Research Report: Open-Source Solutions and Integration Patterns for a Microsoft Teams Planner Natural Language Interface

DATE: 2025-10-06

REPORT OBJECTIVE: Research and evaluate existing open-source solutions for a Microsoft Teams Planner natural language interface, and provide comprehensive guidance on OpenWebUI and Model Context Protocol (MCP) integration patterns. This report is intended for a development team planning to build a conversational AI system for managing Microsoft Teams Planner. It includes evaluation matrices, architectural patterns, GitHub repository assessments, technical specifications, and actionable recommendations for reducing development time through forking or modifying existing solutions.

Executive Summary

This report presents a comprehensive analysis of the open-source landscape for developing a natural language interface for Microsoft Teams Planner. The primary finding is that no turn-key, open-source solution currently exists that directly provides conversational AI capabilities for Planner. The existing open-source projects related to Microsoft Planner are predominantly focused on command-line interfaces, data extraction scripts, and specific automation workflows rather than interactive, natural language-driven task management.

Despite the absence of a ready-made solution, the component technologies required to build such a system are mature, well-documented, and largely open-source. The recommended approach is to construct a custom solution by integrating several powerful platforms: Microsoft Teams as the user-facing client, OpenWebUI as the conversational AI hub, a custom-built backend tool server conforming to the Model Context Protocol (MCP) standard, and the Microsoft Graph API as the definitive interface to Planner data.

The proposed architecture leverages OpenWebUI for its robust user interface, model management, and extensible plugin system. However, as OpenWebUI does not natively support MCP, integration must be achieved through an intermediary proxy server, MCPO (MCP-to-OpenAPI), which translates MCP tool calls into a standard OpenAPI specification that OpenWebUI can consume. The core of the custom development effort lies in creating a dedicated Planner MCP Server. This server will encapsulate all business logic for interacting with the Microsoft Graph API, translating natural language intents into specific API requests for creating, reading, updating, and deleting Planner tasks, buckets, and plans.

This report provides detailed architectural blueprints, evaluates adaptable open-source chatbot frameworks that can serve as foundational codebases, and offers a deep dive into the technical specifications of the Microsoft Graph API for Planner, including its capabilities and critical limitations such as rate throttling and the lack of real-time webhook notifications. By following the phased development approach and integration patterns outlined herein, the development team can efficiently construct a powerful, scalable, and intuitive conversational AI for Microsoft Teams Planner.

1.0 Introduction

The modern collaborative workspace demands efficiency and reduced context-switching. Integrating task management directly into communication platforms like Microsoft Teams through a natural language interface represents a significant step toward this goal. The objective of this research is to provide a development team with a clear and actionable roadmap for building such a system for Microsoft Planner. This involves a thorough evaluation of the existing open-source ecosystem to identify potential accelerators, followed by detailed guidance on architecting a solution using state-of-the-art open-source conversational AI frameworks.

This report addresses the challenge by dissecting the problem into its core technological components. We begin by assessing the landscape of open-source projects related to Microsoft Planner and general-purpose conversational AI. We then delve into the foundational technologies that will underpin the custom solution. A significant portion of the analysis is dedicated to the Microsoft Graph API, the essential gateway to all Planner data and functionality. Understanding its data model, authentication mechanisms, and operational constraints is paramount for successful implementation.

Furthermore, this report provides specific, in-depth integration patterns for two pivotal open-source technologies: **OpenWebUI** and the **Model Context Protocol (MCP)**. OpenWebUI is a feature-rich, self-hosted web interface for large language models (LLMs) that offers an excellent foundation for the conversational hub. MCP is an emerging open standard designed to standardize the way AI agents interact with external tools and data sources, promoting modularity and reusability. By architecting the Planner integration as an MCP-compliant tool, the system gains long-term flexibility and interoperability. The report details how these two systems can be integrated, addressing the technical nuances of bridging OpenWebUI's OpenAPI-based plugin architecture with the MCP standard. The final sections synthesize these findings into actionable recommendations, a proposed development strategy, and an analysis of key technical decisions, empowering the development team to proceed with confidence and clarity.

2.0 Assessment of Existing Open-Source Solutions

A thorough review of the open-source landscape reveals a critical gap: there are no mature, publicly available conversational AI systems specifically designed for Microsoft Planner that can be forked and deployed with minimal modification. Existing repositories tagged with "microsoft-planner" on platforms like GitHub are primarily focused on auxiliary functions. These include command-line tools for bulk management (e.g., `pnp/cli-microsoft365`), scripts for data extraction and reporting (e.g., `ben-howard/Planner-PowerBI-with-PowerAutomate`), and specific automation workflows. While useful in their own right, they do not provide the interactive, natural language processing (NLP) foundation required for a chatbot interface.

Consequently, the most viable strategy is to adapt a general-purpose open-source conversational AI or chatbot framework. These projects provide the front-end components, chat logic, and LLM integration hooks, upon which a custom Planner backend can be built. The following evaluation assesses several prominent and adaptable repositories that could serve as a starting point.

2.1 Evaluation Matrix of Adaptable Repositories

Repository Name	GitHub Stars (Approx.)	Primary Language/ Framework	Key Features	Adaptability for Planner	Forking Recommendation
`mckay-wrigley/chatbot-ui`	20k+	TypeScript, Next.js, React	Clean UI, supports multiple models (OpenAI, Anthropic, etc.), RAG capabilities, prompt library, Supabase/local storage.	High	High
`vercel/ai-chatbot`	7k+	TypeScript, Next.js, React	Vercel AI SDK integration, streaming support, simple and hackable codebase, function calling examples.	High	High
`microsoft/teams-ai`	N/A	TypeScript, C#	Official Microsoft library for building Teams bots, handles Teams-specific auth and activity processing, includes AI primitives.	Medium	N/A (Library)
`ey-altoledano/claude-task-master`	1k+	Python	AI-powered task management system, uses MCP for tool integration, designed for AI editors but	High (as a reference)	Medium

Repository Name	GitHub Stars (Approx.)	Primary Language/ Framework	Key Features	Adaptability for Planner	Forking Recommendation
			logic is adaptable.		

2.2 Detailed Repository Assessment

`mckaywrigley/chatbot-ui` : This is one of the most popular open-source web interfaces for LLMs. Its primary strength lies in its polished and feature-rich user interface, which is ready for deployment. It supports a wide range of models and provides essential features like conversation history, prompt management, and file uploads for Retrieval-Augmented Generation (RAG). For the development team, forking this repository would significantly accelerate front-end development. However, it does not have a built-in, extensible tool or plugin system comparable to OpenWebUI. Integrating the Planner functionality would require building a custom backend API that this front-end would call, and potentially modifying the UI to accommodate task-specific interactions. Its high adaptability stems from its clean codebase and focus on the user interface, leaving backend integration flexible.

`vercel/ai-chatbot` : Developed by Vercel, this repository serves as an excellent template for building AI applications using the Vercel AI SDK. Its codebase is intentionally minimal and easy to understand, making it an ideal candidate for forking and heavy customization. It provides clear examples of modern AI development patterns, including response streaming and function calling. A development team could use this as a skeleton, building out both the UI and the backend logic for Planner integration. Its adaptability is high due to its simplicity and focus on providing a solid, modern foundation. The primary effort would be in building the Planner-specific logic and potentially a more complex UI for displaying task data.

`microsoft/teams-ai` : This is not a standalone application but a library or SDK provided by Microsoft specifically for building intelligent applications within Microsoft Teams. Its value is immense for the final stage of the project: embedding the conversational agent into Teams. The library abstracts away the complexities of the Microsoft Bot Framework, simplifies authentication within the Teams context, and provides AI-centric features like prompt management and stateful conversation modeling. While it cannot be “forked” as a complete solution, it is a critical component for the frontend integration. The recommendation is to use this library to build the Teams bot that will communicate with the backend conversational hub (such as a system built on OpenWebUI or a fork of `chatbot-ui`).

`eyaltoledano/claude-task-master` : This repository is particularly noteworthy because it is a conversational task management system that explicitly uses the Model Context Protocol (MCP). While it is not designed for Planner and is geared towards software development tasks within AI-assisted editors, its architecture is highly relevant. It serves as a practical, open-source example of how to structure an MCP server in Python to handle task-related commands. The development team should study this repository closely as a reference for building the custom Planner MCP Server. Forking it directly might be less efficient than starting fresh, but its patterns for parsing requirements, planning steps, and executing actions via tools are directly applicable.

3.0 Technical Foundation: Microsoft Graph API for Planner

The entire backend of the proposed system hinges on the Microsoft Graph API, which is the single, unified endpoint for accessing data in Microsoft 365, including Microsoft Planner. A deep understanding of its capabilities, data structures, and limitations is non-negotiable for the development team.

3.1 API Capabilities and Data Model

The Planner API exposes a hierarchical and relational data model that logically represents the user's experience in the Planner application. The primary entities are **Plans**, which are the top-level containers for tasks and must be associated with a Microsoft 365 Group. Within each plan, tasks are organized into **Buckets**, which act as customizable columns on a task board. The core unit of work is the **Task**, a rich object containing properties like title, due date, priority, and completion status. To optimize performance, the API separates core task properties from less frequently accessed ones; detailed information such as the task **Description**, **Checklist** items, and **References** (attachments) are stored in a separate `plannerTaskDetails` object that must be explicitly requested. Tasks can be assigned to multiple users within the parent group, and categorized using a set of predefined colored labels known as **Applied Categories**. This granular data model allows a natural language interface to perform highly specific actions, such as "Add a checklist item 'Confirm venue' to the 'Event planning' task and assign it to Alex."

3.2 Authentication and Authorization

Programmatic access to Planner data is secured by the Microsoft identity platform, which uses the OAuth 2.0 protocol. The application must be registered in Microsoft Entra ID (formerly Azure Active Directory) and granted specific permissions, or scopes, to access Planner data. For a system that will perform actions on behalf of a user, it must request **delegated permissions**. The essential scope for a fully functional interface is `Tasks.ReadWrite`, which allows the application to read, create, update, and delete plans and tasks that the signed-in user has permission to access. The application will use a standard OAuth 2.0 flow, such as the Authorization Code Flow, to obtain an access token for the user, which must be included in the `Authorization` header of every API request. This ensures that the application can never access data beyond the user's own permissions.

3.3 Operational Logic and Endpoints

The API provides a complete set of RESTful endpoints for performing Create, Read, Update, and Delete (CRUD) operations on all Planner entities. Operations are executed using standard HTTP methods (`GET`, `POST`, `PATCH`, `DELETE`). A critical mechanism for maintaining data integrity in a collaborative environment is the use of **ETags** for optimistic concurrency control. Every Planner resource returned by the API includes an `@odata.etag` property. To update or delete a resource, the application must provide this ETag in an `If-Match` header. If the resource has been modified by another user since it was last fetched, the ETags will not match, and the API will reject the update with a `412 Precondition Failed` error. The application must then re-fetch the resource, resolve the conflict, and retry the operation. This prevents accidental overwrites and is a mandatory feature for a robust implementation.

3.4 Critical Limitations and Considerations

While powerful, the Graph API for Planner has two significant limitations that directly impact the architecture of a real-time conversational system. First, it is subject to **strict rate limits and throttling**. Applications that make too many requests in a short period will receive `429 Too Many Requests` errors. The application logic must be designed to handle this gracefully, typically by implementing an exponential backoff strategy and respecting the `Retry-After` header in the API response. Optimizing calls using batching is essential.

Second, and more critically, Planner resources are **not supported by the Microsoft Graph change notifications (webhooks) system**. This means the application cannot subscribe to real-time updates. To keep its state synchronized with Planner, the application must implement a polling mechanism. The most efficient way to do this is by using **delta queries**, which allow the application to request only the resources that have changed since the last poll. This is a fundamental architectural constraint; the system will not be instantly aware of changes made outside of its own interface and will have a latency determined by its polling interval.

4.0 Recommended Architecture and Integration Patterns

Based on the analysis of available technologies, the most robust and future-proof approach is to build a modular system that leverages the strengths of specialized open-source components. The recommended architecture connects a user in Microsoft Teams to Planner via a conversational hub (OpenWebUI) and a standardized tool server (MCP).

4.1 High-Level System Architecture

The proposed end-to-end data flow follows a clear, decoupled path, ensuring each component is responsible for a specific part of the process.

1. **User Interface (Microsoft Teams):** The user interacts with a custom bot built using the `microsoft/teams-ai` library. This bot is responsible for capturing user messages and managing the Teams-specific user experience.
2. **Conversational AI Hub (OpenWebUI):** The Teams bot forwards the user's message to the OpenWebUI backend. OpenWebUI manages the conversation history, orchestrates the interaction with the selected LLM, and handles the invocation of registered tools.
3. **Tool Integration Layer (MCPO Proxy):** When the LLM determines that a Planner action is needed, OpenWebUI calls the corresponding tool. Because OpenWebUI uses an OpenAPI specification for tools, and our backend will use MCP, an intermediary is required. The **MCPO (MCP-to-OpenAPI)** proxy server receives the OpenAPI request from OpenWebUI and translates it into a standard MCP request.
4. **Backend Tool Server (Custom Planner MCP Server):** The MCPO proxy forwards the MCP request to the custom-built Planner MCP Server. This server, likely written in Python, contains the core business logic. It interprets the tool call (e.g., `create_task`), constructs the appropriate Microsoft Graph API request, handles authentication, and executes the call.
5. **API Layer (Microsoft Graph API):** The Planner MCP Server communicates directly with the Microsoft Graph API to perform the requested action in Planner.
6. **Response Flow:** The response travels back through the same chain: Graph API to the MCP Server, which formulates an MCP response. This is sent to the MCPO proxy, translated back to an OpenAPI response for OpenWebUI, which then generates a natural language reply for the user, delivered via the Teams bot.

4.2 OpenWebUI Integration Pattern

OpenWebUI's extensibility is centered around its plugin architecture, particularly **Tools**. A custom tool for Planner would be developed as a Python script. This script would define a `Tools` class containing methods that map directly to user-level actions, such as `create_task(title: str, plan_name: str)`, `list_my_tasks()`, or `complete_task(task_name: str)`. Each method must include a detailed docstring and type hints, as this metadata is used by the LLM to understand what the tool does and how to call it correctly.

For security and configuration, the tool can use **Valves**. A Valve is a configurable setting, managed through the OpenWebUI admin panel. This is the ideal place to store non-sensitive configuration, while sensitive credentials, such as the client secret for Graph API authentication, should be managed through secure environment variables on the server hosting the backend components. The OpenWebUI tool itself would not contain the Graph API logic; instead, its methods would be simple wrappers that make HTTP requests to the MCPO proxy endpoint.

4.3 Model Context Protocol (MCP) Integration Pattern

The Model Context Protocol (MCP) provides a standardized, language-agnostic way to build tools for AI agents. By building our backend as an MCP server, we ensure it is reusable and can be connected to other MCP-compatible hosts in the future.

The core development task is to create the **Custom Planner MCP Server**. Using a Python SDK like `FastMCP`, developers can quickly define a server and its capabilities. The "tools" exposed by this server are the functions that directly interact with the Microsoft Graph API. For example, an MCP tool defined as `planner.create_task` would contain the Python code to authenticate with Microsoft Graph, construct the JSON payload for a new task, and send a `POST` request to the `/v1.0/planner/tasks` endpoint. This server must also contain all the necessary logic for handling the OAuth 2.0 token lifecycle, managing ETags for updates, and implementing retry logic for throttling.

The **MCPO Proxy Architecture** is the critical bridge in this design. The MCPO server is a lightweight application that is configured to launch and communicate with one or more MCP servers. It introspects the tools available on the MCP server and automatically generates a corresponding OpenAPI (Swagger) specification. This specification can then be exposed via an HTTP endpoint. In OpenWebUI, you would register a new tool by pointing it to the OpenAPI URL provided by MCPO. This elegant pattern allows OpenWebUI to communicate with any MCP server without requiring native support, fully decoupling the conversational front-end from the tool backend.

5.0 Actionable Recommendations for Development

To translate the proposed architecture into a successful project, a structured, phased approach is recommended, focusing on building and testing components incrementally.

5.1 Phased Development Approach

Phase 1: Backend First - The Planner MCP Server. The highest-risk and most complex part of the system is the interaction with the Microsoft Graph API. Therefore, development should begin here. The team should build the custom Planner MCP Server in Python, focusing exclusively on creating a robust wrapper around the Graph API. Key deliverables for this phase include a fully implemented OAuth 2.0 authentication flow, functions for all required CRUD operations on Planner tasks and buckets, and comprehensive error handling for API throttling and ETag-based concurrency conflicts. This server should be testable in isolation using a simple MCP client or inspector tool.

Phase 2: Integration and Conversational Logic. Once the backend server is stable, the team should set up the integration layer. This involves deploying OpenWebUI and the MCPO proxy server. The Planner MCP Server is then connected to MCPO, and the resulting OpenAPI endpoint is registered as a tool within OpenWebUI. The focus of this phase is on testing the end-to-end communication flow and refining the prompts and tool definitions to ensure the LLM can reliably translate natural language requests into the correct tool calls.

Phase 3: Frontend User Experience. The final phase involves building the Microsoft Teams application. Using the `microsoft/teams-ai` SDK, the team will create a bot that serves as the user's entry point. This bot will handle Teams-specific authentication and then act as a client to the OpenWebUI backend, passing messages back and forth. The work in this phase is primarily focused on UI/UX, ensuring a seamless and intuitive experience within the Teams environment.

5.2 Reducing Development Time: Forking and Reuse Strategy

While no single repository can be forked as a complete solution, strategic reuse of existing open-source code can significantly accelerate development. The primary recommendation is to **not** fork a monolithic application. Instead, the team should leverage libraries and reference architectures.

For the backend, the team should use Microsoft's official **Microsoft Graph SDK for Python**. This library handles much of the boilerplate for making API requests, managing authentication, and parsing responses, allowing developers to focus on the business logic. The `eyaltoledano/claude-task-master` repository should be used as a primary **reference architecture** for structuring the Python-based MCP server, adopting its patterns for defining tools and managing state.

For the conversational hub, **installing and configuring OpenWebUI** is far more efficient than forking and building a custom UI from scratch like `chatbot-ui`. OpenWebUI provides a production-ready platform out of the box. The development effort should be focused on creating the custom tool plugin, not on reinventing the entire chat interface.

5.3 Key Technical Decisions and Trade-offs

The development team will need to address several key decisions early in the project. First is the strategy for **data synchronization**. Given the lack of webhooks, the team must decide on an appropriate polling interval for the delta query mechanism. A shorter interval provides fresher data but increases the load on the Graph API and the risk of throttling. This trade-off must be balanced based on user requirements.

Second is **state management**. The conversational interface needs to maintain context, such as knowing which plan or task the user is currently discussing. This state can be managed within OpenWebUI's conversation history or handled more explicitly within the Planner MCP server. A hybrid approach is likely best, where the LLM uses conversation history for short-term context, and the backend can handle more persistent user-level defaults.

Finally, the **choice of LLM** is critical. The system relies on the model's ability to perform function calling (or tool use). The team should select a model known for its strong performance in this area, as this will directly impact the reliability and accuracy of the natural language interface. Using OpenWebUI's Native function calling mode with a capable model like GPT-4o is recommended for the best performance.

6.0 Conclusion

The development of a natural language interface for Microsoft Teams Planner is a highly feasible project that can deliver significant value by streamlining task management workflows. While the open-source ecosystem does not offer a pre-built solution, it provides a rich set of mature, powerful, and interoperable components from which a sophisticated system can be assembled. The architectural path is clear: a custom-built backend, designed as a standard MCP server wrapping the Microsoft Graph API, connected to the OpenWebUI conversational hub via the MCPO proxy, and surfaced to the user through a dedicated Microsoft Teams bot.

The success of this project will depend on a methodical, phased development approach that prioritizes the creation of a robust and resilient backend. By leveraging existing SDKs, reference architectures, and best-in-class open-source platforms like OpenWebUI, the development team can mitigate risks and focus its efforts on the unique logic of the Planner integration. By carefully navigating the known constraints of the Microsoft Graph API, particularly around rate limiting and data synchronization, the team can build a powerful and intuitive tool that seamlessly integrates into the daily collaboration patterns of its users.

References

Announcing the GitHub integration with Microsoft Teams - GitHub Blog (https://github.blog/news-insights/product-news/announcing-the-github-integration-with-microsoft-teams/)
Build a basic AI chatbot in Teams - Microsoft Teams (https://learn.microsoft.com/en-us/microsoftteams/platform/toolkit/build-a-basic-ai-chatbot-in-teams)
Building a basic MCP server with Python - Medium (https://medium.com/data-engineering-with-dremio/building-a-basic-mcp-server-with-python-4c34c41031ed)
Building an MCP server as an API developer - Medium (https://heeki.medium.com/building-an-mcp-server-as-an-api-developer-cfc162d06a83)
Building custom MCP tool and integrate it to your self-hosted LLM through OpenWebUI (Part 3) - Medium (https://juniarto-samsudin.medium.com/building-custom-mcp-tool-and-integrate-it-to-your-self-hosted-llm-through-openwebui-part-3-3268c4fcac6e)
claude-task-master - GitHub (https://github.com/eyaltoledano/claude-task-master)
Development - OpenWebUI Docs (https://docs.openwebui.com/getting-started/advanced-topics/development/)
Events - OpenWebUI Docs (https://docs.openwebui.com/features/plugin/events/)
Examples - Model Context Protocol (https://modelcontextprotocol.io/examples)
Features - OpenWebUI Docs (https://docs.openwebui.com/features/)
Functions - OpenWebUI Docs (https://docs.openwebui.com/features/plugin/functions/)
Getting Started - OpenWebUI Docs (https://docs.openwebui.com/getting-started/)
GitHub and Microsoft Planner Integration - Make.com (https://www.make.com/en/integrations/github/microsoft-planner)
GitHub Integration with Microsoft Teams - GitHub (https://github.com/integrations/microsoft-teams)
GitHub Topic: ai-chatbot - GitHub (https://github.com/topics/ai-chatbot)
GitHub Topic: conversational-ai - GitHub (https://github.com/topics/conversational-ai)
GitHub Topic: microsoft-planner - GitHub (https://github.com/topics/microsoft-planner)
GitHub Topic: task-management - GitHub (https://github.com/topics/task-management)
GitHub Topic: task-management-app - GitHub (https://github.com/topics/task-management-app)
GitHub Topic: task-oriented-dialogue - GitHub (httpscom/topics/task-oriented-dialogue)
How to build an AI Chatbot with Redis, Python, and GPT - freeCodeCamp (https://www.freecodecamp.org/news/how-to-build-an-ai-chatbot-with-redis-python-and-gpt/)
How to use Tools with Open WebUI - Medium (https://medium.com/write-a-catalyst/how-to-use-tools-with-open-webui-9725db2724bb)
Integrate GitHub with Microsoft Planner - Appy Pie Automate (https://www.appypieautomate.ai/integrate/apps/github/integrations/microsoft-planner)
Introducing the Azure MCP Server - Azure SDK Blog (https://devblogs.microsoft.com/azure-sdk/introducing-the-azure-mcp-server/)
Live-Chatbot-for-Final-Year-Project - GitHub (https://github.com/Vatshayan/Live-Chatbot-for-Final-Year-Project)
MCP integration into OpenWebUI - Reddit (https://www.reddit.com/r/OpenWebUI/comments/1jaidh4/

mcp_integration_into_openwebui/)
MCP Pipe - OpenWebUI Community (https://openwebui.com/f/haervwe/mcp_pipe)
MCP server step-by-step guide to building from scratch - Composio DevBlog (https://composio.dev/blog/mcp-server-step-by-step-guide-to-building-from-scrtch)
MCPO: Supercharge Open WebUI with MCP Tools - Medium (https://mychen76.medium.com/mcpo-supercharge-open-webui-with-mcp-tools-4ee55024c371)
Microsoft Graph API Overview - Microsoft Learn (https://learn.microsoft.com/en-us/graph/overview)
Microsoft Graph Dev Center - Microsoft Developer (https://developer.microsoft.com/en-us/graph)
Microsoft Graph Documentation - GitHub (https://github.com/microsoftgraph/microsoft-graph-docs-contrib/blob/main/concepts/planner-concept-overview.md)
Microsoft Graph GitHub Repositories - GitHub (https://github.com/microsoftgraph)
Microsoft Graph REST API - Microsoft Developer (https://developer.microsoft.com/en-us/graph/rest-api)
Microsoft Planner Tag on Microsoft 365 Developer Blog - Microsoft (https://devblogs.microsoft.com/microsoft365dev/tag/microsoft-planner/)
Model Context Protocol - Anthropic (https://www.anthropic.com/news/model-context-protocol)
Model Context Protocol - A Complete Tutorial - Medium (https://medium.com/@nimritakoul01/the-model-context-protocol-mcp-a-complete-tutorial-a3abe8a7f4ef)
Model Context Protocol - OpenCV Blog (https://opencv.org/blog/model-context-protocol/)
Model Context Protocol - seangoedecke.com (https://www.seangoedecke.com/model-context-protocol/)
Model Context Protocol (MCP) Tutorial - DataCamp (https://www.datacamp.com/tutorial/mcp-model-context-protocol)
Model Context Protocol (MCP) Tutorial - DigitalOcean (https://www.digitalocean.com/community/tutorials/model-context-protocol)
Model Context Protocol (MCP) Tutorial: Build Your First MCP Server in 6 Steps - Towards Data Science (https://towardsdatascience.com/model-context-protocol-mcp-tutorial-build-your-first-mcp-server-in-6-steps/)
Model Context Protocol GitHub Organization - GitHub (https://github.com/modelcontextprotocol)
Model Context Protocol Specification - modelcontextprotocol.io (https://modelcontextprotocol.io/specification/2025-06-18)
Model Context Protocol/servers - GitHub (https://github.com/modelcontextprotocol/servers)
Obsidian BMO Chatbot - AIBase (https://www.aibase.com/repos/project/obsidian-bmo-chatbot)
Open WebUI Tool Creator - OpenWebUI Community (https://openwebui.com/m/mhwhgm/open-web-ui-tool-creator)
OpenAPI Servers for Open WebUI - GitHub Discussions (https://github.com/open-webui/openapi-servers/discussions/58)
OpenAPI Servers: MCP - OpenWebUI Docs (https://docs.openwebui.com/openapi-servers/mcp/)
OpenAPI Servers: Open WebUI - OpenWebUI Docs (https://docs.openwebui.com/openapi-servers/open-webui/)
open-webui/mcpo - GitHub (https://github.com/open-webui/mcpo)
open-webui/open-webui - GitHub (https://github.com/open-webui/open-webui)
open-webui/open-webui/discussions/3134 - GitHub (https://github.com/open-webui/open-webui/discussions/3134)
open-webui/open-webui/discussions/7363 - GitHub (https://github.com/open-webui/open-webui/discussions/7363)
Optimize Open WebUI: Three Practical Extensions for a Better User Experience - Medium (https://medium.com/pythoneers/optimize-open-webui-three-practical-extensions-for-a-better-user-experience-cbe365af60b1)
Overview of the Azure MCP Server - Microsoft Learn (https://learn.microsoft.com/en-us/azure/developer/azure-mcp-server/overview)
Plugin - OpenWebUI Docs (https://docs.openwebui.com/features/plugin/)

PM-bot.ai - AI Project Management Consultant (https://pm-bot.ai/)
PMOtto.ai - AI Project Management Assistant (https://www.pmotto.ai/)
SDKs Overview - Microsoft Graph (https://learn.microsoft.com/en-us/graph/sdks/sdks-overview)
The Conversational AI Pipeline - GitHub (https://github.com/kunan-sa/the-conversational-ai-pipeline)
To Do API Overview - Microsoft Graph (https://learn.microsoft.com/en-us/graph/todo-concept-overview)
Tools - OpenWebUI Community (https://openwebui.com/tools)
Tools - OpenWebUI Docs (https://docs.openwebui.com/features/plugin/tools/)
Tools Development - OpenWebUI Docs (https://docs.openwebui.com/features/plugin/tools/development/)
Use GitHub with Loop - Microsoft Support (https://support.microsoft.com/en-us/office/use-github-with-loop-5a4d95d5-3c59-4de8-a208-c9c8ab05a4fb)
Use the API - Microsoft Graph (https://learn.microsoft.com/en-us/graph/use-the-api)
vercel/ai-chatbot - GitHub (https://github.com/vercel/ai-chatbot)