, 문서화 자동화 필요
MCP 솔루션:

- 코드 저장소 MCP 서버: 로컬 코드베이스 검색 및 분석
- 문서 MCP 서버: 기술 문서 조회 및 업데이트
- Git MCP 서버: 변경 사항 관리 및 PR 자동화

결과:

- 코드 검토 시간 40% 감소
- 새로운 팀원의 코드베이스 이해 시간 단축
- 자동 생성된 PR 설명 및 코드 문서화

데이터 분석 사례

사례: 금융 데이터 분석 자동화
기업: 중견 금융 서비스 회사
도전 과제: 다양한 시스템에 분산된 재무 데이터 통합 및 분석 자동화
MCP 솔루션:

- 데이터베이스 MCP 서버: 여러 데이터베이스 쿼리 통합
- Excel MCP 서버: 기존 재무 모델 및 스프레드시트 접근
- 시각화 MCP 서버: 데이터 시각화 및 대시보드 생성

결과:

- 분석 보고서 작성 시간 70% 단축
- 산재된 데이터 소스 접근성 향상
- 인사이트 도출 및 의사결정 속도 개선

기업 지식 관리 사례

사례: 글로벌 제조 기업의 지식 관리 시스템
기업: 다국적 제조 기업
도전 과제: 사내 지식 접근성 향상, 기술 문서 통합, 여러 언어 장벽 극복
MCP 솔루션:

- 문서 저장소 MCP 서버: 사내 문서 및 매뉴얼 접근
- 번역 MCP 서버: 다국어 문서 실시간 번역
- QA 시스템 MCP 서버: 기존 FAQ 및 지식 베이스 활용

결과:

- 직원의 정보 검색 시간 60% 단축
- 다국어 팀 간 협업 개선
- 신규 직원 온보딩 시간 단축
- 지식 격차 해소 및 문제 해결 시간 단축

8.2. 교훈과 모범 사례

기술적 교훈

1. **모듈화된 서버 설계**

   범용 서버보다 특정 기능에 집중한 작은 서버들이 더 효과적입니다. 이는 유지보수 용이성, 성능 최적화, 보안 격리에 도움이 됩니다.

2. **오류 회복력 중요성**

   실제 환경에서는 네트워크 문제, 서비스 중단 등 다양한 오류가 발생할 수 있습니다. 재시도 로직, 단계적 복구, 우아한 실패 처리가 필수적입니다.

3. **성능 병목 식별**

   데이터 전송량, API 호출 빈도, 메모리 사용량 최적화가 사용자 경험에 큰 영향을 미칩니다. 초기부터 성능 모니터링 도구를 통합하는 것이 좋습니다.

4. **입력 검증의 중요성**

   MCP 서버는 파일 시스템, 데이터베이스 등 민감한 자원에 접근할 수 있으므로, 모든 입력에 대한 철저한 검증이 필수적입니다.

프로젝트 관리 교훈

1. **점진적 구현**

   한 번에 모든 것을 구현하려 하기보다, 가장 가치 있는 기능부터 점진적으로 구현하고 테스트하는 것이 효과적입니다.

2. **사용자 피드백 중요성**

   AI와 사용자 간의 상호작용은 예상과 다를 수 있습니다. 초기부터 실제 사용자 테스트를 통합하고 지속적인 피드백 루프를 구축하세요.

3. **보안 초기화**

   "나중에 보안을 추가하자"는 접근 방식은 실패의 지름길입니다. 초기 설계 단계부터 보안을 고려하고 정기적인 보안 검토를 수행하세요.

4. **문서화 중요성**

   복잡한 통합 시스템에서는 명확한 문서화가 필수적입니다. API 스펙, 구성 옵션, 배포 가이드, 문제 해결 팁을 포함한 포괄적 문서를 유지하세요.

모범 사례

영역	모범 사례
서버 설계	• 단일 책임 원칙 적용 • 명확한 도구 이름과 설명 제공 • 입력 스키마 철저한 정의 • 모든 오류 상황 처리
배포	• 컨테이너화를 통한 환경 일관성 유지 • 자동화된 테스트 및 배포 파이프라인 구축 • 점진적 롤아웃 및 롤백 전략 수립 • 버전 관리 및 변경 이력 추적
운영	• 포괄적 로깅 및 모니터링 구현 • 성능 지표 수집 및 분석 • 알림 시스템 구축 • 정기적인 보안 점검 및 업데이트

영역	모범 사례
사용자 경험	• 명확한 오류 메시지 및 해결 방법 제공 • 진행 상황 표시 및 피드백 • 응답 시간 최적화 • 사용자 피드백 채널 구축

중급

9. 미래 방향성

MCP는 계속 발전하고 있는 프로토콜입니다. 이 섹션에서는 MCP의 미래 방향성과 새로운 기회에 대해 탐색합니다.

9.1. MCP 로드맵

Anthropic과 커뮤니티가 발표한 MCP의 향후 방향성은 다음과 같습니다:

프로토콜 확장

- **리소스 관리 개선**: 더 정교한 리소스 접근 제어 및 관리 기능
- **멀티모달 지원 강화**: 이미지, 오디오, 비디오와 같은 멀티모달 데이터 처리 기능
- **양방향 스트리밍**: 클라이언트와 서버 간 양방향 실시간 스트리밍 지원
- **세션 관리**: 상태 유지 세션 및 컨텍스트 보존 지원
- **캐싱 표준화**: 응답 캐싱 및 재사용을 위한 표준 메커니즘

보안 향상

- **세분화된 권한 모델**: 도구 및 리소스별 상세 권한 제어
- **엔드 투 엔드 암호화**: 더 강력한 데이터 보안 지원
- **인증 표준화**: OAuth, SAML 등과의 통합 지원
- **감사 기능 강화**: 더 정교한 활동 로깅 및 감사 기능

개발자 경험 개선

- **더 많은 언어 지원**: 추가 프로그래밍 언어에 대한 SDK 확장
- **개발 도구 통합**: IDE 플러그인 및 디버깅 도구 개선

- **서버 배포 간소화**: 더 쉬운 배포 및 관리 옵션
- **문서 및 예제 확대**: 더 많은 용례와 튜토리얼 제공

9.2. 주요 동향 및 기회

MCP 생태계에서 나타나는 주요 동향과 기회는 다음과 같습니다:

분야별 특화 서버

다양한 산업 분야에 맞춘 특화된 MCP 서버가 등장하고 있습니다:

의료
의료 기록 접근, 의학 연구 데이터 분석, 의료 이미지 처리

법률
판례 검색, 법률 문서 분석, 계약서 작성 지원

금융
금융 데이터 분석, 투자 포트폴리오 관리, 리스크 평가

교육
학습 자료 접근, 학생 진도 추적, 맞춤형 학습 경로

제조
생산 데이터 분석, 품질 관리, 공급망 최적화

고급 MCP 에이전트

MCP를 활용한 복잡한 태스크를 자율적으로 수행하는 AI 에이전트의 발전이 예상됩니다:

자율 개발 에이전트
코드 작성, 테스트, 배포를 자동화하는 에이전트

데이터 분석 에이전트
데이터 탐색, 시각화, 인사이트 도출을 수행

비즈니스 프로세스 에이전트

기업 워크플로우 자동화 및 최적화

리서치 에이전트
다양한 소스에서 정보를 수집하고 종합하는 에이전트

MCP 기반 애플리케이션 플랫폼

MCP를 중심으로 한 새로운 애플리케이션 개발 플랫폼의 부상이 예상됩니다:

AI 기반 개발 환경
MCP를 통해 AI와 협업하는 개발 도구

비즈니스 인텔리전스 플랫폼
데이터와 인사이트를 쉽게 접근할 수 있는 플랫폼

지식 관리 시스템
기업 지식과 정보를 통합하고 활용하는 플랫폼

통합 워크플로우 도구
다양한 도구와 시스템을 연결하는 통합 환경

커뮤니티 및 오픈 소스 발전

MCP 커뮤니티와 오픈 소스 생태계의 활발한 성장이 기대됩니다:

서버 라이브러리 확장
다양한 시스템과 서비스를 위한 오픈 소스 MCP 서버

개발 도구 생태계
MCP 개발을 위한 도구, 템플릿, 프레임워크

표준화 노력
호환성과 상호운용성을 위한 커뮤니티 표준

교육 및 지식 공유
튜토리얼, 워크샵, 학습 자료의 확대

기초

10. 결론

Model Context Protocol(MCP)은 AI 모델과 외부 시스템을 연결하는 혁신적인 프로토콜로, AI의 활용 가능성을 크게 확장하고 있습니다. 이 전자책에서는 MCP의 기본 개념부터 실제 구현, 그리고 미래 방향성까지 포괄적으로 다루었습니다.

MCP의 주요 가치

MCP가 제공하는 주요 가치는 다음과 같습니다:

- **AI의 기능 확장**: AI가 다양한 외부 시스템과 통합하여 더 강력한 기능을 제공할 수 있습니다.
- **표준화된 통합**: 다양한 데이터 소스와 도구를 일관된 방식으로 연결할 수 있습니다.
- **개발 복잡성 감소**: 표준화된 프로토콜을 통해 통합 개발의 복잡성을 줄일 수 있습니다.
- **재사용성 증가**: 한 번 개발한 서버를 여러 AI 애플리케이션에서 재사용할 수 있습니다.
- **보안 및 제어**: 중요한 데이터와 시스템에 대한 접근을 안전하게 제어할 수 있습니다.

MCP 여정을 시작하는 방법

MCP 활용을 시작하기 위한 단계별 접근 방법:

1. **학습 및 탐색**
   MCP의 기본 개념과 아키텍처를 이해합니다.
2. **기존 서버 활용**
   공개된 MCP 서버를 설치하고 테스트해 봅니다.
3. **소규모 서버 개발**
   단일 기능에 집중한 간단한 MCP 서버를 개발합니다.
4. **통합 및 확장**
   여러 서버를 통합하고 기능을 확장합니다.
5. **최적화 및 배포**
   성능을 최적화하고 프로덕션 환경에 배포합니다.
6. **커뮤니티 참여**
   MCP 커뮤니티에 참여하여 지식을 공유하고 발전시킵니다.

마무리

MCP는 AI와 데이터, 도구, 시스템 간의 벽을 허물어 더 강력하고 유용한 AI 애플리케이션의 개발을 가능하게 합니다. Claude와 같은 AI 모델의 능력을 확장하고, 실제 비즈니스 문제를 해결하며, 새로운 가능성을 탐색하는 데 MCP가 중요한 역할을 할 것입니다.

이 전자책이 여러분의 MCP 여정에 유용한 가이드가 되기를 바랍니다. MCP 생태계가 계속 성장하고 발전함에 따라, 여러분이 개발하는 혁신적인 솔루션이 AI의 미래를 함께 만들어 나갈 것입니다.

> *MCP를 통해 AI의 새로운 가능성을 탐색하고, 실제 가치를 창출하는 여정을 시작하세요!*
