Building Production MCP Servers: FastMCP, Python & Docker

A Practical Guide for Hackathon-Speed Development

Version: November 2025
Target Audience: Python developers building MCP servers in 6-hour hackathons
Prerequisites: Python 3.10+, Docker installed

Table of Contents

1. Executive Summary {#executive-summary}

The Model Context Protocol (MCP) is an open standard that enables AI assistants to connect to external data sources and tools. Think of it as **USB-C for AI applications**—a universal connector that works across Claude Desktop, VS Code, Cursor, OpenAI clients, and more[1] [2].

Key Points:

- **MCP servers expose tools, resources, and prompts** to LLM clients via JSON-RPC 2.0
- **FastMCP** is the fastest way to build Python MCP servers (released mid-2024, actively maintained)
- **Current spec version:** 2025-06-18, with next version releasing November 25, 2025[1]
- **Runtime secrets** (API keys) are configured **per-user in the MCP client**, not baked into your server
- **Docker Hub integration** allows publishing containerized MCP servers for one-click installation

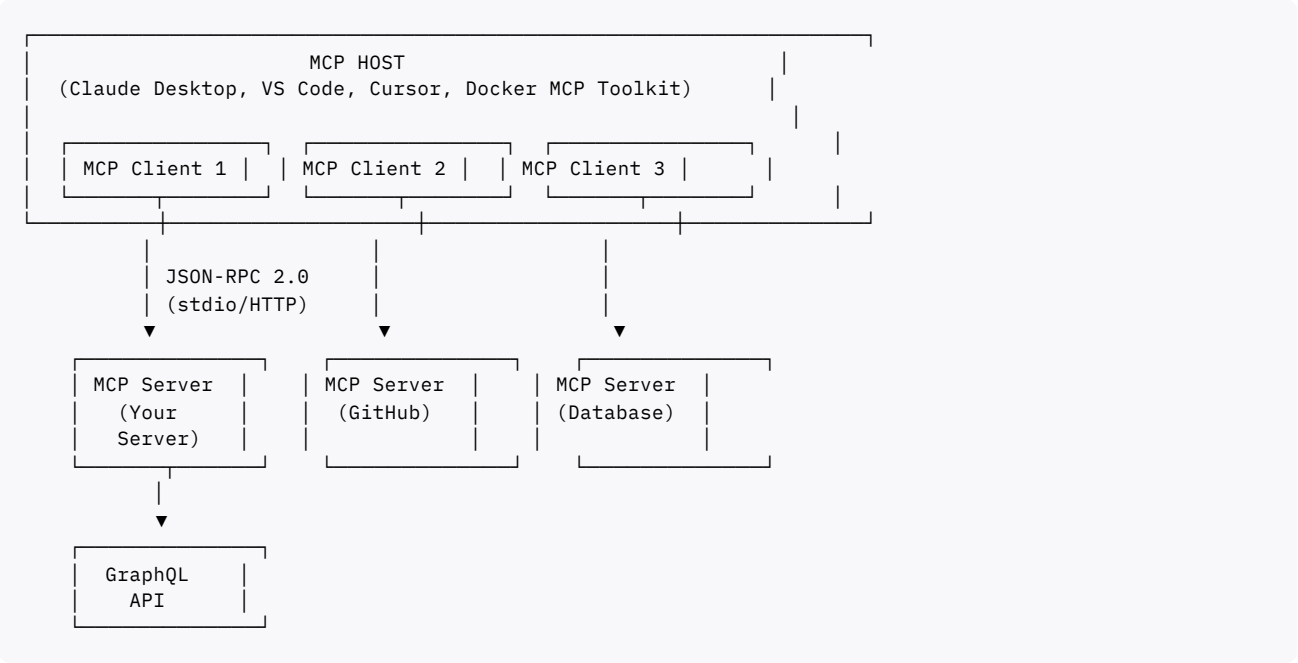
What You'll Build:

A FastMCP server that:

- Exposes 2+ tools calling a GraphQL API
- Handles per-user API keys via environment variables
- Runs bare-metal (uvicorn), locally (Docker), and on Docker Hub
- Follows agent-first design principles

2. MCP Fundamentals: What Makes It Different {#mcp-fundamentals}

2.1 Architecture Overview



2.2 Core Concepts

Hosts: LLM applications (Claude Desktop, VS Code) that initiate connections

Clients: Protocol connectors *within* the host that communicate with servers

Servers: Services (your FastMCP server) that provide:

- **Tools**: Functions the AI model can execute (e.g., `search_patients`, `create_playlist`)
- **Resources**: Static/dynamic data contexts (e.g., database schemas, file contents)
- **Prompts**: Templated workflows for users (e.g., "Draft email to patient")
- **Sampling**: Server-initiated LLM requests (advanced feature)

2.3 Protocol Characteristics

Transport: JSON-RPC 2.0 over stdio (local) or HTTP (remote)

Stateful: Connections persist; capability negotiation happens at initialization

Security Model:

- User consent required for all data access[2]
- MCP servers are **OAuth Resource Servers** (as of June 2025)[3]
- No session-based auth—use bearer tokens or API keys[4]

Key Principle: MCP is designed for **LLM agents**, not humans. This fundamentally changes how you architect your API.

3. Environment Setup {#environment-setup}

3.1 Installation

```
# Python 3.10+ required
python --version  # Should be 3.10 or higher

# Install FastMCP
pip install fastmcp

# Optional: WebSocket support
pip install fastmcp[websockets]

# Optional: For GraphQL integration
pip install httpx  # Modern async HTTP client
# or
pip install requests  # Simpler sync client

# Verify installation
fastmcp version
```

3.2 Project Initialization

```
# Create project directory
mkdir my-mcp-server
cd my-mcp-server

# Create virtual environment (recommended)
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate

# Create project structure
mkdir -p app/{tools,utils}
touch app/__init__.py
touch app/server.py
touch .env.example
touch requirements.txt
touch Dockerfile
touch .gitignore
```

3.3 Requirements File

requirements.txt:

```
fastmcp&gt;=0.5.0
httpx&gt;=0.25.0
pydantic&gt;=2.0.0
python-dotenv&gt;=1.0.0
```

3.4 Git Ignore Essentials

.gitignore:

```
# Python
__pycache__/
*.py[cod]
*$py.class
venv/
.Python

# Environment &amp;amp; Secrets
.env
.env.*
!.env.example

# IDEs
.vscode/
```

```
.idea/
*.swp

# Docker
.dockerignore
```

4. Project Structure & Best Practices {#project-structure}

4.1 Recommended Layout

```
my-mcp-server/
├── app/
│   ├── __init__.py
│   ├── server.py              # FastMCP server definition
│   ├── config.py              # Configuration &amp;amp; env vars
│   ├── tools/
│   │   ├── __init__.py
│   │   ├── patient_tools.py   # Domain-specific tool groups
│   │   └── analytics_tools.py
│   ├── utils/
│   │   ├── __init__.py
│   │   ├── graphql_client.py  # GraphQL integration
│   │   └── validators.py      # Input validation
│   └── schemas/
│       ├── __init__.py
│       └── graphql_schema.graphql  # Optional: cached schema
├── tests/
│   ├── __init__.py
│   ├── test_tools.py
│   └── test_graphql.py
├── .env.example               # Template for required env vars
├── .gitignore
├── Dockerfile
├── docker-compose.yml         # Optional: for local dev
├── requirements.txt
├── pyproject.toml             # Optional: Poetry config
└── README.md
```

4.2 Configuration Management

app/config.py:

```
"""
Configuration management for MCP server.
All secrets are loaded from environment variables.
"""
from pydantic import BaseSettings, Field, validator
from typing import Optional
import os

class Settings(BaseSettings):
    """
    MCP server configuration.

    Environment variables should be set by:
    1. Local dev: .env file
    2. Docker: docker run -e VAR=value
    3. MCP clients: Client configuration (Claude Desktop, etc.)
    """

    # Server metadata
    SERVER_NAME: str = Field(
        default="HealthCare MCP",
        description="Display name for MCP server"
    )
    SERVER_VERSION: str = Field(
```

```
        default="1.0.0",
        description="Semantic version"
    )

    # GraphQL API configuration (REQUIRED)
    GRAPHQL_ENDPOINT: str = Field(
        ...,  # Required field
        description="GraphQL API endpoint URL"
    )

    # Per-user API key (REQUIRED at runtime)
    API_KEY: str = Field(
        ...,  # Required field
        description="User-specific API key for GraphQL API"
    )

    # Optional: Additional auth
    OAUTH_TOKEN: Optional[str] = Field(
        default=None,
        description="OAuth bearer token (if using OAuth)"
    )

    # Operational settings
    LOG_LEVEL: str = Field(
        default="INFO",
        description="Logging level: DEBUG, INFO, WARN, ERROR"
    )
    PORT: int = Field(
        default=8080,
        description="HTTP port for MCP server"
    )
    TIMEOUT: int = Field(
        default=30,
        description="Request timeout in seconds"
    )

    # Security
    MASK_ERROR_DETAILS: bool = Field(
        default=True,
        description="Hide internal error details in production"
    )

    @validator("GRAPHQL_ENDPOINT")
    def validate_endpoint(cls, v):
        """Ensure endpoint is a valid URL."""
        if not v.startswith(("http://", "https://")):
            raise ValueError("GRAPHQL_ENDPOINT must start with http:// or https://")
        return v.rstrip("/")  # Remove trailing slash

    @validator("LOG_LEVEL")
    def validate_log_level(cls, v):
        """Ensure valid log level."""
        allowed = ["DEBUG", "INFO", "WARN", "WARNING", "ERROR", "CRITICAL"]
        if v.upper() not in allowed:
            raise ValueError(f"LOG_LEVEL must be one of {allowed}")
        return v.upper()

    class Config:
        env_file = ".env"
        env_file_encoding = "utf-8"
        case_sensitive = True

# Global settings instance
settings = Settings()
```

.env.example:

```
# MCP Server Configuration Template
# Copy to .env and fill in your values
# NEVER commit .env to version control
```

```
# Required: GraphQL API endpoint
GRAPHQL_ENDPOINT=https://api.example.com/graphql

# Required: Your personal API key (per-user)
# Each user running this MCP will provide their own key
API_KEY=your_api_key_here

# Optional: OAuth token (if using OAuth flow)
# OAUTH_TOKEN=your_oauth_token_here

# Optional: Logging level
LOG_LEVEL=INFO

# Optional: Server port (for HTTP transport)
PORT=8080

# Optional: Request timeout (seconds)
TIMEOUT=30

# Optional: Mask errors in production
MASK_ERROR_DETAILS=true
```

5. Building Your First FastMCP Server {#building-fastmcp-server}

5.1 Minimal Example

app/server.py (minimal version):

```
"""
Minimal FastMCP server example.
Demonstrates basic tool definition and server setup.
"""
from fastmcp import FastMCP

# Create server instance
mcp = FastMCP("Calculator MCP")

@mcp.tool()
def add(a: int, b: int) -&gt; int:
    """Add two numbers together.

    Args:
        a: First number
        b: Second number

    Returns:
        Sum of a and b
    """
    return a + b

@mcp.tool()
def multiply(a: int, b: int) -&gt; int:
    """Multiply two numbers.

    Args:
        a: First number
        b: Second number

    Returns:
        Product of a and b
    """
    return a * b

if __name__ == "__main__":
    # Run server (defaults to stdio transport)
    mcp.run()
```

Running it:

```
# Method 1: Direct Python execution
python app/server.py

# Method 2: FastMCP CLI (recommended)
fastmcp run app/server.py

# Method 3: Development mode with auto-reload
fastmcp dev app/server.py --log-level DEBUG
```

5.2 Production-Ready Structure

app/server.py (production version):

```
"""
Production MCP server with GraphQL integration.
"""
import logging
from fastmcp import FastMCP, Context
from typing import List, Dict, Any, Optional
import sys

from app.config import settings
from app.utils.graphql_client import GraphQLClient
from app.tools.patient_tools import register_patient_tools
from app.tools.analytics_tools import register_analytics_tools

# Configure logging
logging.basicConfig(
    level=getattr(logging, settings.LOG_LEVEL),
    format="%(asctime)s - %(name)s - %(levelname)s - %(message)s",
    handlers=[
        logging.StreamHandler(sys.stdout)
    ]
)
logger = logging.getLogger(__name__)

# Create MCP server instance
mcp = FastMCP(
    name=settings.SERVER_NAME,
    logger=logger,
    mask_error_details=settings.MASK_ERROR_DETAILS
)

# Initialize GraphQL client (singleton pattern)
graphql_client = GraphQLClient(
    endpoint=settings.GRAPHQL_ENDPOINT,
    api_key=settings.API_KEY,
    timeout=settings.TIMEOUT
)

# Register tool groups
register_patient_tools(mcp, graphql_client)
register_analytics_tools(mcp, graphql_client)

# Health check resource
@mcp.resource("health://status")
def health_check() -&gt; Dict[str, Any]:
    """Server health status.

    Returns:
        Health status information
    """
    return {
        "status": "healthy",
        "version": settings.SERVER_VERSION,
        "graphql_endpoint": settings.GRAPHQL_ENDPOINT,
        "timestamp": __import__("datetime").datetime.utcnow().isoformat()
    }
```

```
# Server metadata resource
@mcp.resource("config://server-info")
def server_info() -&gt; Dict[str, str]:
    """Server metadata and capabilities.

    Returns:
        Server information
    """
    return {
        "name": settings.SERVER_NAME,
        "version": settings.SERVER_VERSION,
        "description": "MCP server for healthcare data access via GraphQL"
    }

if __name__ == "__main__":
    logger.info(f"Starting {settings.SERVER_NAME} v{settings.SERVER_VERSION}")
    logger.info(f"GraphQL endpoint: {settings.GRAPHQL_ENDPOINT}")
    logger.info(f"Log level: {settings.LOG_LEVEL}")

    # Validate required configuration
    if not settings.API_KEY:
        logger.error("API_KEY environment variable is required")
        sys.exit(1)

    # Run server
    mcp.run(port=settings.PORT)
```

5.3 Type Annotations & Schema Generation

FastMCP **automatically generates JSON schemas** from your type hints and docstrings[5]. This is critical for LLM agents to understand your tools.

Best practices:

```
from typing import List, Dict, Optional, Literal
from enum import Enum
from pydantic import BaseModel, Field

# ✅ GOOD: Use specific types
@mcp.tool()
def search_patients(
    name: str,
    status: Literal["active", "inactive", "all"] = "active",
    limit: int = 10
) -&gt; List[Dict[str, Any]]:
    """Search for patients by name.

    Args:
        name: Patient name (partial match supported)
        status: Filter by patient status
        limit: Maximum number of results (1-100)

    Returns:
        List of patient records
    """
    # Implementation
    pass

# ✅ BETTER: Use Pydantic models for complex types
class PatientStatus(str, Enum):
    ACTIVE = "active"
    INACTIVE = "inactive"
    ALL = "all"

class PatientSearchResult(BaseModel):
    id: str = Field(..., description="Patient unique identifier")
    name: str = Field(..., description="Full name")
    status: PatientStatus = Field(..., description="Current status")
    last_visit: Optional[str] = Field(None, description="Last visit date (ISO 8601)")
```

```
@mcp.tool()
def search_patients_typed(
    name: str = Field(..., description="Patient name (partial match)"),
    status: PatientStatus = PatientStatus.ACTIVE,
    limit: int = Field(10, ge=1, le=100, description="Max results")
) -&gt; List[PatientSearchResult]:
    """Search for patients by name (type-safe version)."""
    # Implementation
    pass

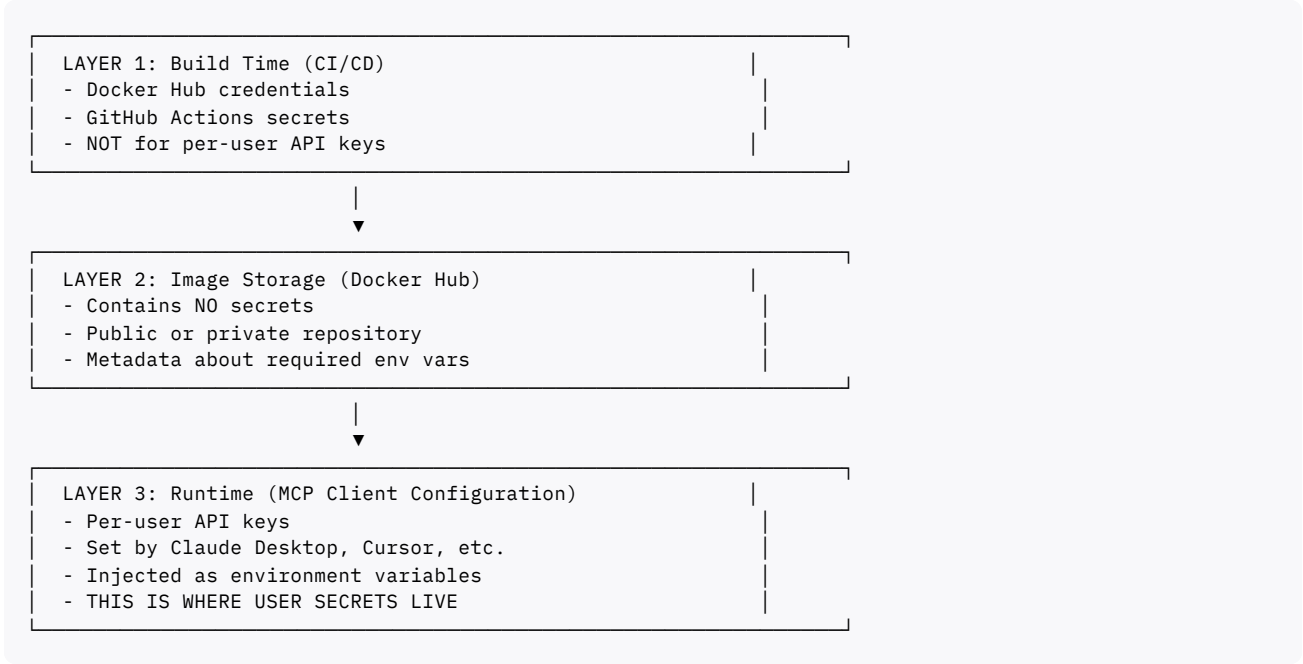
# ✗ AVOID: Generic types without constraints
@mcp.tool()
def bad_search(
    data: dict,  # Too generic
    options: Any  # No type information
) -&gt; list:  # Untyped elements
    """This provides no useful schema to the LLM."""
    pass
```

Key takeaway: Strong typing = Better LLM understanding = More reliable tool calls.

6. Secrets & API Key Management {#secrets-management}

6.1 The Three-Layer Secret Model

Understanding the architecture:



6.2 Implementation Patterns

Pattern 1: Environment Variable Loading (Recommended)

```
# app/utils/secrets.py
"""
Secure secret management for MCP servers.
"""
import os
import logging
from typing import Optional

logger = logging.getLogger(__name__)

class SecretManager:
```

```
    """
    Manages runtime secrets loaded from environment variables.

    Design principles:
    1. Fail fast if required secrets are missing
    2. Never log secret values
    3. Validate secrets at startup
    4. Support multiple secret sources (env vars, files, vaults)
    """

    def __init__(self):
        self._secrets = {}
        self._load_secrets()

    def _load_secrets(self):
        """Load secrets from environment variables."""
        required = ["API_KEY", "GRAPHQL_ENDPOINT"]
        optional = ["OAUTH_TOKEN", "REFRESH_TOKEN"]

        for key in required:
            value = os.getenv(key)
            if not value:
                raise ValueError(
                    f"Required environment variable {key} is not set. "
                    f"Set it in your MCP client configuration."
                )
            self._secrets[key] = value
            logger.info(f"Loaded required secret: {key}")

        for key in optional:
            value = os.getenv(key)
            if value:
                self._secrets[key] = value
                logger.info(f"Loaded optional secret: {key}")

    def get(self, key: str, default: Optional[str] = None) -&gt; Optional[str]:
        """
        Retrieve a secret value.

        Args:
            key: Secret name
            default: Default value if secret not found

        Returns:
            Secret value or default
        """
        return self._secrets.get(key, default)

    def mask(self, secret: str) -&gt; str:
        """
        Mask a secret for safe logging.

        Args:
            secret: Secret to mask

        Returns:
            Masked string (e.g., "abc***xyz")
        """
        if not secret or len(secret) &lt; 8:
            return "***"
        return f"{secret[:3]}{'*' * (len(secret) - 6)}{secret[-3:]}"

    def validate_api_key(self, api_key: str) -&gt; bool:
        """
        Validate API key format (example).

        Args:
            api_key: API key to validate

        Returns:
            True if valid format
        """
```

```
        # Example: Check prefix and length
        if not api_key.startswith("sk_"):
            logger.warning("API key does not match expected format")
            return False
        if len(api_key) &lt; 32:
            logger.warning("API key appears too short")
            return False
        return True

# Global instance
secret_manager = SecretManager()
```

Usage in server:

```
from app.utils.secrets import secret_manager

# In GraphQL client initialization
api_key = secret_manager.get("API_KEY")
logger.info(f"Using API key: {secret_manager.mask(api_key)}")

graphql_client = GraphQLClient(
    endpoint=secret_manager.get("GRAPHQL_ENDPOINT"),
    api_key=api_key
)
```

6.3 Security Best Practices

DO:

- ✅ Load secrets from environment variables at runtime
- ✅ Validate secrets at server startup (fail fast)
- ✅ Use secret masking in logs (`api_key=abc***xyz`)
- ✅ Document required env vars in README and `.env.example`
- ✅ Use different secrets for dev/staging/production
- ✅ Rotate secrets regularly
- ✅ Apply least privilege (minimal scopes)

DON'T:

- ❌ Hard-code secrets in source code
- ❌ Commit `.env` files to version control
- ❌ Log full secret values (even in debug mode)
- ❌ Bake secrets into Docker images
- ❌ Share secrets across multiple users
- ❌ Use the same secret for dev and production
- ❌ Store secrets in client-side code

6.4 Client Configuration Examples

Claude Desktop (stdio transport):

~/.config/claude/claude_desktop_config.json:

```
{
  "mcpServers": {
    "healthcare-mcp": {
      "command": "docker",
      "args": [
        "run",
        "-i",
        "--rm",
```

```
        "-e", "API_KEY=sk_live_abc123xyz789",
        "-e", "GRAPHQL_ENDPOINT=https://api.example.com/graphql",
        "-e", "LOG_LEVEL=INFO",
        "your-dockerhub-user/healthcare-mcp:latest"
      ]
    }
  }
}
```

VS Code / Cursor (HTTP transport):

.mcp.json:

```
{
  "servers": {
    "healthcare-mcp": {
      "transport": "http",
      "url": "http://localhost:8080/mcp",
      "env": {
        "API_KEY": "${API_KEY}",
        "GRAPHQL_ENDPOINT": "${GRAPHQL_ENDPOINT}"
      }
    }
  },
  "inputs": [
    {
      "id": "API_KEY",
      "type": "secret",
      "description": "Your API key for the healthcare GraphQL API"
    },
    {
      "id": "GRAPHQL_ENDPOINT",
      "type": "string",
      "description": "GraphQL API endpoint URL",
      "default": "https://api.example.com/graphql"
    }
  ]
}
```

Docker Hub MCP Toolkit:

When published to Docker Hub with MCP integration:

1. Users enable your server in Docker Desktop MCP Toolkit
2. Toolkit prompts for required environment variables
3. Secrets stored securely in Docker Desktop
4. Injected at container runtime

Documentation in your README.md**:**

```
## Configuration

This MCP server requires the following environment variables:

### Required

- `API_KEY` (string): Your personal API key for the GraphQL API
  - Obtain from: https://api.example.com/settings/api-keys
  - Format: `sk_live_...` or `sk_test_...`
  - Scope required: `read:patients`, `write:appointments`

- `GRAPHQL_ENDPOINT` (string): GraphQL API endpoint URL
  - Production: `https://api.example.com/graphql`
  - Staging: `https://staging-api.example.com/graphql`

### Optional

- `LOG_LEVEL` (string): Logging verbosity
```

```
  - Options: `DEBUG`, `INFO`, `WARN`, `ERROR`
  - Default: `INFO`

- `TIMEOUT` (integer): Request timeout in seconds
  - Default: `30`
  - Range: `5` to `300`

### Example: Running with Docker

```bash
docker run -i --rm \
  -e API_KEY=sk_live_your_key_here \
  -e GRAPHQL_ENDPOINT=https://api.example.com/graphql \
  your-dockerhub-user/healthcare-mcp:latest
```

---

7. GraphQL Integration Patterns {#graphql-integration}

7.1 Architecture Decision: Wrapper vs Pass-Through

Option A: Generic GraphQL Pass-Through

```python
@mcp.tool()
def execute_graphql(
    query: str,
    variables: Optional[Dict[str, Any]] = None
) -> Dict[str, Any]:
    """
    Execute arbitrary GraphQL query.

    ⚠ Security consideration: Allows arbitrary queries.
    """
    return graphql_client.execute(query, variables)
```

Pros:

- Maximum flexibility for LLM
- Single tool covers all use cases
- Easy to implement

Cons:

- Security risk (users can craft any query)
- No input validation
- Hard for LLM to construct correct queries
- Poor error messages
- No query optimization

Verdict: ✕ Avoid for hackathons and production

Option B: Specific, High-Level Tools (RECOMMENDED)

```
@mcp.tool()
def search_patients(
    name: str,
    status: Literal["active", "inactive"] = "active",
    limit: int = 10
) -> List[Dict[str, Any]]:
    """
    Search patients by name.

    Internally executes GraphQL query:
      query SearchPatients($name: String!, $status: String!, $limit: Int!) {
```
```

```
        patients(name: $name, status: $status, limit: $limit) {
          id
          name
          status
          lastVisit
        }
      }
    """
    query = """
        query SearchPatients($name: String!, $status: String!, $limit: Int!) {
            patients(name: $name, status: $status, limit: $limit) {
                id
                name
                status
                lastVisit
                contactInfo {
                    email
                    phone
                }
            }
        }
    """
    variables = {"name": name, "status": status, "limit": limit}
    result = graphql_client.execute(query, variables)

    # Normalize response for LLM
    if "errors" in result:
        raise ToolError(f"GraphQL error: {result['errors']}")

    return result["data"]["patients"]

@mcp.tool()
def get_patient_appointments(
    patient_id: str,
    from_date: Optional[str] = None,
    to_date: Optional[str] = None
) -&gt; List[Dict[str, Any]]:
    """
    Get appointments for a specific patient.

    Args:
        patient_id: Patient UUID
        from_date: Start date (ISO 8601, optional)
        to_date: End date (ISO 8601, optional)
    """
    query = """
        query GetAppointments($patientId: ID!, $from: Date, $to: Date) {
            patient(id: $patientId) {
                appointments(from: $from, to: $to) {
                    id
                    dateTime
                    provider {
                        name
                        specialty
                    }
                    status
                    notes
                }
            }
        }
    """
    variables = {
        "patientId": patient_id,
        "from": from_date,
        "to": to_date
    }
    result = graphql_client.execute(query, variables)

    if "errors" in result:
        raise ToolError(f"Failed to fetch appointments: {result['errors']}")
```

```
    return result["data"]["patient"]["appointments"]
```

Pros:

- ✅ Strong input validation
- ✅ Security: controlled queries only
- ✅ Clear tool descriptions for LLM
- ✅ Optimized queries (only necessary fields)
- ✅ Clean error handling
- ✅ Easy to test

Cons:

- More code per operation
- Requires planning which operations to expose

Verdict: ✅ **Recommended for hackathons and production**

7.2 GraphQL Client Implementation

app/utils/graphql_client.py:

```
"""
GraphQL client for MCP server.
Handles authentication, retries, and error normalization.
"""
import httpx
import logging
from typing import Dict, Any, Optional
from tenacity import retry, stop_after_attempt, wait_exponential

logger = logging.getLogger(__name__)

class GraphQLError(Exception):
    """GraphQL-specific error."""
    pass

class GraphQLClient:
    """
    Async GraphQL client with authentication and retry logic.

    Design principles:
    1. Single responsibility: GraphQL communication only
    2. Transparent error handling
    3. Automatic retries for transient failures
    4. Request logging (without exposing secrets)
    """

    def __init__(
        self,
        endpoint: str,
        api_key: str,
        timeout: int = 30,
        max_retries: int = 3
    ):
        """
        Initialize GraphQL client.

        Args:
            endpoint: GraphQL API endpoint URL
            api_key: API key for authentication
            timeout: Request timeout in seconds
            max_retries: Maximum retry attempts
        """
        self.endpoint = endpoint
```

```
        self.api_key = api_key
        self.timeout = timeout
        self.max_retries = max_retries

        # Create persistent HTTP client
        self.client = httpx.AsyncClient(
            timeout=httpx.Timeout(timeout),
            headers=self._build_headers(),
            follow_redirects=True
        )

        logger.info(f"GraphQL client initialized: {endpoint}")

    def _build_headers(self) -&gt; Dict[str, str]:
        """Build request headers with authentication."""
        return {
            "Content-Type": "application/json",
            "Authorization": f"Bearer {self.api_key}",
            "User-Agent": "MCP-Server/1.0"
        }

    @retry(
        stop=stop_after_attempt(3),
        wait=wait_exponential(multiplier=1, min=1, max=10),
        reraise=True
    )
    async def execute(
        self,
        query: str,
        variables: Optional[Dict[str, Any]] = None,
        operation_name: Optional[str] = None
    ) -&gt; Dict[str, Any]:
        """
        Execute GraphQL query/mutation.

        Args:
            query: GraphQL query string
            variables: Query variables
            operation_name: Operation name (for logging)

        Returns:
            GraphQL response data

        Raises:
            GraphQLError: If query fails
        """
        payload = {
            "query": query,
            "variables": variables or {}
        }
        if operation_name:
            payload["operationName"] = operation_name

        logger.debug(f"Executing GraphQL: {operation_name or 'unnamed'}")

        try:
            response = await self.client.post(
                self.endpoint,
                json=payload
            )
            response.raise_for_status()

            data = response.json()

            # GraphQL can return 200 with errors
            if "errors" in data:
                error_messages = [e.get("message", "Unknown error") for e in data["errors"]]
                logger.error(f"GraphQL errors: {error_messages}")
                raise GraphQLError(f"GraphQL errors: {', '.join(error_messages)}")

            logger.debug(f"GraphQL success: {operation_name or 'unnamed'}")
            return data
```

```
        except httpx.HTTPStatusError as e:
            logger.error(f"HTTP error: {e.response.status_code} - {e.response.text}")
            raise GraphQLError(f"HTTP {e.response.status_code}: {e.response.text}")

        except httpx.RequestError as e:
            logger.error(f"Request failed: {e}")
            raise GraphQLError(f"Network error: {e}")

    async def close(self):
        """Close HTTP client (cleanup)."""
        await self.client.aclose()

    async def introspect_schema(self) -&gt; Dict[str, Any]:
        """
        Fetch GraphQL schema via introspection.
        Useful for validation and documentation.
        """
        introspection_query = """
            query IntrospectionQuery {
                __schema {
                    queryType { name }
                    mutationType { name }
                    types {
                        name
                        kind
                        description
                        fields {
                            name
                            description
                            type {
                                name
                                kind
                            }
                        }
                    }
                }
            }
        """
        result = await self.execute(
            introspection_query,
            operation_name="IntrospectionQuery"
        )
        return result["data"]["__schema"]
```

Synchronous alternative (for simple cases):

```
import requests

class SyncGraphQLClient:
    """Synchronous GraphQL client (simpler, but blocks)."""

    def __init__(self, endpoint: str, api_key: str, timeout: int = 30):
        self.endpoint = endpoint
        self.timeout = timeout
        self.headers = {
            "Content-Type": "application/json",
            "Authorization": f"Bearer {api_key}"
        }

    def execute(
        self,
        query: str,
        variables: Optional[Dict[str, Any]] = None
    ) -&gt; Dict[str, Any]:
        """Execute GraphQL query (synchronous)."""
        payload = {
            "query": query,
            "variables": variables or {}
        }
```

```
        response = requests.post(
            self.endpoint,
            json=payload,
            headers=self.headers,
            timeout=self.timeout
        )
        response.raise_for_status()

        data = response.json()
        if "errors" in data:
            raise GraphQLError(f"GraphQL errors: {data['errors']}")

        return data
```

7.3 Tool Implementation Pattern

app/tools/patient_tools.py:

```
"""
Patient-related MCP tools backed by GraphQL.
"""
from fastmcp import FastMCP, Context, ToolError
from typing import List, Dict, Any, Optional, Literal
from app.utils.graphql_client import GraphQLClient, GraphQLError
import logging

logger = logging.getLogger(__name__)

def register_patient_tools(mcp: FastMCP, graphql_client: GraphQLClient):
    """
    Register patient-related tools with the MCP server.

    Args:
        mcp: FastMCP server instance
        graphql_client: GraphQL client instance
    """

    @mcp.tool()
    async def search_patients(
        ctx: Context,
        name: str,
        status: Literal["active", "inactive", "all"] = "active",
        limit: int = 10
    ) -&gt; List[Dict[str, Any]]:
        """
        Search for patients by name.

        Returns a list of patients matching the search criteria,
        including basic contact information.

        Args:
            name: Patient name (partial matches supported)
            status: Filter by patient status
            limit: Maximum number of results (1-100)

        Returns:
            List of patient records with id, name, status, and contact info
        """
        # Input validation
        if not name or len(name) &lt; 2:
            raise ToolError("Name must be at least 2 characters")
        if not 1 &lt;= limit &lt;= 100:
            raise ToolError("Limit must be between 1 and 100")

        ctx.info(f"Searching patients: name='{name}', status={status}, limit={limit}")

        # GraphQL query
        query = """
            query SearchPatients($name: String!, $status: String!, $limit: Int!) {
                searchPatients(name: $name, status: $status, limit: $limit) {
```

```
                    id
                    name
                    status
                    dateOfBirth
                    contactInfo {
                        email
                        phone
                    }
                    lastVisit
                }
            }
        """

        variables = {
            "name": name,
            "status": status,
            "limit": limit
        }

        try:
            result = await graphql_client.execute(
                query,
                variables,
                operation_name="SearchPatients"
            )

            patients = result["data"]["searchPatients"]
            ctx.info(f"Found {len(patients)} patients")

            return patients

        except GraphQLError as e:
            ctx.error(f"GraphQL error: {e}")
            raise ToolError(f"Failed to search patients: {e}")

    @mcp.tool()
    async def get_patient_details(
        ctx: Context,
        patient_id: str
    ) -&gt; Dict[str, Any]:
        """
        Get detailed information for a specific patient.

        Args:
            patient_id: Patient UUID

        Returns:
            Complete patient record including medical history
        """
        ctx.info(f"Fetching patient details: {patient_id}")

        query = """
            query GetPatient($id: ID!) {
                patient(id: $id) {
                    id
                    name
                    dateOfBirth
                    status
                    contactInfo {
                        email
                        phone
                        address {
                            street
                            city
                            state
                            zipCode
                        }
                    }
                    medicalHistory {
                        conditions
                        allergies
                        medications
```

```
                    }
                    lastVisit
                    primaryProvider {
                        id
                        name
                        specialty
                    }
                }
            }
        """

        try:
            result = await graphql_client.execute(
                query,
                {"id": patient_id},
                operation_name="GetPatient"
            )

            patient = result["data"]["patient"]
            if not patient:
                raise ToolError(f"Patient not found: {patient_id}")

            ctx.info(f"Retrieved patient: {patient['name']}")
            return patient

        except GraphQLError as e:
            ctx.error(f"GraphQL error: {e}")
            raise ToolError(f"Failed to fetch patient: {e}")

    logger.info("Patient tools registered")
```

7.4 Hackathon-Friendly Shortcuts

For a 6-hour hackathon:

1. **Start with 2-3 core tools**
   - Don't try to expose every GraphQL operation
   - Focus on the most common user actions
2. **Hard-code queries initially**
   - You can extract to files later
   - Keep queries co-located with tools for speed
3. **Minimal error handling**
   - Catch GraphQLError and re-raise as ToolError
   - Add detailed errors if time permits
4. **Skip query optimization**
   - Request all fields you might need
   - Optimize later based on usage
5. **Use synchronous client**
   - Simpler code (no async/await)
   - Good enough for hackathon scale
   - Upgrade to async for production

Example hackathon-speed tool:

```
@mcp.tool()
def quick_search(name: str) -&gt; list:
    """Search patients by name (quick version)."""
    query = "{ searchPatients(name: $name) { id name email } }"
    result = graphql_client.execute(query, {"name": name})
    return result["data"]["searchPatients"]
```

8. Dockerization Strategy {#dockerization}

8.1 Dockerfile Best Practices

Dockerfile (production-ready):

```
# syntax=docker/dockerfile:1
FROM python:3.11-slim as base

# Metadata
LABEL org.opencontainers.image.title="Healthcare MCP Server"
LABEL org.opencontainers.image.description="MCP server for healthcare data via GraphQL"
LABEL org.opencontainers.image.version="1.0.0"
LABEL org.opencontainers.image.authors="Your Name &lt;you@example.com&gt;"
LABEL org.opencontainers.image.source="https://github.com/yourusername/healthcare-mcp"

# Set environment variables
ENV PYTHONUNBUFFERED=1 \
    PYTHONDONTWRITEBYTECODE=1 \
    PIP_NO_CACHE_DIR=1 \
    PIP_DISABLE_PIP_VERSION_CHECK=1

# Create non-root user for security
RUN groupadd -r mcpuser &amp;&amp; \
    useradd -r -g mcpuser -u 1000 mcpuser

# Set working directory
WORKDIR /app

# Install system dependencies (if needed)
RUN apt-get update &amp;&amp; \
    apt-get install -y --no-install-recommends \
        curl \
    &amp;&amp; rm -rf /var/lib/apt/lists/*

# Copy dependency files
COPY --chown=mcpuser:mcpuser requirements.txt .

# Install Python dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY --chown=mcpuser:mcpuser app/ ./app/

# Switch to non-root user
USER mcpuser

# Health check (optional but recommended)
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
    CMD curl -f http://localhost:8080/health || exit 1

# Expose port (for HTTP transport)
EXPOSE 8080

# Default command
# Note: Secrets are injected via environment variables at runtime
CMD ["python", "-m", "app.server"]
```

Multi-stage build (for smaller images):

```
# syntax=docker/dockerfile:1

# Stage 1: Builder
FROM python:3.11-slim as builder

ENV PIP_NO_CACHE_DIR=1

WORKDIR /build
```

```
# Install build dependencies
RUN apt-get update &amp;amp;&amp;amp; \
    apt-get install -y --no-install-recommends \
        gcc \
        python3-dev \
    &amp;amp;&amp;amp; rm -rf /var/lib/apt/lists/*

# Copy and install dependencies
COPY requirements.txt .
RUN pip install --user --no-cache-dir -r requirements.txt

# Stage 2: Runtime
FROM python:3.11-slim as runtime

LABEL org.opencontainers.image.title="Healthcare MCP Server"
LABEL org.opencontainers.image.version="1.0.0"

ENV PYTHONUNBUFFERED=1 \
    PYTHONDONTWRITEBYTECODE=1 \
    PATH="/home/mcpuser/.local/bin:$PATH"

# Create non-root user
RUN groupadd -r mcpuser &amp;amp;&amp;amp; \
    useradd -r -g mcpuser -u 1000 -m mcpuser

WORKDIR /app

# Copy installed packages from builder
COPY --from=builder --chown=mcpuser:mcpuser /root/.local /home/mcpuser/.local

# Copy application
COPY --chown=mcpuser:mcpuser app/ ./app/

USER mcpuser

EXPOSE 8080

CMD ["python", "-m", "app.server"]
```

8.2 Building & Running Locally

Build:

```
# Build image
docker build -t healthcare-mcp:latest .

# Build with custom tag
docker build -t healthcare-mcp:1.0.0 .

# Build with build args (if needed)
docker build \
  --build-arg PYTHON_VERSION=3.11 \
  -t healthcare-mcp:latest .
```

Run (stdio transport for MCP):

```
# Run with environment variables
docker run -i --rm \
  -e API_KEY=sk_live_your_key_here \
  -e GRAPHQL_ENDPOINT=https://api.example.com/graphql \
  -e LOG_LEVEL=INFO \
  healthcare-mcp:latest

# Run with env file
docker run -i --rm \
  --env-file .env \
  healthcare-mcp:latest
```

```
# Run with volume mount (for development)
docker run -i --rm \
  -v $(pwd)/app:/app/app:ro \
  -e API_KEY=sk_test_dev_key \
  -e GRAPHQL_ENDPOINT=http://localhost:4000/graphql \
  healthcare-mcp:latest
```

Run (HTTP transport for testing):

```
# Expose HTTP port
docker run -d --rm \
  -p 8080:8080 \
  -e API_KEY=sk_test_key \
  -e GRAPHQL_ENDPOINT=https://api.example.com/graphql \
  --name mcp-server \
  healthcare-mcp:latest

# Test health endpoint
curl http://localhost:8080/health

# View logs
docker logs -f mcp-server

# Stop
docker stop mcp-server
```

8.3 Docker Compose (Development)

docker-compose.yml:

```
version: '3.8'

services:
  mcp-server:
    build:
      context: .
      dockerfile: Dockerfile
    image: healthcare-mcp:dev
    container_name: healthcare-mcp-dev
    ports:
      - "8080:8080"
    environment:
      # Load from .env file
      API_KEY: ${API_KEY}
      GRAPHQL_ENDPOINT: ${GRAPHQL_ENDPOINT}
      LOG_LEVEL: DEBUG
      MASK_ERROR_DETAILS: false
    volumes:
      # Hot reload during development
      - ./app:/app/app:ro
    restart: unless-stopped
    networks:
      - mcp-network

  # Optional: Mock GraphQL API for testing
  mock-graphql:
    image: graphql-mock-server:latest
    container_name: mock-graphql-api
    ports:
      - "4000:4000"
    networks:
      - mcp-network

networks:
  mcp-network:
    driver: bridge
```

Usage:

```
# Start all services
docker-compose up -d

# View logs
docker-compose logs -f mcp-server

# Rebuild after code changes
docker-compose up -d --build

# Stop all services
docker-compose down

# Clean up volumes
docker-compose down -v
```

8.4 Image Size Optimization

Techniques:

1. **Use slim base images**
   - `python:3.11-slim` instead of `python:3.11`
   - Saves ~800MB
2. **Multi-stage builds**
   - Separate build and runtime stages
   - Only ship runtime dependencies
3. **Clean up in same layer**

   ```
   RUN apt-get update &amp;&amp; \
       apt-get install -y --no-install-recommends gcc &amp;&amp; \
       pip install requirements &amp;&amp; \
       apt-get purge -y gcc &amp;&amp; \
       rm -rf /var/lib/apt/lists/*
   ```

4. **Use .dockerignore**

   ```
   .git
   .env
   .env.*
   __pycache__
   *.pyc
   tests/
   docs/
   *.md
   .vscode/
   .idea/
   node_modules/
   ```

5. **Minimize layers**
   - Combine RUN commands
   - Copy only necessary files

Result:

- Full Python image: ~1.2GB
- Optimized: ~200-300MB

9. Docker Hub Publishing {#docker-hub-publishing}

9.1 Preparation

1. Create Docker Hub account

- Sign up at https://hub.docker.com
- Create access token: Account Settings → Security → New Access Token

2. Create repository

- Name: `your-username/healthcare-mcp`
- Visibility: Public or Private
- Description: "MCP server for healthcare data via GraphQL"

3. Add MCP metadata (optional but recommended)

- Add `mcp` tag to repository
- Add README with configuration instructions
- Include example configuration snippets

9.2 Tagging Strategy

```
# Login to Docker Hub
docker login -u your-username

# Tag conventions
docker tag healthcare-mcp:latest your-username/healthcare-mcp:latest
docker tag healthcare-mcp:latest your-username/healthcare-mcp:1.0.0
docker tag healthcare-mcp:latest your-username/healthcare-mcp:1.0
docker tag healthcare-mcp:latest your-username/healthcare-mcp:1

# Push all tags
docker push your-username/healthcare-mcp:latest
docker push your-username/healthcare-mcp:1.0.0
docker push your-username/healthcare-mcp:1.0
docker push your-username/healthcare-mcp:1

# Or push all at once
docker push --all-tags your-username/healthcare-mcp
```

Semantic versioning:

- `latest` - Latest stable release
- `1.0.0` - Specific version (immutable)
- `1.0` - Minor version (receives patches)
- `1` - Major version (receives minor updates)
- `dev` - Development/unstable builds

9.3 README Template for Docker Hub

Docker Hub README.md**:**

```
# Healthcare MCP Server

Model Context Protocol (MCP) server providing AI assistants with access to healthcare data via GraphQL.

## Features

- 🏥 Patient search and lookup
- 📅 Appointment management
- 📊 Analytics and reporting
- 🔒 Secure API key authentication
```

- 🐳 One-click Docker deployment

Quick Start

Using Docker Hub MCP Toolkit

1. Open Docker Desktop
2. Navigate to MCP Toolkit
3. Search for "healthcare-mcp"
4. Click "Enable"
5. Provide your API key when prompted

Manual Docker Run

```bash
docker run -i --rm \
  -e API_KEY=your_api_key_here \
  -e GRAPHQL_ENDPOINT=https://api.example.com/graphql \
  your-username/healthcare-mcp:latest
```

Claude Desktop Integration

Add to `~/.config/claude/claude_desktop_config.json`:

```json
{
  "mcpServers": {
    "healthcare": {
      "command": "docker",
      "args": [
        "run", "-i", "--rm",
        "-e", "API_KEY=your_api_key",
        "-e", "GRAPHQL_ENDPOINT=https://api.example.com/graphql",
        "your-username/healthcare-mcp:latest"
      ]
    }
  }
}
```

Configuration

Required Environment Variables

Variable	Description	Example
`API_KEY`	Your API key from healthcare platform	`sk_live_abc123...`
`GRAPHQL_ENDPOINT`	GraphQL API endpoint URL	`https://api.example.com/graphql`

Optional Environment Variables

Variable	Description	Default
`LOG_LEVEL`	Logging verbosity (DEBUG/INFO/WARN/ERROR)	`INFO`
`TIMEOUT`	Request timeout in seconds	`30`

Obtaining API Keys

1. Visit https://api.example.com/settings/api-keys
2. Click "Create New Key"
3. Select scopes: `read:patients`, `write:appointments`
4. Copy the generated key (starts with `sk_live_` or `sk_test_`)

Available Tools

`search_patients`
Search for patients by name with status filtering.

Parameters:
- `name` (string): Patient name (partial matches supported)
- `status` (string): Filter by "active" or "inactive" (default: "active")

```
- `limit` (integer): Max results, 1-100 (default: 10)

### `get_patient_details`
Retrieve complete patient record including medical history.

**Parameters:**
- `patient_id` (string): Patient UUID

### `get_patient_appointments`
List appointments for a specific patient.

**Parameters:**
- `patient_id` (string): Patient UUID
- `from_date` (string, optional): Start date (ISO 8601)
- `to_date` (string, optional): End date (ISO 8601)

## Security

- ✅ API keys are **never** stored in the image
- ✅ Keys are provided at runtime via environment variables
- ✅ All traffic uses HTTPS
- ✅ Runs as non-root user inside container
- ✅ No data persisted in container

## Troubleshooting

### "API_KEY environment variable is not set"
Make sure you're passing `-e API_KEY=...` when running the container.

### "Failed to connect to GraphQL endpoint"
Check that `GRAPHQL_ENDPOINT` is correct and accessible from your network.

### "GraphQL errors: Unauthorized"
Verify your API key is valid and has the required scopes.

## Support

- 📖 Documentation: https://github.com/yourname/healthcare-mcp
- 🐛 Issues: https://github.com/yourname/healthcare-mcp/issues
- 💬 Discussions: https://github.com/yourname/healthcare-mcp/discussions

## License

MIT License - see LICENSE file for details.
```

9.4 Automated Builds with GitHub Actions

.github/workflows/docker-publish.yml:

```
name: Build and Push Docker Image

on:
  push:
    branches: [ main ]
    tags: [ 'v*' ]
  pull_request:
    branches: [ main ]

env:
  REGISTRY: docker.io
  IMAGE_NAME: your-username/healthcare-mcp

jobs:
  build-and-push:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write

    steps:
```

```
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3

      - name: Log in to Docker Hub
        if: github.event_name != 'pull_request'
        uses: docker/login-action@v3
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}

      - name: Extract metadata
        id: meta
        uses: docker/metadata-action@v5
        with:
          images: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}
          tags: |
            type=ref,event=branch
            type=ref,event=pr
            type=semver,pattern={{version}}
            type=semver,pattern={{major}}.{{minor}}
            type=semver,pattern={{major}}
            type=sha,prefix={{branch}}-

      - name: Build and push Docker image
        uses: docker/build-push-action@v5
        with:
          context: .
          push: ${{ github.event_name != 'pull_request' }}
          tags: ${{ steps.meta.outputs.tags }}
          labels: ${{ steps.meta.outputs.labels }}
          cache-from: type=gha
          cache-to: type=gha,mode=max
          platforms: linux/amd64,linux/arm64

      - name: Update Docker Hub description
        if: github.event_name != 'pull_request'
        uses: peter-evans/dockerhub-description@v4
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}
          repository: ${{ env.IMAGE_NAME }}
          short-description: ${{ github.event.repository.description }}
          readme-filepath: ./README.md
```

Setup:

1. Add secrets to GitHub repository:
   - `DOCKERHUB_USERNAME`: Your Docker Hub username
   - `DOCKERHUB_TOKEN`: Access token from Docker Hub
2. Create release:

   ```
   git tag -a v1.0.0 -m "Release version 1.0.0"
   git push origin v1.0.0
   ```

3. GitHub Actions automatically:
   - Builds multi-platform image (amd64, arm64)
   - Pushes to Docker Hub with semantic version tags
   - Updates Docker Hub README

9.5 Docker Hub MCP Integration

As of November 2025, Docker Hub has first-class MCP support[6] [7]:

Features:

- **MCP Catalog**: Discover 270+ MCP servers
- **One-click installation** from Docker Desktop
- **Secrets management**: Secure credential storage
- **OAuth integration**: Built-in authentication flows
- **Automatic updates**: Pull latest versions

To make your server discoverable:

1. **Tag repository** with `mcp` topic
2. **Add metadata** in Docker Hub settings:
   - MCP Server Name
   - Description
   - Required environment variables
   - Example configurations
3. **Submit to catalog** (if public):
   - Visit https://hub.docker.com/mcp/submit
   - Provide server details
   - Link to GitHub repository
4. **Publish agent card** (optional):
   - Create `.well-known/agent-card.json` in your domain
   - Describes capabilities and configuration

User experience:

1. User opens Docker Desktop → MCP Toolkit
2. Searches for "healthcare-mcp"
3. Clicks "Enable"
4. Docker Desktop prompts for `API_KEY` and `GRAPHQL_ENDPOINT`
5. Secrets stored securely (never exposed)
6. MCP server starts automatically
7. Available to all MCP clients (Claude, Cursor, etc.)

10. MCP vs REST: Design Philosophy {#mcp-vs-rest}

10.1 Fundamental Differences

Aspect	REST API	MCP Server
Primary Consumer	Human developers	LLM agents
Discovery	OpenAPI/Swagger docs	JSON-RPC capability negotiation + tool schemas
Interface Design	Resource-oriented (nouns)	Action-oriented (verbs/tools)
Granularity	Coarse endpoints (e.g., `/api/patients`)	Fine-grained tools (e.g., `search_patients`)
Documentation	Human-readable descriptions	Machine-readable schemas (JSON Schema)

Aspect	REST API	MCP Server
Statefulness	Typically stateless	Stateful connections
Error Handling	HTTP status codes	JSON-RPC error objects
Authorization	Per-request (JWT, API key)	Connection-level + OAuth Resource Server
Composability	Client-side chaining	LLM orchestrates multi-tool workflows

10.2 Design Principles for MCP

1. Small, Single-Purpose Tools

```
# ✗ REST-style: One endpoint, many actions
@mcp.tool()
def manage_patient(action: str, patient_id: str, data: dict):
    """
    Generic patient management.
    action: "get", "update", "delete", "list_appointments", etc.
    """
    if action == "get":
        # ...
    elif action == "update":
        # ...
    # (Hard for LLM to use correctly)

# ✓ MCP-style: Dedicated tools
@mcp.tool()
def get_patient(patient_id: str) -&gt; Patient:
    """Get patient by ID."""
    pass

@mcp.tool()
def update_patient(patient_id: str, name: str, email: str) -&gt; Patient:
    """Update patient information."""
    pass

@mcp.tool()
def delete_patient(patient_id: str) -&gt; bool:
    """Delete patient record."""
    pass

@mcp.tool()
def list_patient_appointments(patient_id: str) -&gt; List[Appointment]:
    """List appointments for patient."""
    pass
```

Why? LLMs are better at selecting the right tool from many specific options than parsing complex multi-action interfaces.

2. Explicit, Structured Schemas

```
# ✗ Vague types
@mcp.tool()
def search(query: str, options: dict) -&gt; list:
    """Search for stuff."""
    pass

# ✓ Precise types with constraints
@mcp.tool()
def search_patients(
    name: str = Field(..., min_length=2, max_length=100, description="Patient name"),
    status: Literal["active", "inactive", "all"] = "active",
    limit: int = Field(10, ge=1, le=100, description="Max results")
) -&gt; List[PatientSearchResult]:
    """
    Search patients by name.
```

```
    Returns patients matching the search criteria, ordered by relevance.
    Only active patients are returned by default.
    """
    pass
```

Why? LLMs use schemas to validate inputs before calling. Strong typing prevents errors.

3. Idempotence & Retry-Friendliness

```
# ✅ Idempotent operations
@mcp.tool()
def create_appointment(
    patient_id: str,
    datetime: str,
    provider_id: str,
    idempotency_key: Optional[str] = None  # Prevents duplicates
) -&gt; Appointment:
    """
    Create new appointment.

    If idempotency_key is provided and matches an existing appointment,
    returns that appointment instead of creating a duplicate.
    """
    if idempotency_key:
        existing = db.get_appointment_by_idempotency_key(idempotency_key)
        if existing:
            return existing

    # Create new appointment
    appointment = db.create_appointment(...)
    if idempotency_key:
        db.store_idempotency_key(idempotency_key, appointment.id)

    return appointment
```

Why? LLMs may retry failed operations. Idempotence prevents unintended side effects.

4. Structured, Actionable Errors

```
from fastmcp import ToolError

# ❌ Generic errors
@mcp.tool()
def bad_error_handling(patient_id: str):
    patient = db.get(patient_id)
    if not patient:
        raise Exception("not found")  # Unclear to LLM

# ✅ Structured errors
@mcp.tool()
def good_error_handling(patient_id: str) -&gt; Patient:
    """Get patient by ID."""
    patient = db.get(patient_id)

    if not patient:
        raise ToolError(
            f"Patient not found: {patient_id}. "
            f"Please verify the patient ID is correct. "
            f"Use search_patients() to find patients by name."
        )

    if patient.status == "deleted":
        raise ToolError(
            f"Patient {patient_id} has been deleted. "
            f"Deleted patients cannot be accessed."
        )
```

```
    return patient
```

Why? LLMs read error messages to adjust their approach. Actionable errors improve success rates.

5. Prefer Normalized Outputs

```
# ✖ Nested, redundant data (REST-style)
{
    "patient": {
        "id": "123",
        "name": "John Doe",
        "provider": {
            "id": "456",
            "name": "Dr. Smith",
            "specialty": "Cardiology",
            "hospital": {
                "id": "789",
                "name": "City Hospital",
                "address": {...}
            }
        },
        "appointments": [
            {
                "id": "appt-1",
                "provider": {
                    "id": "456",  # Duplicate data
                    "name": "Dr. Smith"
                }
            }
        ]
    }
}

# ✅ Normalized, tool-friendly (MCP-style)
{
    "id": "123",
    "name": "John Doe",
    "primary_provider_id": "456",
    "appointment_ids": ["appt-1", "appt-2"]
}

# LLM can call:
# - get_provider(provider_id="456")
# - get_appointment(appointment_id="appt-1")
```

Why? LLMs excel at chaining tool calls. Normalized data encourages compositional workflows.

10.3 Design Comparison: REST vs MCP

Example: Blog platform

REST API Design:

```
GET    /api/posts              # List posts
POST   /api/posts              # Create post
GET    /api/posts/:id          # Get post
PUT    /api/posts/:id          # Update post
DELETE /api/posts/:id          # Delete post
GET    /api/posts/:id/comments # List comments
POST   /api/posts/:id/comments # Create comment
```

MCP Server Design:

```
@mcp.tool()
def search_posts(
```

```
    query: Optional[str] = None,
    author_id: Optional[str] = None,
    tag: Optional[str] = None,
    limit: int = 20
) -&gt; List[PostSummary]:
    """Search blog posts by query, author, or tag."""

@mcp.tool()
def get_post(post_id: str) -&gt; PostDetail:
    """Get full post content by ID."""

@mcp.tool()
def create_post(
    title: str,
    content: str,
    tags: List[str],
    publish: bool = False
) -&gt; Post:
    """Create new blog post (draft or published)."""

@mcp.tool()
def update_post(
    post_id: str,
    title: Optional[str] = None,
    content: Optional[str] = None,
    tags: Optional[List[str]] = None
) -&gt; Post:
    """Update existing post (only provided fields)."""

@mcp.tool()
def publish_post(post_id: str, scheduled_time: Optional[str] = None) -&gt; Post:
    """Publish draft post immediately or schedule for later."""

@mcp.tool()
def delete_post(post_id: str) -&gt; bool:
    """Permanently delete post."""

@mcp.tool()
def get_post_comments(post_id: str, limit: int = 50) -&gt; List[Comment]:
    """List comments on a post."""

@mcp.tool()
def add_comment(post_id: str, content: str, author_name: str) -&gt; Comment:
    """Add comment to post."""
```

Key differences:

1. **MCP separates** `publish_post` from `update_post` (clear intent)
2. **MCP uses** specific search parameters instead of query string parsing
3. **MCP returns** typed objects, not generic JSON
4. **Each tool** has a single, clear purpose

10.4 Workflow Comparison

Scenario: "Find recent posts by Alice, get the most popular one, and summarize it."

REST approach (client-side):

```
# 1. Search posts by author
response = requests.get("/api/posts?author=Alice&amp;sort=date")
posts = response.json()["posts"]

# 2. Get comments for each to find most popular
popular_post = None
max_comments = 0
for post in posts[:10]:  # Limit search
    comments_response = requests.get(f"/api/posts/{post['id']}/comments")
```

```
    comment_count = len(comments_response.json()["comments"])
    if comment_count &gt; max_comments:
        max_comments = comment_count
        popular_post = post

# 3. Get full content
post_response = requests.get(f"/api/posts/{popular_post['id']}")
full_post = post_response.json()["post"]

# 4. Generate summary (external LLM call)
summary = llm.summarize(full_post["content"])
```

MCP approach (LLM-orchestrated):

```
User: "Find recent posts by Alice, get the most popular one, and summarize it."

LLM internal plan:
1. Call search_posts(author_id="alice", limit=10)
   → Returns: [post1, post2, ..., post10]

2. For each post, call get_post_comments(post_id)
   → Finds post5 has most comments

3. Call get_post(post_id="post5")
   → Returns full content

4. Generate summary using post content
   → Returns to user
```

LLM handles orchestration—your MCP just provides atomic tools.

10.5 Mental Model: MCP as a REPL for LLMs

Think of MCP tools as **Python functions** an LLM can call:

```
# REST: You provide a kitchen (API)
kitchen = RestAPI()
ingredients = kitchen.get_ingredients()
recipe = kitchen.find_recipe(ingredients)
meal = kitchen.cook(recipe)

# MCP: You provide utensils (tools)
# LLM is the chef who uses them
ingredients = get_ingredients()
if have_flour(ingredients):
    dough = make_dough()
    pizza = bake_pizza(dough)
else:
    salad = make_salad(ingredients)
```

Design implication: Make tools as simple and composable as possible.

11. Testing & Debugging {#testing-debugging}

11.1 Unit Testing Tools

tests/test_tools.py:

```
"""
Unit tests for MCP tools.
"""
import pytest
from unittest.mock import Mock, AsyncMock
from app.tools.patient_tools import register_patient_tools
```

```
from fastmcp import FastMCP, Context

@pytest.fixture
def mock_graphql_client():
    """Mock GraphQL client for testing."""
    client = Mock()
    client.execute = AsyncMock()
    return client

@pytest.fixture
def mcp_server(mock_graphql_client):
    """Create MCP server with mocked dependencies."""
    mcp = FastMCP("Test Server")
    register_patient_tools(mcp, mock_graphql_client)
    return mcp

@pytest.mark.asyncio
async def test_search_patients_success(mcp_server, mock_graphql_client):
    """Test successful patient search."""
    # Setup mock response
    mock_graphql_client.execute.return_value = {
        "data": {
            "searchPatients": [
                {"id": "123", "name": "John Doe", "status": "active"}
            ]
        }
    }

    # Call tool
    result = await mcp_server.call_tool("search_patients", {
        "name": "John",
        "status": "active",
        "limit": 10
    })

    # Assertions
    assert len(result) == 1
    assert result[^0]["name"] == "John Doe"

    # Verify GraphQL call
    mock_graphql_client.execute.assert_called_once()
    call_args = mock_graphql_client.execute.call_args
    assert "SearchPatients" in call_args[^0][^0]  # Query
    assert call_args[^1]["variables"]["name"] == "John"

@pytest.mark.asyncio
async def test_search_patients_validation_error(mcp_server):
    """Test input validation."""
    from fastmcp import ToolError

    with pytest.raises(ToolError) as exc_info:
        await mcp_server.call_tool("search_patients", {
            "name": "J",  # Too short
            "status": "active",
            "limit": 10
        })

    assert "at least 2 characters" in str(exc_info.value)

@pytest.mark.asyncio
async def test_search_patients_graphql_error(mcp_server, mock_graphql_client):
    """Test GraphQL error handling."""
    from app.utils.graphql_client import GraphQLError
    from fastmcp import ToolError

    # Mock GraphQL error
    mock_graphql_client.execute.side_effect = GraphQLError("Invalid query")

    with pytest.raises(ToolError) as exc_info:
        await mcp_server.call_tool("search_patients", {
            "name": "John",
            "status": "active",
```

```
            "limit": 10
        })

    assert "Failed to search patients" in str(exc_info.value)
```

11.2 Integration Testing

tests/test_graphql.py:

```
"""
Integration tests for GraphQL client.
"""
import pytest
from app.utils.graphql_client import GraphQLClient, GraphQLError

@pytest.fixture
def graphql_client():
    """Create real GraphQL client (requires test API)."""
    return GraphQLClient(
        endpoint="https://api.test.example.com/graphql",
        api_key="sk_test_key",
        timeout=10
    )

@pytest.mark.integration
@pytest.mark.asyncio
async def test_introspection_query(graphql_client):
    """Test schema introspection (smoke test)."""
    schema = await graphql_client.introspect_schema()

    assert "queryType" in schema
    assert "types" in schema
    assert len(schema["types"]) &gt; 0

@pytest.mark.integration
@pytest.mark.asyncio
async def test_search_patients_integration(graphql_client):
    """Test real patient search."""
    query = """
        query SearchPatients($name: String!) {
            searchPatients(name: $name, limit: 5) {
                id
                name
            }
        }
    """

    result = await graphql_client.execute(
        query,
        {"name": "test"},
        operation_name="SearchPatients"
    )

    assert "data" in result
    assert "searchPatients" in result["data"]
```

11.3 Manual Testing with FastMCP

Interactive testing:

```
# Start server in development mode
fastmcp dev app/server.py --log-level DEBUG

# In another terminal, use FastMCP client
python -c "
from fastmcp import Client
import asyncio
```

```
async def test():
    async with Client('http://localhost:8080') as client:
        # List available tools
        tools = await client.list_tools()
        print('Available tools:', tools)

        # Call a tool
        result = await client.call_tool('search_patients', {
            'name': 'John',
            'status': 'active',
            'limit': 5
        })
        print('Result:', result)

asyncio.run(test())
"
```

11.4 Debugging with MCP Inspector

MCP Inspector is a visual debugging tool for MCP servers[5]:

```
# Install
npm install -g @modelcontextprotocol/inspector

# Run your server
fastmcp run app/server.py &amp;

# Start inspector
mcp-inspector http://localhost:8080

# Opens browser UI showing:
# - Available tools and their schemas
# - Real-time logs
# - Interactive tool calling
# - Message trace
```

11.5 Logging Best Practices

```
import logging
from fastmcp import Context

logger = logging.getLogger(__name__)

@mcp.tool()
async def instrumented_tool(ctx: Context, patient_id: str):
    """Tool with comprehensive logging."""
    # Use Context for user-visible logs
    ctx.info(f"Searching for patient: {patient_id}")

    # Use standard logging for server-side logs
    logger.debug(f"Tool called with patient_id={patient_id}")

    try:
        result = await graphql_client.execute(...)

        logger.info(f"Query succeeded: {len(result)} results")
        ctx.info(f"Found patient: {result['name']}")

        return result

    except GraphQLError as e:
        # Log error details server-side
        logger.error(f"GraphQL error: {e}", exc_info=True)

        # Send user-friendly message to client
        ctx.error(f"Failed to fetch patient: {patient_id}")
```

```
        raise ToolError(f"Unable to retrieve patient information")
```

Logging levels:

- `DEBUG`: Detailed info for development
- `INFO`: Important state changes
- `WARN`: Recoverable issues
- `ERROR`: Failures requiring attention

Context methods:

- `ctx.info()`: Show info to MCP client user
- `ctx.warning()`: Show warning to user
- `ctx.error()`: Show error to user
- `ctx.debug()`: Debug info for user (if enabled)

12. Quick Start Checklist {#quick-start-checklist}

From Zero to Running MCP in 30 Minutes

- [ ] **Setup (5 minutes)**
  - [ ] Install Python 3.10+
  - [ ] Install Docker
  - [ ] `pip install fastmcp httpx python-dotenv`
  - [ ] Create project directory structure
- [ ] **Configuration (3 minutes)**
  - [ ] Create `.env.example` with required variables
  - [ ] Copy to `.env` and fill in actual values
  - [ ] Add `.env` to `.gitignore`
- [ ] **Server Implementation (10 minutes)**
  - [ ] Create `app/server.py` with FastMCP instance
  - [ ] Implement 2-3 basic tools (can use mock data initially)
  - [ ] Add health check resource
  - [ ] Test bare-metal: `fastmcp dev app/server.py`
- [ ] **Dockerization (5 minutes)**
  - [ ] Create `Dockerfile` (copy template from section 8.1)
  - [ ] Build: `docker build -t my-mcp:latest .`
  - [ ] Test: `docker run -i --rm -e API_KEY=test my-mcp:latest`
- [ ] **Client Integration (5 minutes)**
  - [ ] Choose client (Claude Desktop, VS Code, Cursor)
  - [ ] Add MCP server configuration
  - [ ] Restart client and verify connection
- [ ] **Validation (2 minutes)**
  - [ ] Test tool calls from MCP client
  - [ ] Check logs for errors
  - [ ] Verify secrets are not logged

Hackathon mode: Skip Docker initially—run bare-metal and containerize later if needed.

13. Design Checklist {#design-checklist}

Agent-Friendly MCP Design

Tool Design:

- [ ] Each tool does **one thing well**
- [ ] Tool names are **action-oriented verbs** (e.g., `search_patients`, not `patients`)
- [ ] Docstrings are **clear and specific** (LLM reads these)
- [ ] Input parameters have **strong types** (Literal, Enum, Pydantic models)
- [ ] Parameters include **Field() descriptions and constraints**
- [ ] Return types are **explicit and structured**
- [ ] Outputs are **normalized** (IDs instead of nested objects)

Error Handling:

- [ ] Custom errors use `ToolError` with **actionable messages**
- [ ] Errors suggest **next steps** or alternative tools
- [ ] Validation happens **before** external API calls
- [ ] Errors don't expose **internal implementation details**

Security:

- [ ] **No secrets** hard-coded or logged
- [ ] All secrets loaded from **environment variables**
- [ ] API keys validated at **startup** (fail fast)
- [ ] Input sanitization prevents **injection attacks**
- [ ] Errors don't leak **sensitive data**

GraphQL Integration:

- [ ] Each tool maps to **specific GraphQL operation**
- [ ] Queries are **fixed** (not user-provided)
- [ ] Responses are **normalized** before returning to LLM
- [ ] GraphQL errors are **caught and translated** to ToolErrors
- [ ] Retries are implemented for **transient failures**

Operational:

- [ ] Health check resource implemented
- [ ] Logging includes **structured data** (not just strings)
- [ ] Server metadata exposed via resource
- [ ] README documents **all required env vars**
- [ ] Example configurations provided for **major MCP clients**

Docker:

- [ ] Image runs as **non-root user**
- [ ] No secrets in **image layers**
- [ ] `.dockerignore` excludes unnecessary files
- [ ] Health check defined in Dockerfile
- [ ] Multi-platform build (amd64, arm64) if publishing

Documentation:

- [ ] README has **quick start instructions**

- [ ] Each tool's purpose is **clear from description**
- [ ] Required vs optional env vars **clearly marked**
- [ ] Example configurations for **3+ MCP clients**
- [ ] Troubleshooting section addresses **common issues**

References

[1] Model Context Protocol Specification Update. https://modelcontextprotocol.info/blog/mcp-next-version-update/

[5] Build MCP Servers in Python with FastMCP. https://mcpcat.io/guides/building-mcp-server-python-fastmcp/

[8] MCP Server Best Practices. https://docs.glean.com/administration/platform/mcp/best-practices

[3] MCP Specs Update - OAuth & Security. https://auth0.com/blog/mcp-specs-update-all-about-auth/

[9] How to Build Your Own MCP Server with Python. https://www.freecodecamp.org/news/how-to-build-your-own-mcp-server-with-python/

[4] MCP Security Best Practices. https://modelcontextprotocol.io/specification/draft/basic/security_best_practices

[2] MCP Specification. https://modelcontextprotocol.io/specification/2025-06-18

[6] Introducing Docker Hub MCP Server. https://www.docker.com/blog/introducing-docker-hub-mcp-server/

[7] Docker Embraces MCP for AI Agent Integration. https://cloudnativenow.com/news/docker-inc-embraces-mcp-to-make-ai-agent-integration-simpler/

End of Guide

Next Steps:

1. Clone the example project: `git clone https://github.com/examples/healthcare-mcp`
2. Follow Quick Start Checklist
3. Join MCP community: https://discord.gg/mcp
4. Explore MCP Registry: https://modelcontextprotocol.io/registry

Happy hacking!

[10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40]

⁂

1. https://modelcontextprotocol.info/blog/mcp-next-version-update/
2. https://modelcontextprotocol.io/specification/2025-06-18
3. https://auth0.com/blog/mcp-specs-update-all-about-auth/
4. https://modelcontextprotocol.io/specification/draft/basic/security_best_practices
5. https://mcpcat.io/guides/building-mcp-server-python-fastmcp/
6. https://www.docker.com/blog/introducing-docker-hub-mcp-server/
7. https://cloudnativenow.com/news/docker-inc-embraces-mcp-to-make-ai-agent-integration-simpler/
8. https://docs.glean.com/administration/platform/mcp/best-practices
9. https://www.freecodecamp.org/news/how-to-build-your-own-mcp-server-with-python/
10. https://modelcontextprotocol.io/specification/2025-06-18/basic
11. https://gofastmcp.com/getting-started/quickstart
12. https://scalesec.com/blog/mcp-server-security-best-practices
13. https://www.anthropic.com/news/model-context-protocol
14. https://github.com/jlowin/fastmcp
15. https://github.blog/ai-and-ml/generative-ai/how-to-build-secure-and-scalable-remote-mcp-servers/
16. https://modelcontextprotocol.io/development/roadmap

17. https://www.firecrawl.dev/blog/fastmcp-tutorial-building-mcp-servers-python
18. https://www.reddit.com/r/mcp/comments/1kujaxw/does_anyone_have_a_best_practices_guide_or/
19. https://code.visualstudio.com/blogs/2025/06/12/full-mcp-spec-support
20. https://apidog.com/blog/fastmcp/
21. https://www.kagent.dev/docs/kmcp/secrets
22. https://www.flowhunt.io/mcp-servers/graphql/
23. https://www.youtube.com/watch?v=Jy5AseyqEcg
24. https://docs.stacklok.com/toolhive/guides-cli/run-mcp-servers
25. https://glama.ai/mcp/servers/@marias1lva/mcp-graphql-generator
26. https://1password.com/blog/securing-mcp-servers-with-1password-stop-credential-exposure-in-your-agent
27. https://github.com/hannesj/mcp-graphql-schema
28. https://apidog.com/blog/how-to-use-the/
29. https://www.reddit.com/r/mcp/comments/1kb01dg/please_stop_storing_secrets_in_env/
30. https://skywork.ai/skypage/en/graphql-mcp-server-ai-engineers-guide/1981566520480862208
31. https://www.ajeetraina.com/5-docker-mcp-servers-every-frontend-and-backend-developer-must-be-aware-of-in-2025/
32. https://github.com/anthropics/claude-code/issues/2065
33. https://learn.microsoft.com/en-us/fabric/data-engineering/api-graphql-local-model-context-protocol
34. https://hub.docker.com/mcp
35. https://workos.com/guide/best-practices-for-mcp-secrets-management
36. https://www.apollographql.com/blog/the-future-of-mcp-is-graphql
37. https://hub.docker.com/u/mcp
38. https://code.visualstudio.com/docs/copilot/customization/mcp-servers
39. https://www.youtube.com/watch?v=rnljvmHorQw
40. https://modelcontextprotocol.info/docs/best-practices/