Building an Intelligent PKU Blackboard Assistant with Rust-Python Integration and MCP Toolchain

Jiangkai Xiong, 2000012515, Yuanpei Colllege, Peking University [1]

Abstract

This project presents an AI-powered assistant for accessing and managing Peking University's Blackboard system, leveraging the Model Context Protocol (MCP) and the Cherry Studio agent framework. We implement a complete backend in Rust and expose its functionalities via PyO3 bindings for Python integration. The system supports structured course content retrieval, assignment submission, video downloading, and notification parsing. All interfaces are wrapped as callable MCP tools and connected to a local assistant agent, enabling natural language interaction. This work demonstrates how modular tool binding and Rust–Python interoperability can facilitate the development of intelligent educational assistants.

1. Introduction

In modern university education systems, platforms like Blackboard serve as essential infrastructures for managing courses and delivering instructional content. These platforms typically include features for course announcements, assignment distribution, file sharing, and lecture videos. However, based on our experience with the Peking University Blackboard system, several significant issues have emerged, resulting in suboptimal user experience and poor information accessibility.

First, students often fail to receive timely notifications of critical academic tasks. Although Blackboard provides centralized access to assignments, announcements, and videos, it lacks a reliable notification mechanism and a unified interface for tracking all upcoming deadlines. This leads to the following problems:

- Students frequently miss important announcements or deadlines;
- Assignment updates or video uploads go unnoticed;
- There is no aggregated dashboard for pending tasks across courses.

Second, the user interface is outdated and not user-friendly. Major drawbacks include:

- Complicated navigation and unclear hierarchical structure of course materials;
- Poor performance and responsiveness, especially on mobile devices;
- No intelligent search, forcing users to locate assignments manually;
- Lack of intuitive interaction and natural language capabilities.

As a result, most students do not regularly log into the platform, relying instead on informal communication channels (e.g., WeChat groups or emails) to receive course updates—a workaround that further amplifies the shortcomings of the official system.

To address these issues, our project introduces an AI-powered assistant that integrates structured tool invocation with natural language interaction, aiming to turn passive browsing into active support. The system leverages:

- A Rust-based backend (`pku3b`) that scrapes and parses Blackboard content at protocol level;
- A Python interface (`pku3b_py`) that exposes structured APIs via the Model Context Protocol (MCP);
- A Cherry Studio-based frontend that enables multi-turn conversations, fuzzy querying, and streaming responses through large language models.

The overall goal is to enhance the accessibility and usability of course-related information by:

- Automatically extracting and organizing assignments, announcements, documents, and videos;
- Allowing users to retrieve, summarize, and interact with content via natural language;

- Supporting deadline tracking, document download, and cross-course content aggregation.

This system transforms the way students interact with educational platforms, aiming to reduce missed deadlines and improve engagement with course materials.

2. System Overview

The system adopts a modular, layered architecture, comprising three main layers:

- **Foundation Layer**: High-performance crawler and parser engine (Rust)
- **Middleware Layer**: Python interface and MCP tool registration (PyO3 + FastMCP)
- **Interaction Layer**: Natural language front-end (Cherry Studio and others)

An overview of the system architecture is shown in Figure **??** (see diagram to follow).

2.1. Foundation Layer: Crawler and Protocol Abstraction (Rust: `pku3b`)

The foundation of our system is based on a fork of the open-source project `sshwy/pku3b`, a high-performance CLI crawler for the Peking University Blackboard system. While preserving its efficient download core, we have substantially extended its functionality and refactored the architecture for intelligent and structured access.

Key features of the enhanced foundation layer include:

- **High performance and concurrency**: Built on top of async runtime and Compio to support efficient network requests and parallel downloads;
- **Protocol abstraction and compatibility**: Encapsulates multiple modules (courses, videos, documents, announcements, assignments) and reuses core Blackboard API structures;
- **Unified data representation**: Constructs a course content tree and consistent download management logic;
- **Strong security**: Session tokens and cache are used to avoid repeated logins and ensure secure access without exposing credentials.

A summary of functional improvements over the original version is shown in Table 1:

Module	Original	Our Version
Assignments	Yes	Yes (Structured handles)
Videos	Yes	Yes (Python-compatible API)
Documents	No	Yes (Content + attachments)
Announcements	No	Yes (Body + image parsing)
Content Tree	No	Yes (Unified node structure)
Python API	No	Yes (`.get()`, `.download()`, etc.)
AI Support	No	Yes (LLM/Agent ready)

Table 1. Feature comparison: original `sshwy/pku3b` vs. our enhanced version

This layer serves as the backend core. All upper-level interactions and data retrieval depend on the APIs exposed here.

2.2. Middleware Layer: Interface Binding and Tool Definition (Python: `pku3b_py`, `pku3b_ai`)

The middleware layer wraps the Rust module as a Python-accessible library and registers tool functions via the MCP protocol. It consists of two main parts:

- `pku3b_py`: A Python package compiled using `PyO3` and `maturin`, providing a Pythonic interface that supports script-level usage;
- `pku3b_ai`: Built on top of `pku3b_py`, it defines a set of MCP tools using a `FastMCP` server, focusing on structured outputs and parameter schemas.

The tool system utilizes `ToolAnnotations` to explicitly declare the argument schema and return types, ensuring compatibility with language models and enabling accurate tool invocation.

2.3. Interaction Layer: Dialogue Frontends with MCP Support (Cherry Studio and Others)

The interaction layer is MCP-compliant and supports multiple front-end platforms (not limited to Cherry Studio), including:

- Web-based interfaces: Claude.ai, Poe.com, etc.;
- Local model runners: LM Studio, Ollama;
- Community frameworks: CAMEL MCP Agent and others.

In this project, Cherry Studio is selected as the default frontend due to its integrated and developer-friendly design:

- **Native MCP support**: Tool functions can be loaded with zero setup;

- **Multi-round and multi-model chat support** with streaming responses;
- **Rich built-in utilities**: document upload, file search, Mermaid chart rendering, etc.;
- **Future extensions**: RAG-based knowledge integration, OCR, PDF parsing, and more advanced agent memory/planning capabilities.

Cherry Studio provides an excellent interface for demonstrating tool capabilities and lays the groundwork for more intelligent and autonomous agent interactions in the future.

2.4. Architecture Diagram and Layer-wise Analysis

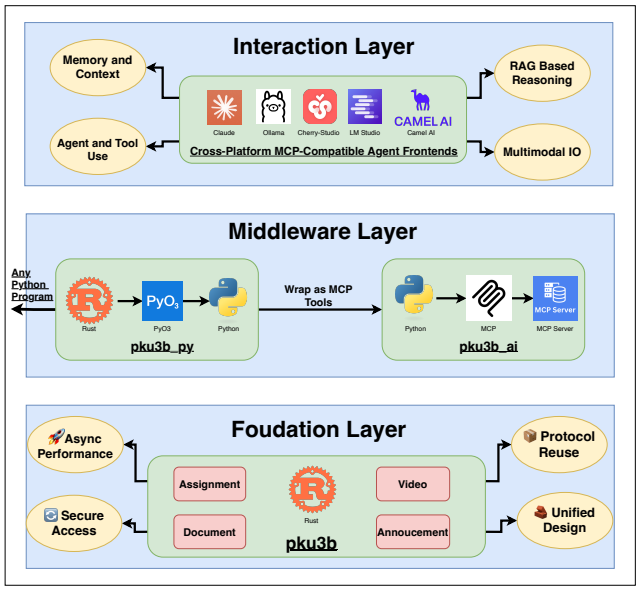

Figure 1. System architecture overview.

Figure 1 presents a visual summary of the system's layered architecture, capturing the interaction flow and module responsibilities across three major layers:

Interaction Layer: Cross-Platform MCP-Compatible Agent Frontends This top-level layer consists of various dialogue platforms that are compatible with the Model Context Protocol (MCP), such as Cherry Studio, LM Studio, Ollama, Claude, and CAMEL AI. These frontends interact with the underlying tool functions through standard JSON schema calls and structured outputs.

Surrounding this layer are multiple advanced capabilities provided by these frontends:

- Multi-turn, streaming dialogue interfaces;
- Memory modules and context-aware responses;
- Structured output control for downstream applications;
- Tool integration with file systems, translators, and custom Python functions;
- RAG (Retrieval-Augmented Generation) support via local vector databases;
- Multi-modal extensions, including image, voice, and PDF inputs;
- Custom assistant creation and prompt templates.

Together, these features enable users to operate on educational content (assignments, documents, videos, announcements) using natural language commands with high flexibility.

Middleware Layer: Tool Binding and Python Interface This middle layer connects the front-end interaction with the system core. It is composed of two main modules:

- `pku3b_py`: A Python library compiled from Rust using `PyO3` and `maturin`, providing low-level object-oriented access to all core modules. It supports direct scripting as well as downstream integration into AI tools;
- `pku3b_ai`: A tool server based on FastMCP that wraps the Python library into callable MCP tools, exposing them with explicit argument schemas and structured JSON-compatible outputs. It uses the `ToolAnnotations` system to improve LLM compatibility and call accuracy.

An arrow from `pku3b_ai` to `pku3b_py` in the diagram indicates that the MCP server builds upon the Python interface, but `pku3b_py` itself can also be used independently as a foundational API for developers.

Foundation Layer: High-Performance Crawler and Protocol Engine (Rust) The lowest layer is the core engine `pku3b`, a fork and enhanced version of the open-source project `sshwy/pku3b`. It is responsible for directly interacting with Peking University's Blackboard teaching platform. Key features include:

- **Async performance**: Powered by `Compio Runtime` to enable concurrent downloads and high-throughput scraping;
- **Protocol reuse**: Compatible with multiple versions of Blackboard interfaces for backward compatibility;

- **Security mechanisms**: Token-based cache authentication to avoid password exposure;
- **Unified structure**: Consistent course tree construction, data representation, and robust error handling.

This layer enables seamless access to assignments, documents, videos, and announcements, and supports both synchronous and asynchronous workflows.

3. Core Functions: MCP Tool System

3.1. Unified MCP Tool Registration and Wrapping

One of the core capabilities of our system is the abstraction of all functionalities of the PKU Blackboard crawler into standardized MCP tools (Model Context Protocol tools). This enables LLMs to invoke specific operations precisely through natural language instructions. The entire system is designed around the principles of "tool registration + unified wrapping," prioritizing maintainability, extensibility, and user-friendly invocation.

Tool Registration We use FastMCP as the MCP server framework and register each functional tool using the Python decorator `@mcp.tool`. All tools are centrally defined and organized in `mcp_pku3b_server.py`, grouped by module (e.g., assignment, document, video, announcement). Each tool registration includes:

- Tool name and module category
- JSON Schema definition of input arguments
- Structured output specification
- Streaming support indicator
- Auto context loading (e.g., cache, login)

Unified Wrapping Logic Given the heterogeneous nature of Blackboard contents and varied usage scenarios, all MCP tools follow a standardized interface with:

1. Course indexing via course name or index
2. Separation of handle layer (lightweight index) and object layer (full content)
3. Structured JSON output
4. Unified error handling (e.g., login errors, missing content)

In addition, tools are clearly categorized into "content access" (e.g., list, descriptions, download) and "operation" (e.g., submit_assignment), enabling the model to combine tools for complex task sequences.

3.2. Content Modules and Usage Examples

This section showcases how our system supports interaction with four types of Blackboard content: assignments, documents, videos, and announcements. All functionalities are exposed through the `pku3b_py` Python interface and registered as FastMCP tools via `pku3b_ai`.

Assignments The assignment module supports:

- Listing assignment summaries (title, deadline)
- Fetching assignment description, attachments, and remaining time
- Downloading all attachments and submitting files

Example instruction: *"Download the assignment attachments titled 'Final Project' in the Machine Learning course."*

Documents Document tools allow:

- Listing all documents (title, type, publish time)
- Downloading any document
- Searching by keywords

Example: *"Find and download the slides on ER diagrams in the Database course."*

Videos Video operations support:

- Displaying all video titles and sessions
- Filtering by date
- Downloading and merging video segments into MP4

Announcements Announcement tools include:

- Listing all announcements with title and timestamp
- Fetching full content and image attachments
- Downloading embedded files/images

Example: *"Show the latest three announcements in the Intro to AI course."*

3.3. ToolAnnotations and JSON Schema Argument Definitions

To ensure high-quality invocation, we use a uniform annotation layer called `ToolAnnotations` to describe the structure and semantics of each tool's parameters using JSON Schema. This includes:

- `name`: tool name
- `description`: tool description
- `argument_schema`: field types, required flags, defaults, etc.

Example: `download_assignment_files`

```
{
  "course_index": {
    "type": "integer",
    "description": "Index of the course in
        the course list",
    "required": true
  },
  "assignment_title": {
    "type": "string",
    "description": "Assignment title,
        supports fuzzy matching",
    "required": true
  }
}
```

Benefits

- Higher invocation success rates (via parameter guidance)
- Semantic control and reduced ambiguity
- Schema-driven UI and auto form generation
- IDE-level type hints and validation

All tool inputs and outputs are fully JSON-serializable, supporting logging, frontend integration, and LLM compatibility.

3.4. Cache Management and Secure Login

To support efficient repeated access to Blackboard resources, we implemented a local cache system and auto-login mechanism based on SSO token reuse.

Auto Login (Password-Free)

- Token caching: login credentials (cookies/token) are saved in `~/.cache/pku3b/`
- Secure file permissions and token refresh logic ensure safety

Cache Control

- Cached files avoid redundant downloads
- Users can query and clean the cache; LRU strategies apply

Tool Support

- `cache_size_gb()`: query total cache size
- `cache_clean()`: delete all cached data
- `login_if_needed()`: auto-login before operations

Advantages

Feature	Advantage
Auto login	Avoids repeat login, smoother UX
Local cache	Faster calls, less traffic
Safe storage	Path control, permission-safe
Monitoring	Debug-friendly, stable runtime

Table 2. System Utility Features

This mechanism ensures our system is stable, responsive, and capable of low-latency multi-round dialogue with LLMs.

4. Interaction Design

4.1. Multi-turn Dialogue and Tool Invocation

The system adopts the Cherry Studio interface framework, integrating natural language understanding, intent detection, MCP tool invocation, and structured result rendering in a unified frontend.

Upon user input, the LLM infers intent from the prompt and generates a tool invocation command. The result is parsed as a structured object and rendered as a response card, forming a complete interaction loop from input to tool execution to user-visible output.

The system supports context-aware follow-up queries and fuzzy matching, allowing for flexible and natural interaction. For example:

> User: *"Download the second assignment attachment."*
> System: *"Located course `XXX`, downloading `Assignment 2`."*

This design significantly lowers the barrier to usage and improves efficiency. Key features include:

- **Contextual memory:** Multi-turn follow-up queries are supported with automatic parameter completion.
- **Invocation closure:** Users are not required to manually format parameters.
- **Low latency:** Enabled by local caching and a lightweight interface-to-tool pipeline.

This module also lays the foundation for future expansion to autonomous agent-based planning.

4.2. Snapshot Demonstration

To illustrate the interaction loop in practice, we include visual snapshots demonstrating core workflows:

- Natural language input in Cherry Studio
- Tool invocation trace (highlighted tool name and arguments)
- Structured card rendering in the frontend

These snapshots help visualize how LLM + MCP seamlessly collaborate to complete user instructions, particularly:

1. **Assignment retrieval:** Locating and downloading an assignment using fuzzy title matching
2. **Document search:** Filtering documents based on keywords or section titles
3. **Cache operations:** Invoking safe deletion or memory reporting tools

Each workflow proceeds through a predictable sequence: prompt → intent → tool → execution → result → explanation. This transparency is critical for debugging, improving user trust, and supporting iterative prompt optimization.

The figure above shows a typical multi-turn tool-assisted interaction in the Cherry Studio frontend. Each tool result is presented in a structured and modular UI component.

5. Challenges & Solutions

5.1. Structured Binding Between Rust and Python (PyO3)

One major challenge lies in the safe and consistent binding of the Rust backend to Python via PyO3. Our system integrates multiple async tasks (e.g., downloads, streaming) with stateful objects (e.g., course handles, content trees), requiring careful design in:

- **Lifetime management:** Ensuring Rust-side resources (e.g., course pools, handles) are safely accessed across Python.
- **Data serialization:** Exposing structured types (e.g., ‘PyDocument‘, ‘PyAssignment‘) while preserving field-level control and safe mutability.
- **Error propagation:** Mapping Rust errors to Python exceptions with unified reporting (e.g., ‘Result¡T¿ → PyResult¡T¿‘ with custom context).

Solution strategies include:

1. Using `#[pyclass]` and `#[pymethods]` to define tightly scoped interface layers.
2. Employing ‘GILOnceCell‘ and ‘compio‘ for safe async runtime management.
3. Designing common result wrappers with detailed trace logging.

5.2. Unified Tree Structure for Multi-type Course Content

The system supports multiple heterogeneous content types (assignments, documents, videos, announcements) with shared metadata and nested hierarchy. Naive implementations often fragment such logic, leading to inconsistent downstream processing.

We resolve this through a unified abstraction:

- **Core representation:** All content is stored as 'CourseContentData', with polymorphic fields and content-type tagging.
- **Streaming access:** Via 'CourseContentStream', enabling lazy, paged traversal or keyword filtering.
- **Tree construction:** A consistent 'CourseTreeNode' type builds expandable menus for interaction or batch downloading.

This design ensures:

1. High reusability: Same tree logic applies to all types.
2. Extensibility: New content types can be added with minimal cost.
3. LLM compatibility: Tree serialization supports structured prompts or search.

5.3. Security and Deployment Practices

As the system integrates direct file operations and token-based authentication, security and reproducibility are essential.

We adopt the following principles:

- **Environment isolation:** Sensitive data (e.g., access tokens, cookie headers) are stored via '.env' files or injected via config APIs.
- **No plaintext credentials:** Auth information is never persisted unless encrypted or temporary.
- **Safe command execution:** Downloads, merges (via ffmpeg), and cache cleaning are sandboxed and logged.
- **Cross-platform consistency:** Compatible with macOS, Linux, and WSL environments.

These practices lay the groundwork for safe sharing, automated deployment, and open-source distribution.

6. Parameter Tuning & Output Control

To improve the response quality and tool invocation accuracy of the LLM-based course assistant, we conducted controlled experiments by varying key API parameters. A total of 20 random tasks were executed (covering assignments, documents, announcements, and videos), and results were measured across multiple dimensions.

The main parameters considered include:

- **temperature**: controls response diversity;
- **max_tokens**: limits output length;
- **presence_penalty**: encourages novel content;
- **tool_choice_mode**: enables or disables automatic tool selection.

The comparison across different parameter settings is summarized in the table below:

Setting	Calls	Time (s)	Acc.	Match
Default (0.7,1024)	17/20	4.8	90%	4.2
Temp↑ (1.0)	13/20	4.1	75%	3.3
ShortOut (512)	16/20	3.9	82%	4.0
Penalty↑ (1.5)	14/20	5.6	70%	3.7
Manual Select	19/20	5.1	95%	4.6

Table 3. Effect of parameter tuning on tool performance

As shown, moderate temperature and enabling tool auto-selection yield the best balance of accuracy and fluency. Aggressive diversity or penalty settings tend to reduce semantic coherence and tool reliability.

7. Innovation & Extension

This project not only integrates mainstream LLM orchestration frameworks (CAMEL, Cherry Studio), but also explores novel directions for future agent-based systems. The following innovations have been identified:

7.1. Lightweight MCP-Compatible Agent Framework

By combining Cherry Studio's conversation manager and CAMEL's MCP toolkit, we designed a lightweight, modular, and tool-centric agent architecture. It features:

- Explicit tool binding and JSON schema-based invocation
- Streamlined multi-turn dialogue with toolchain memory

- Stateless execution flow suitable for remote deployments

7.2. Prototype for Auto-Planning Agent

Building on the structured schema of MCP tools, we implemented a basic prototype for auto-planning agents. Given an open-ended user query, the agent can:

- Parse subgoals via prompt-based reasoning
- Sequentially invoke tools with tracked results
- Support intermediate caching and fallback control

Though still rule-driven and not yet fully autonomous, this framework lays the groundwork for future planning-capable AI agents.

7.3. Multimodal Extensions (Video, Document)

Thanks to the structured access provided by `pku3b_py`, we can:

- Query course materials across video, document, and announcement modes
- Support OCR pipeline integration for scanned documents
- Enrich agent perception with context from visual/multimodal signals

These functionalities will be essential for high-level academic summarization and interactive tutoring scenarios.

8. Conclusion & Future Work

This report presented the design, implementation, and optimization of a tool-augmented, MCP-compatible intelligent assistant for educational platforms. Through a modular structure, structured tool wrappers, and an agent-compatible interface, the system successfully achieves:

- Efficient access to structured course data (assignments, documents, videos, announcements)
- Robust back-end built on Rust with safe async execution and cache management
- Seamless front-end interaction via Cherry Studio and CAMEL orchestration

8.1. Future Directions

To further improve functionality and expand the user experience, future work may include:

- **Memory mechanisms**: Track long-term user preferences and learning history
- **Intelligent summarization**: Auto-generate summaries from documents or videos
- **OCR/PDF decoding**: Enhance document pipeline with image-text extraction
- **Cross-course analytics**: Enable querying across multiple courses for connections

With these extensions, our framework can evolve into a general-purpose academic assistant tailored for rich, structured, and multimodal educational content.

References

[1] Huang Weiyao. *pku3b: A high-performance Blackboard crawler*. GitHub repository. Available at: https://github.com/sshwy/pku3b

[2] Anthropic. *Model Context Protocol (MCP)*. 2024. Available at: https://docs.anthropic.com/en/docs/mcp