MCP Repository Analyzer – Final Detailed Project Report

1. Introduction
This report provides a complete, detailed analysis of the development of an ML-driven MCP (Model Context Protocol) Repository Analyzer. The goal of the project was to design a system capable of analyzing an entire Git repository or ZIP and automatically extracting MCP characteristics, including transport type, tools, input/output schemas, syscalls, run templates, and confidence scores. The solution is multi-language capable and integrates both static analysis and machine learning refinement.

2. Project Journey and Evolution

The development followed an iterative journey marked by trial, error, and continuous improvement.

Stage 1 – Initial Approach (API-based LLMs)

The project began with attempts to use cloud-based LLMs such as OpenAI and Gemini to analyze entire repositories. These models were powerful but unsuitable due to limits on:
- API rate restrictions
- Large file processing limitations
- Inconsistent schema extraction

This approach was abandoned as it didn't provide consistent large-scale repository analysis.

Stage 2 – Local LLMs with Ollama

Next, local LLMs were tested through Ollama, but large models such as Qwen and Llama produced errors due to hardware constraints and model incompatibility. The system struggled with CPU execution and often crashed or froze.

Stage 3 – Static Analysis Pipeline

The breakthrough came with designing a full static-analysis-based MCP extractor. This approach did not require model inference initially and handled Python repositories extremely well. The first reliable MCP analysis pipeline was developed here.

Stage 4 – ML Refinement Integration

To fulfill assignment requirements, ML refinement became necessary. A lightweight model (GPT■2) replaced heavier models. This allowed refining tool descriptions and adding confidence scores efficiently on a low-resource CPU laptop.

3. Dataset Creation

A manual dataset was created by extracting examples of MCP tools, schemas, and descriptions from known repositories. These examples were used to design prompts for the ML refinement model. The dataset included:

- Tool names
- Descriptions
- Input/output schemas
- Payload shapes
- Example code snippets

This dataset supported prompt engineering for GPT■2 refinement.

4. Multi-Language Analyzer Architecture
To fulfill the requirement of “supporting as many languages as possible”, a modular architecture was created:

Components:
- **language_detector.py** — detects languages (Python, TypeScript, JS, Go, Java, Rust).
- **schema_extractor_v3.py** — full-featured Python MCP extractor.
- Language-specific extractors for TS, JS, Go, Java, Rust (partial support).
- **multi_lang_analyzer.py** — orchestrates full pipeline.

The architecture ensures future languages can be added simply by plugging in new extractors.

5. Python MCP SDK – Full Static Analysis

The Python SDK was the main target repository due to its complete MCP implementations. Static analysis successfully extracted:

- 325 MCP tools
- Input schemas
- Output schemas
- Payload structures
- Class-based schema definitions
- Transport definitions (stdio/websocket/http)
- Syscalls (scanned across the entire repository)
- Run template prediction

Tools were saved into:

- mcp_analysis_output.json
- mcp_analysis_output_classes_resolved.json
- mcp_analysis_output_synced.json
- mcp_analysis_output_with_syscalls.json
- mcp_analysis_output_final.json

6. ML Refinement Pipeline

An ML-driven refinement step was required by the assignment. Due to hardware limitations, Qwen and Llama models could not be used. A smaller, faster model (GPT■2) was selected.

ml_refiner.py performs:

- Refinement of tool descriptions
- Confidence scoring (0–1)
- Metadata enhancement

The final ML-refined report:
mcp_analysis_output_final_ml.json

7. Challenges Faced & Solutions

Hardware Limitations

The system (i3 CPU, 4GB RAM) struggled with large models. **Solution:** switch from Qwen 0.5B to GPT■2.

API Limits

External LLMs had strict rate and size limits.
Solution: independent local static analysis pipeline.

TypeScript Repositories With No Tools

Some TS repos contained no MCP logic at all.
Solution: clarify partial support and rely on Python SDK for full extraction.

Long ML Runtime

Large refinement loops froze the system.
Solution: smaller model, reduced tokens (<60), optimized prompts.

8. Final Outputs Delivered

The final project includes:
- Full static analysis pipeline
- MCP tool extraction
- Multi-language analyzer framework
- ML refinement using GPT■2
- Syscall scanner
- Final JSON reports
- Readable Markdown report
- This detailed PDF report

Submitted Files:
- mcp_analysis_output_final.json
- mcp_analysis_output_final_ml.json
- mcp_final_report_readable.md
- All analyzer scripts (.py)
- Final ZIP submission

9. Conclusion

The MCP Repository Analyzer successfully fulfills all assignment objectives. It offers:

- Multi-language architecture
- Full Python MCP extraction
- ML-driven refinement
- Structured JSON output
- Real-world applicability

The project represents a complete end-to-end solution integrating static analysis, ML modeling, and repository intelligence.