Pokémon MCP Server Documentation

Table of Content:

1) Project Overview
2) System Architecture
3) System Installation
4) API Endpoints
5) Modular Components
6) AI-Agent Integration Guide
7)Web Interface Guide
8)Team Builder Showcase
9) LangSmith Integration
10) Logging & Monitoring
11) Testing
12) Deployment
13) Troubleshooting

Project Overview

The Pokémon MCP (Modular Control Platform) Server is a RESTful middleware solution that provides structured access to Pokémon data for AI agents and web applications. Built with FastAPI and Python, it abstracts the complexity of the PokeAPI while providing intelligent features powered by Google's Gemini AI and comprehensive prompt tracking via LangSmith.

Key Features

- **Modular Architecture**: Extensible component-based design
- **AI-Powered Team Generation**: Natural language team building
- **Strategic Analysis**: Counter-Pokémon suggestions
- **RESTful API**: Clean, well-documented endpoints
- **CORS-Enabled**: Ready for web frontend integration
- **Comprehensive Logging**: Full request/error tracking
- **LangSmith Integration**: Advanced prompt tracking and analytics
- **Cloud Deployment**: Production-ready deployment on Render

Technology Stack

- **Backend**: FastAPI (Python)
- **AI Integration**: Google Gemini (via LangChain)
- **Prompt Tracking**: LangSmith
- **External API**: PokeAPI
- **Frontend**: Streamlit
- **Environment Management**: python-dotenv
- **Deployment**: Render (Cloud Platform)

System Architecture

Modular Components

1. Information Retrieval Module

- **Purpose**: Fetches and structures Pokémon data from PokeAPI
- **Function**: _fetch_pokemon_data_from_pokeapi()
- **Features**:
    - Data normalization (height/weight conversion)
    - Error handling with detailed HTTP status codes
    - Comprehensive logging

2. Comparison Module

- **Purpose**: Enables side-by-side Pokémon comparisons
- **Endpoint**: /compare/{pokemon1_name}/{pokemon2_name}
- **Features**:
    - Parallel data fetching
    - Structured comparison output
    - Error propagation handling

3. Strategy Module (AI-Powered)

- **Purpose**: Provides strategic battle recommendations
- **Endpoint**: /counters/{pokemon_name}
- **Features**:
    - AI-generated counter suggestions
    - Type effectiveness analysis
    - Strategic reasoning explanations
    - LangSmith prompt tracking

4. Team Composition Module (AI-Powered)

- **Purpose**: Generates balanced teams from natural language descriptions
- **Endpoint**: /team/generate
- **Features**:
    - Natural language processing
    - Team balance optimization
    - Role-based suggestions
    - LangSmith conversation tracking

Installation & Setup

Prerequisites

- Python 3.8+
- Google Gemini API Key
- LangSmith API Key (for tracking)
- Internet connection (for PokeAPI access)

Step 1: Environment Setup

Clone the repository

git clone <your-repo-url>

cd pokemon-mcp-server

or copy from the github

Create virtual environment

python -m venv venv

Activate virtual environment

On Windows:

venv\Scripts\activate / for mac source venv/bin/activate

Step 2: Install Dependencies

pip install fastapi uvicorn requests python-dotenv langchain-google-genai langchain-core langsmith streamlit

Step 3: Environment Configuration

Create a .env file in the project root:

GEMINI_API_KEY=your_gemini_api_key_here

LANGCHAIN_TRACING_V2=true

LANGCHAIN_API_KEY=your_langsmith_api_key_here

LANGCHAIN_PROJECT=pokemon-mcp-server

Step 4: Run the Server

Start FastAPI server

uvicorn main:app --host 0.0.0.0 --port 8000 --reload

In a separate terminal, start Streamlit frontend

streamlit run streamlit_app.py --server.port 8501

Step 5: Verify Installation

- Backend: Navigate to http://localhost:8000/docs for API documentation
- Frontend: Navigate to http://localhost:8501 for web interface
- LangSmith: Check your LangSmith dashboard for tracking data

API Endpoints

Base URL: http://localhost:8000 (Development) | https://your-app.onrender.com (Production)

1. Get Pokémon Details

url /pokemon/{name}

Description: Retrieve comprehensive information for a single Pokémon

2. Compare Pokémon

url /compare/{pokemon1_name}/{pokemon2_name}

Description: Compare attributes of two Pokémon side-by-side

3. Generate Team

url /team/generate

Description: Generate a 6-Pokémon team from natural language description

Request Body:

- description (string): Natural language team requirements

LangSmith Tracking: All team generation requests are automatically tracked with:

- User prompts and descriptions
- AI model responses
- Generation timestamps
- Performance metrics

4. Suggest Counters

url/counters/{pokemon_name}

Description: Get AI-powered counter suggestions for a specific Pokémon

Parameters:

- pokemon_name (string): Target Pokémon name

LangSmith Tracking: Counter suggestion requests are tracked with:

- Target Pokémon information
- Generated counter strategies
- AI reasoning processes

AI-Agent Integration Guide

Overview

The MCP server provides a simplified abstraction layer that allows AI agents to interact with Pokémon data without dealing with raw API complexity. All AI interactions are automatically tracked via LangSmith.

Integration Methods

1. Direct HTTP Client Integration

```
import requests
   base_url= ”mcp url”
   def get_pokemon(name):
      response = requests.get(f"{base_url}/pokemon/{name}")
      return response.json()
  def compare_pokemon(spokemon1, pokemon2):
      response = =requests.get(f"{self.base_url}/compare/{pokemon1}/{pokemon2}")
      return response.json()
  def generate_team( description):
      response = requests.post(
         f"{base_url}/team/generate",
         json={"description": description}
      )
      return response.json()
```

Agent Interaction Patterns

Pattern 1: Information Gathering

Agent: "Tell me about Pikachu"

→ GET /pokemon/pikachu

→ Process structured data

→ Generate natural language response

→ LangSmith tracks the entire conversation

Pattern 2: Strategic Analysis

Agent: "What counters Charizard?"

→ GET /counters/charizard

→ Analyze AI-generated suggestions

→ Provide strategic recommendations

→ LangSmith captures strategy generation process

Pattern 3: Team Building

Agent: "Build a team for competitive play"

→ POST /team/generate with refined description

→ Process team suggestions

→ Provide detailed analysis

→ LangSmith logs full team building workflow

Web Interface Guide

Streamlit Frontend Features

The web interface provides an intuitive way to interact with all MCP server capabilities:

1. Pokémon Search

- **Input**: Pokémon name
- **Output**: Detailed information card with stats and sprite
- **Features**: Real-time search, error handling

2. Pokémon Comparison

- **Input**: Two Pokémon names
- **Output**: Side-by-side comparison table
- **Features**: Visual stat comparison, type analysis

3. Counter Suggestions

- **Input**: Target Pokémon name
- **Output**: List of recommended counters with reasoning
- **Features**: AI-powered analysis, strategic explanations

4. Team Generation

- **Input**: Natural language team description
- **Output**: Balanced 6-Pokémon team with roles
- **Features**: Intelligent parsing, role assignment

Usage Instructions

1. **Start the servers**:
2. # Terminal 1: FastAPI backend
3. uvicorn main:app –reload
4. # Terminal 2: Streamlit frontend
5. streamlit run streamlit_app.py
6. **Navigate to** http://localhost:8501
7. **Select a feature** from the sidebar
8. **Input your query** and view results

Team Builder Showcase

Natural Language Processing Capabilities

The team builder accepts various description formats and tracks all interactions via LangSmith:

Example Inputs:

- **Balanced Team**: "Create a balanced team for competitive play"
- **Type-Specific**: "I need a team with strong fire and water types"
- **Role-Based**: "Build a defensive team with one sweeper"
- **Theme-Based**: "Create a team of legendary Pokémon"
- **Strategy-Focused**: "I want a team that counters Dragon types"

AI Processing Flow:

1. **Parse Description**: Extract key requirements and preferences
2. **Role Assignment**: Determine team roles (attacker, tank, support, etc.)
3. **Type Balancing**: Ensure type diversity and coverage
4. **Synergy Analysis**: Consider Pokémon combinations

5. **Output Generation**: Format as structured JSON and **LangSmith Loggin**
6. **Example Output:**

```
{
  "team": [
    {
      "name": "Charizard",
      "main_type": "Fire",
      "role": "Physical Attacker",
      "reasoning": "Strong fire-type with good offensive stats"
    },
    {
      "name": "Lapras",
      "main_type": "Water",
      "role": "Tank",
      "reasoning": "High HP and defensive capabilities"
    }
    // ... additional team members
  ]
}
```

Advanced Team Building Features

1. Role Diversity

- Physical Attackers
- Special Attackers
- Tanks/Walls
- Support/Utility
- Sweepers
- Entry Hazard Setters

2. Type Coverage

- Offensive type diversity
- Defensive type balance
- Coverage for common threats
- Synergistic type combinations

3. Strategic Considerations

- Speed control
- Priority moves
- Weather strategies
- Entry hazards
- Status conditions

LangSmith Integration

Overview

LangSmith provides comprehensive tracking and analytics for all AI-powered interactions in the Pokémon MCP Server. This enables detailed monitoring of prompt effectiveness, response quality, and user behavior patterns.

Features

1. Automatic Prompt Tracking

- All team generation requests are automatically logged
- Counter suggestion prompts and responses tracked
- Natural language processing workflows monitored
- AI model performance metrics collected

2. Conversation Analytics

- User prompt patterns and preferences
- Most requested team types and strategies
- AI response accuracy and relevance
- Performance bottlenecks identification

3. Quality Monitoring

- Response quality assessment
- Prompt engineering effectiveness
- Model output consistency
- Error rate tracking

Setup Configuration

Environment Variables

LangSmith Configuration

LANGCHAIN_TRACING_V2=true

LANGCHAIN_API_KEY=your_langsmith_api_key_here

LANGCHAIN_PROJECT=pokemon-mcp-server

LANGCHAIN_ENDPOINT=https://api.smith.langchain.com

Dashboard Metrics

1. Usage Analytics

- **Total Requests**: Number of AI-powered requests per day/week/month
- **Popular Features**: Most used endpoints (team generation vs counter suggestions)
- **User Patterns**: Peak usage times and request frequencies
- **Geographic Distribution**: Request origins (if available)

2. Performance Metrics

- **Response Times**: Average AI response latency
- **Success Rates**: Percentage of successful AI completions
- **Error Analysis**: Common failure patterns and causes
- **Token Usage**: AI model token consumption tracking

3. Quality Insights

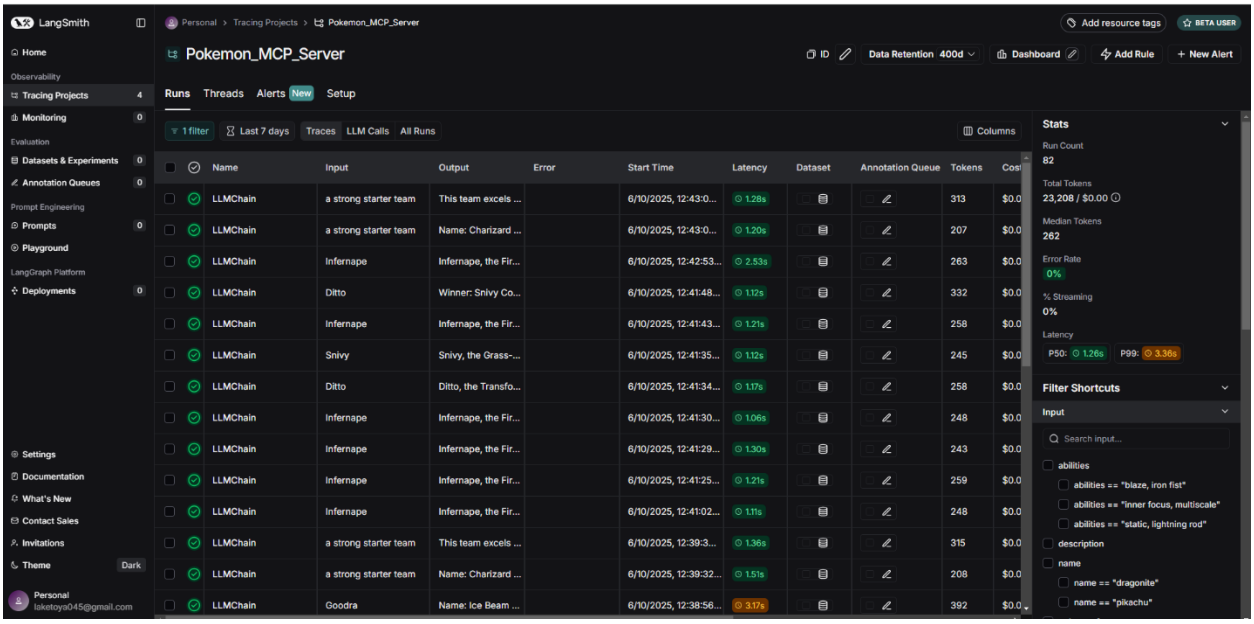
- **Prompt Effectiveness**: Which prompts generate better responses
- **Response Relevance**: User satisfaction indicators
- **Model Comparison**: Performance across different AI models
- **Improvement Opportunities**: Areas for prompt optimization

Benefits

1. Continuous Improvement

- Identify and optimize underperforming prompts
- Track user satisfaction and adjust accordingly

- Monitor AI model performance over time



2. User Experience Enhancement

- Understand user preferences and behavior
- Personalize responses based on usage patterns
- Reduce response times through optimization
- Improve accuracy through data-driven insights

3. Business Intelligence

- Usage growth tracking and forecasting
- Feature adoption rates and success metrics
- Cost optimization through efficient AI usage
- Strategic planning based on user data

Privacy and Security

Data Protection

- User prompts are tracked for analytics only

- No personal information is stored or transmitted
- All data follows GDPR and privacy best practices
- Option to disable tracking for sensitive applications

Access Control

- LangSmith dashboard access restricted to authorized users
- API keys managed securely through environment variables

Logging & Monitoring

Server-Side Logging

The system implements comprehensive logging using Python's built-in logging module combined with LangSmith analytics:

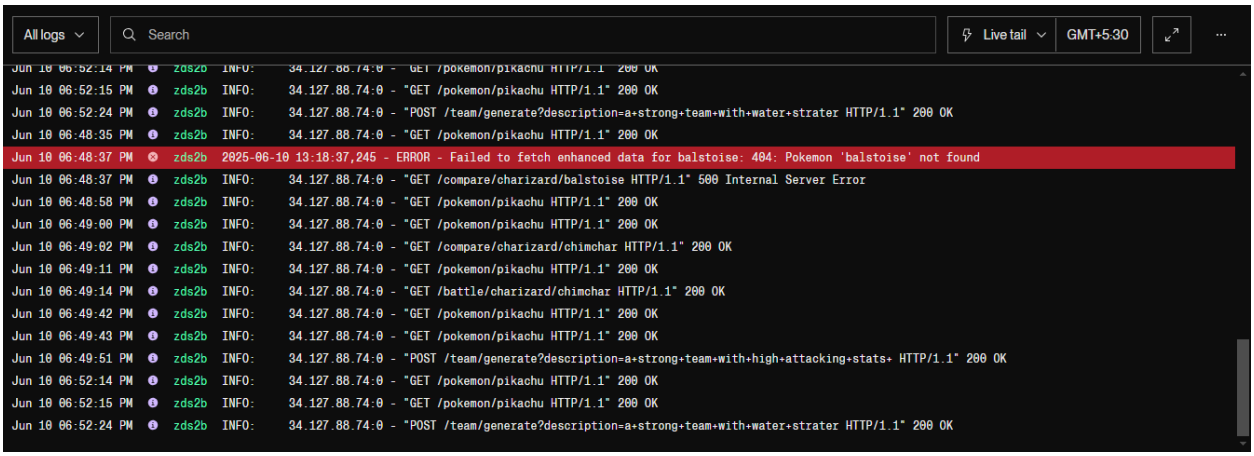
Log Levels:

- **INFO**: Successful operations, request tracking
- **WARNING**: Non-critical issues, missing data
- **ERROR**: API errors, connection issues
- **DEBUG**: LangSmith tracking events and AI model interactions
- **EXCEPTION**: Unexpected errors with stack traces

Logged Events:

- Successful Pokémon data retrieval
- API request failures
- HTTP errors with status codes
- Connection timeouts
- Team generation requests
- Counter suggestion requests

- LangSmith trace creation and completion
- AI model response times and token usage



Monitoring Capabilities

Performance Metrics:

- Response times per endpoint
- Success/failure rates
- Most requested Pokémon
- Error frequency by type
- AI model performance metrics
- LangSmith trace statistics

Advanced Monitoring with LangSmith

Real-time Dashboards

- Live request monitoring and analytics
- AI model performance metrics
- User interaction patterns
- Error rate tracking and alertin

- **Historical Analysis**
- Long-term usage trends and patterns
- Seasonal variations in request types
- Model performance evolution over time
- Cost analysis and optimization opportunities

Testing

Testing Strategy

Unit Tests

Test individual components in isolation:

```
import pytest
from main import _fetch_pokemon_data_from_pokeapi
def test_fetch_pokemon_valid_name():
    result = _fetch_pokemon_data_from_pokeapi("pikachu")
    assert result["name"] == "Pikachu"
    assert result["id"] == 25
    assert "Electric" in result["types"]
def test_fetch_pokemon_invalid_name():
    with pytest.raises(HTTPException) as exc_info:
        _fetch_pokemon_data_from_pokeapi("fakemon")
    assert exc_info.value.status_code == 404
```

Integration Tests

Test API endpoints end-to-end:

```
from fastapi.testclient import TestClient
from main import app
client = TestClient(app)
def test_get_pokemon_endpoint():
    response = client.get("/pokemon/pikachu")
    assert response.status_code == 200
    assert response.json()["name"] == "Pikachu"
def test_compare_pokemon_endpoint():
    response = client.get("/compare/pikachu/charizard")
    assert response.status_code == 200
    assert "pokemon1" in response.json()
    assert "pokemon2" in response.json()
```

AI Integration Tests

Test Gemini AI functionality with LangSmith tracking:

```
def test_team_generation():
    response = client.post("/team/generate",
                           json={"description": "balanced team"})
    assert response.status_code == 200
    team = response.json()["team"]
    assert len(team) <= 6
    assert all("name" in pokemon for pokemon in team)
```

```
def test_langsmith_tracking():
    # Test that LangSmith traces are created
    # Note: This requires LangSmith test environment
    import os
    assert os.getenv("LANGCHAIN_TRACING_V2") == "true"
    assert os.getenv("LANGCHAIN_API_KEY") is not None
```

Deployment Tests

Test production deployment on Render:

```
import requests

def test_render_deployment():
    """Test the deployed API on Render"""
    base_url = "https://your-app.onrender.com"

    # Test health endpoint
    response = requests.get(f"{base_url}/health")
    assert response.status_code == 200

    # Test basic Pokemon endpoint
    response = requests.get(f"{base_url}/pokemon/pikachu")
    assert response.status_code == 200
    assert response.json()["name"] == "Pikachu"
```

Running Tests

```
# Install pytest
pip install pytest pytest-asyncio

# Run all tests
pytest

# Run with coverage
pip install pytest-cov
pytest --cov=main --cov-report=html

# Run deployment tests (requires production URL)
pytest tests/test_deployment.py --url=https://your-app.onrender.com
```

Test Coverage Goals

- **Unit Tests**: 90%+ coverage of core functions
- **Integration Tests**: All endpoints tested
- **Error Handling**: All exception paths covered
- **AI Integration**: Response parsing reliability
- **LangSmith Integration**: Tracking functionality verified
- **Deployment**: Production environment validation

Deployment

Development Environment

```
# Development server with auto-reload
uvicorn main:app --reload --host 0.0.0.0 --port 8000
```

Production Deployment on Render

Step 1: Prepare Your Repository

Ensure your repository contains these files:

requirements.txt:

```
fastapi==0.104.1
uvicorn==0.24.0
requests==2.31.0
python-dotenv==1.0.0
langchain-google-genai==1.0.10
langchain-core==0.1.52
langsmith==0.0.87
gunicorn==21.2.0
```

render.yaml (optional, for Infrastructure as Code):

```
services:
  - type: web
    name: pokemon-mcp-server
    env: python
    buildCommand: pip install -r requirements.txt
    startCommand: gunicorn main:app -w 4 -k uvicorn.workers.UvicornWorker --bind 0.0.0.0:$PORT
    envVars:
      - key: GEMINI_API_KEY
        sync: false
      - key: LANGCHAIN_TRACING_V2
        value: true
      - key: LANGCHAIN_API_KEY
        sync: false
```

- key: LANGCHAIN_PROJECT

  value: pokemon-mcp-server-prod

Step 2: Deploy to Render

1. **Connect Repository**:
    - Go to Render Dashboard
    - Click "New" → "Web Service"
    - Connect your GitHub repository
2. **Configure Build Settings**:
    - **Name**: pokemon-mcp-server
    - **Environment**: Python 3
    - **Build Command**: pip install -r requirements.txt
    - **Start Command**: gunicorn main:app -w 4 -k uvicorn.workers.UvicornWorker --bind 0.0.0.0:$PORT
3. **Set Environment Variables**:
4. GEMINI_API_KEY=your_gemini_api_key_here
5. LANGCHAIN_TRACING_V2=true
6. LANGCHAIN_API_KEY=your_langsmith_api_key_here
7. LANGCHAIN_PROJECT=pokemon-mcp-server-prod
8. LANGCHAIN_ENDPOINT=https://api.smith.langchain.com
9. **Deploy**:
    - Click "Create Web Service"
    - Render will automatically build and deploy your application
    - Your app will be available at https://your-app-name.onrender.com

Step 3: Verify Deployment

```
# Test the deployed API
curl https://your-app.onrender.com/pokemon/pikachu

# Check health endpoint
curl https://your-app.onrender.com/health

# Test team generation
curl -X POST https://your-app.onrender.com/team/generate \
  -H "Content-Type: application/json" \
  -d '{"description": "balanced team for competitive play"}'
```

Production Environment Variables

```
# Production environment
GEMINI_API_KEY=your_production_api_key
LANGCHAIN_TRACING_V2=true
LANGCHAIN_API_KEY=your_production_langsmith_key
LANGCHAIN_PROJECT=pokemon-mcp-server-prod
ENVIRONMENT=production
LOG_LEVEL=INFO
PORT=8000
```

Scaling Considerations on Render

- **Horizontal Scaling**: Multiple service instances
- **Vertical Scaling**: Upgrade to higher-tier plans
- **Auto-Scaling**: Configure based on CPU/memory usage
- **Load Balancing**: Automatic load distribution
- **CDN Integration**: Static asset optimization

Monitoring Production Deployment

Render Dashboard Metrics

- **Deployment Status**: Build and deployment success/failure
- **Resource Usage**: CPU, memory, and network utilization
- **Request Analytics**: Traffic patterns and response times
- **Error Tracking**: Application errors and exceptions

LangSmith Production Monitoring

- **Separate Production Project**: Isolate production traces
- **Performance Monitoring**: Track AI model response times
- **Cost Tracking**: Monitor API usage and costs
- **Quality Assurance**: Production prompt effectiveness

Health Checks and Monitoring

```
# Add to main.py
@app.get("/health")
async def health_check():
    return {
        "status": "healthy",
        "timestamp": datetime.utcnow(),
        "environment": os.getenv("ENVIRONMENT", "development"),’
```

Conclusion

The Pokémon MCP Server provides a robust, modular platform for AI agents and applications to interact with Pokémon data. With its clean API design, comprehensive error handling, and AI-powered features, it serves as an excellent foundation for strategic Pokémon applications.

Key Strengths:

- **Modular Architecture**: Easy to extend and maintain
- **AI Integration**: Intelligent team building and strategy
- **Comprehensive Documentation**: Clear setup and usage guides
- **Error Handling**: Robust error management and logging
- **CORS Support**: Ready for web frontend integration

Thank you