A Comprehensive Guide to Integrating External Tools with OpenAI Apps via the Model Context Protocol

This report provides a comprehensive and insightful analysis of the process for integrating external tools, including Figma, Blender, Ollama, DrawThings, and Google services, into applications built with the OpenAI Apps SDK. It serves as a guide for developers who intend to build custom MCP (Model Context Protocol) servers to connect these tools with ChatGPT. The report delves into the architectural underpinnings of the protocol, provides detailed implementation steps for each target service, establishes a robust security framework based on modern OAuth 2.1 standards, and outlines best practices for deployment and management. The information is derived exclusively from the provided context blocks, ensuring fidelity and accuracy.

Architectural Foundations: Understanding the OpenAI Apps SDK and MCP

The integration of external tools with ChatGPT is not achieved through a monolithic platform but rather through a standardized, open protocol that enables secure, two-way communication between AI systems and any external data source or application [6,8]. This standard is the Model Context Protocol (MCP), which was adopted by OpenAI and is now the foundation of its new Apps SDK [7,16]. Understanding this architecture is the first critical step in developing powerful, integrated applications. The core principle is a client-server model where the tool developer creates an MCP server that exposes specific capabilities—such as functions to execute or data to retrieve—which are then made available to an AI agent running inside ChatGPT [15,22].

The primary components of this architecture are the Host, the Client, and the Server. The Host is the application that runs the Large Language Model (LLM) and initiates connections. For developers using the OpenAI Apps SDK, the host is ChatGPT itself [13,22]. Other notable hosts include IDEs like Visual Studio Code, Cursor, and Claude Desktop [4,5]. The Client acts as a mediator, managing the connection between a single Host and one or more MCP Servers [3,17]. Within the context of ChatGPT, this client functionality is handled internally by OpenAI's infrastructure. The Server is the crucial piece that developers build; it is the service that provides the actual tools, resources, and prompts [1,3]. Officially, OpenAI provides official SDKs for Python and TypeScript to facilitate the creation of these servers [2,22]. These servers can be deployed locally for development or remotely for public consumption, communicating with the client over various transports like HTTP with Server-Sent Events (SSE) or a local command-line interface (stdio) [8,19].

The OpenAI Apps SDK introduces two key concepts to this architecture: the "Brain" and the "Face" [22]. The "Brain" refers to the backend MCP server, which contains the application logic, handles authentication, and executes the tools defined for the app [22]. This server is responsible for exposing its capabilities to ChatGPT. The "Face" is the custom user interface, built with standard web technologies like HTML, CSS, and JavaScript, which is rendered directly within the ChatGPT conversation [16,22]. This UI is not a full website but a focused component designed for a specific task, such as displaying a preview, showing settings, or rendering a complex output generated by a tool [22]. When a user interacts with the "Face," it triggers a function call that invokes the corresponding tool on the "Brain." This separation allows for rich, interactive experiences while keeping the complex logic securely encapsulated within the MCP server. The entire system is designed to be stateless on the client side, with ChatGPT managing the overall conversation history, making it cost-efficient and scalable [37].

Building Your First MCP Server: A Step-by-Step Implementation Guide

Creating a custom MCP server involves a series of well-defined steps, from setting up your development environment to deploying your application for use in ChatGPT. The process is facilitated by official SDKs for Python and TypeScript, which abstract away much of the underlying complexity of the JSON-RPC protocol [2,16]. The following guide provides a general roadmap applicable to most tool integrations.

First, you must establish your development environment. The prerequisites typically include having Node.js 18+ with pnpm or Python 3.10+ installed [9,22]. You will also need a tunneling service like ngrok to expose your local development server to the internet, allowing the ChatGPT client to communicate with it [9,22]. After cloning the starter repository for the OpenAI Apps SDK, you can begin building your server logic [9].

The core of your MCP server is the registration of tools. Each tool represents a callable function that your external application can perform. Using the Python FastMCP SDK, you would define a tool using a decorator. For example, to create a simple tool that adds two numbers, you would write:

```
from mcp import fastmcp
import pydantic

server = fastmcp.MCP()

class AddParams(pydantic.BaseModel):
    a: float
    b: float

@server.tool(
    name="add_numbers",
    description="Add two numbers together.",
    input_schema=AddParams,
```

```
)
def add(params: AddParams):
    return {"result": params.a + params.b}
```

In this code, `@server.tool` decorates a Python function, providing metadata like its name, description, and a JSON Schema defined by the `input_schema` parameter [8 9]. The function itself takes the validated input parameters and returns a dictionary, which will be sent back to the ChatGPT client. Similar patterns exist in the TypeScript SDK using libraries like Zod for schema validation [30 35]. Once your server script is written, you can run it on a local port, such as 8000 [9]. To test it within ChatGPT, you would use ngrok to create a secure public URL (e.g., `https://abcd-123-45-67.ngrok.io`) and configure this URL in the `mcpServers` section of ChatGPT's Developer Mode settings [22 37].

For production deployment, you have several options. The server can be containerized using Docker for consistency and then deployed to cloud platforms like Google Cloud Run, AWS ECS, or Azure App Service [2]. These platforms offer features like autoscaling, which is a best practice for handling variable loads [2]. During deployment, sensitive credentials like API keys should never be hardcoded. Instead, they must be managed securely using environment variables or dedicated secret management services like AWS Secrets Manager or Azure Key Vault [37 41]. The final step is to register your live server's public URL with OpenAI through their submission portal, which will undergo a review for functionality, safety, UX, and performance before being made available to users [37].

Development Phase	Prerequisites	Key Tools & Libraries	Deployment Target
Local Development	Node.js/Python, ngrok, IDE	OpenAI Apps SDK (Python/TS), Pydantic/Zod	Local machine, exposed via ngrok
Production Deployment	Docker, CI/CD pipeline	OpenAI Apps SDK (Python/TS), FastAPI (for Python)	Cloud Platforms (GCR, AWS ECS, Azure)
Secret Management	N/A	Environment Variables, AWS Secrets Manager, Azure Key Vault, HashiCorp Vault	Cloud-based secret stores

This structured approach ensures a smooth transition from a local proof-of-concept to a robust, scalable, and secure production-ready MCP server.

Integrating Creative and Data Services: Figma, Blender, and Google

Integrating creative and data-centric tools like Figma, Blender, and Google services into the OpenAI Apps ecosystem requires understanding their respective APIs and adapting them to the MCP paradigm of exposing tools and resources. While direct examples for Blender and DrawThings are not present in the provided sources, their integration follows the same fundamental principles as other custom tools. For Figma and Google services, we can leverage existing documentation and known patterns.

Figma Integration To integrate Figma, your MCP server would act as a bridge between ChatGPT and the Figma REST API [12 26]. The first step is to register an application on the Figma Developer Portal to obtain a `client_id` and `client_secret`, which are necessary for authenticating with Figma's OAuth 2.0 endpoint [12 25]. Your MCP server would then implement the OAuth 2.0 authorization code flow. When a user wants to interact with a Figma file, the MCP server would redirect them to Figma's consent page. Upon granting permission, Figma redirects back to a `redirect_uri` specified by your server, providing an authorization code. Your server exchanges this code for an access token, which is then used to make authenticated requests to the `/v1/files/{file_key}` endpoint to fetch design data [25 26].

Within the MCP server, you would define tools that wrap these API calls. For example, a tool named `get_figma_component_preview` could take a Figma file URL as input, extract the file ID, fetch the component details from the Figma API, and return a structured response containing an image preview of the component. This structured content, especially the image data, can be returned as a base64-encoded string with a `mimeType` [20]. Real-world plugins for Figma, like the one developed for Playbook, demonstrate the feasibility of this workflow, using OAuth 2.0 with PKCE for login and custom frontends to display assets [27]. The key challenge is correctly mapping Figma's API capabilities to a set of discrete, manageable tools that an LLM can invoke predictably.

Blender and DrawThings Integration Since specific documentation for Blender and DrawThings is unavailable, we can infer their integration strategy from the general principles of MCP. Both are likely to be local applications. * Blender: An integration would likely involve exposing Blender's Python scripting engine as a local MCP server running via the `stdio` transport [8 20]. The MCP server would define tools that correspond to operations executable via Blender's bpy module. For instance, a tool could be created to generate a 3D model from text, which would translate the natural language description into Blender Python commands to build the object. This approach keeps all data and processing local, adhering to privacy principles. * DrawThings: As a closed-source macOS/iOS application, integration would be more challenging without official APIs. However, if it offers a network-accessible gRPC server for offloading tasks (a feature mentioned for 'Server Assistance'), it could be integrated similarly to how Blender might be [40]. The MCP server would communicate with this local gRPC service to perform image generation tasks. If no such API exists, integration would require reverse-engineering the application's network behavior, a method fraught with legal and technical risks.

Google Services Integration Integrating Google services is streamlined by the availability of fully managed MCP servers offered by third-party providers like Composio [8]. Services such as Google Drive, Sheets, Docs, and Gmail have pre-built connectors that can be integrated with a single line of configuration, abstracting away the complexities of OAuth and API interactions [8]. For developers who prefer to build their own server, the process is similar to Figma. Each Google API has its own OAuth 2.0 endpoints and requires enabling the specific API in the Google Cloud Console [26]. The MCP server would manage the OAuth flow, store the refresh token securely, and use it to obtain short-lived access tokens to call APIs like Google Sheets or Drive. The server would then expose these functionalities as MCP tools, allowing ChatGPT to read, write, and manipulate data within Google's ecosystem.

Connecting to Local AI: Integrating Ollama and Image Generation

Integrating local AI models, particularly those run with Ollama, presents a unique opportunity to provide private, offline-capable, and highly customizable AI-powered tools within the OpenAI Apps framework. Unlike cloud-based models, local models keep user data on-device, addressing significant privacy concerns [11]. The process involves creating an MCP server that communicates with the Ollama daemon running on the user's machine.

Ollama acts as a package manager for large language models, allowing users to download and run models like Llama 2, Mistral, or specialized vision models like LLaVA for image generation [11 21]. To integrate this with an MCP server, the server must be able to execute shell commands to interact with the `ollama` binary. The first step is to ensure Ollama is installed and running on the target machine [11 39]. For enhanced usability, some developers use GUI wrappers like Msty, which simplifies interaction with Ollama on macOS by eliminating the need for terminal commands [39].

An MCP server for Ollama would define tools that map to common Ollama commands. A primary tool would be `run_model`, which takes two parameters: the model name (e.g., `llava`) and a prompt. The server would then execute a command like `ollama run llava "generate: {your_image_prompt}"` and capture the output [21]. The response from the Ollama API, which often includes generated text and image data, would be processed and formatted into the structured content expected by the MCP protocol. For image generation, the image can be encoded as a base64 string and returned alongside any relevant text, allowing the calling application to render it [20].

The table below outlines potential tool definitions for an Ollama MCP server.

Tool Name	Description	Input Schema (JSON)	Example Use Case
`list_models`	Lists all models currently downloaded and available on the local machine.	`{}`	`{"toolCallId": "1", "toolName": "list_models"}`
`run_model`	Runs a specified model with the given prompt and returns the result.	`{"model": "string", "prompt": "string"}`	`{"toolCallId": "2", "toolName": "run_model", "args": {"model": "llava", "prompt": "Create a digital art style`

Tool Name	Description	Input Schema (JSON)	Example Use Case
			`image of a cyberpunk dragon."}}`
`pull_model`	Downloads a specified model from a registry (like Ollama Hub) to the local machine.	`{"model_name": "string"}`	`{"toolCallId": "3", "toolName": "pull_model", "args": {"model_name": "mistral"}}`
`delete_model`	Removes a specified model from the local machine to free up disk space.	`{"model_name": "string"}`	`{"toolCallId": "4", "toolName": "delete_model", "args": {"model_name": "llama2"}}`

By defining such tools, the MCP server effectively virtualizes the Ollama command-line interface, making its powerful capabilities accessible through the natural language and structured data flows of ChatGPT. This approach empowers users to leverage the full power of their local hardware for generative AI tasks without compromising data privacy.

Implementing Secure Authentication and Authorization with OAuth 2.1

Security is paramount when connecting an LLM agent to external tools, as this integration creates a potential attack surface for data exfiltration, unauthorized actions, and misuse. The Model Context Protocol addresses this by mandating a sophisticated, standardized authorization framework based on OAuth 2.1 [15 23]. This framework moves beyond insecure static API keys to a model of dynamic, scoped, and auditable access control. For developers building an MCP server, implementing this correctly is not optional; it is a core requirement for a safe and compliant application.

The heart of the modern MCP authentication flow is Dynamic Client Registration (DCR) combined with Proof Key for Code Exchange (PKCE) [17 31]. Here's how the process works: 1. Initialization: When a user first connects your app in ChatGPT, the internal MCP client (managed by OpenAI) makes a request to your server's root path. If the server is not yet registered, it responds with a `401 Unauthorized` status and a `WWW-Authenticate` header pointing to its Protected Resource Metadata discovery document, typically located at `/.well-known/oauth-protected-resource` [19 31]. 2. Dynamic Registration: The client fetches this metadata, which contains the URL of the Authorization Server's registration endpoint (e.g., `https://login.example.com/register`). The client then sends a POST request to this DCR endpoint, registering itself with your server. This request includes information like the client name and a redirect URI. In response, the Authorization Server dynamically provisions a new `client_id` and, if required by the spec, a `client_secret` [31,34]. 3. User Consent: With a registered client, the process proceeds to the authorization code grant flow. The client redirects the user to the Authorization Server's `/authorize` endpoint. This URL includes the `client_id`, requested scopes (e.g., `files:read`, `draw:write`), a randomly generated code verifier, and the redirect URI. Crucially, the server must implement a built-in consent screen where the user explicitly approves the requested permissions [19]. 4. Token Issuance: After the user consents, they are redirected back to your server's callback URL with an authorization code. Your server then exchanges this code for an access token at the Authorization Server's `/token` endpoint. This exchange uses the previously supplied code verifier to prove that the client that requested the code is the same one redeeming it. The resulting access token is short-lived and scoped strictly to the permissions granted by the user [31,45]. 5. Token Validation: Every subsequent request from the client to your MCP server must include this access token in an `Authorization: Bearer <token>` header. Your server's middleware must validate this token on every call. This involves verifying the JWT signature (using JWKS), checking the audience (`aud`) claim to ensure the token is for your server, and validating the scopes to ensure the caller has permission to execute the requested tool [19,41].

This entire flow is mandated by the June 2025 MCP specification update, which formally classifies MCP servers as OAuth 2.0 Resource Servers [15,23]. Best practices extend beyond just the flow. Developers must mitigate risks like the 'Confused Deputy Problem' by never passing along upstream tokens ('token passthrough') and instead fetching data on behalf of the user [28]. All communication must be encrypted using TLS [41]. Furthermore, identity providers like Stytch, Auth0, and Keycloak offer solutions that handle the complexities of this flow, including OIDC discovery, consent management, and token rotation, significantly simplifying implementation [19,29,44].

Advanced Security, Deployment, and Operational Best Practices

Beyond initial setup and authentication, ensuring the long-term security, reliability, and scalability of an MCP server requires adherence to a set of advanced operational best practices. The rapid adoption of MCP has highlighted inherent risks, making proactive security measures essential for protecting both the application and its users.

One of the most significant risks identified is the exposure of MCP servers to the public internet without proper authentication. A July 2025 report from Knostic revealed approximately 2,000 internet-exposed MCP servers lacking any form of authentication, creating a massive vulnerability [5]. Another study by Backslash Security found widespread issues with over-permissioning, where servers were configured to request excessive scopes, and insecure local network exposure, where servers were accessible from other devices on the same network [5]. To counter this, developers must assume a zero-trust posture. This means treating every request as untrusted until verified, avoiding session-based authentication in favor of stateless, token-validated requests, and binding session IDs tightly to user identifiers to prevent cross-user attacks [28,36].

For production environments, scalability and observability are critical. Autoscaling should be enabled on cloud platforms to handle traffic spikes gracefully [2]. The server's performance must meet strict guidelines, such as ensuring 95% of requests respond in under five seconds, to pass OpenAI's app review process [37]. To achieve high performance, specialized gateways like TrueFoundry AI Gateway can be used, which are optimized to handle MCP traffic with low latency (sub-5ms) and high throughput (~350 RPS on a single vCPU) [36]. Observability is non-negotiable; every request lifecycle event—from initialization to tool execution and error—should be logged [42]. This logging is vital for debugging and for maintaining an audit trail in case of security incidents.

Finally, developers must stay ahead of evolving threats and protocol updates. The Replit incident, where an AI agent deleted a production database despite safeguards, underscores that even with proper controls, risks remain [5]. Future MCP developments aim to address these gaps, such as planned features for secure elicitation of out-of-band secrets and more granular, progressive scoping of permissions [5]. A promising pattern emerging is the use of a secure, external agent gateway to handle the most complex aspects of authorization, credential storage, and policy enforcement, effectively shielding the MCP server from the most dangerous parts of the security landscape [32]. By combining a robust implementation of OAuth 2.1, proactive security hardening, and a commitment to continuous monitoring and improvement, developers can build trustworthy and resilient MCP applications that safely unlock the power of external tools for AI agents.

Reference

1. Specification https://modelcontextprotocol.io/specification/latest
2. Development Guide for MCP Servers https://www.flowhunt.io/blog/mcp-server-development-guide/
3. Architecture overview https://modelcontextprotocol.io/docs/concepts/architecture
4. Building an MCP Server: A Developer's Guide https://www.opslevel.com/resources/building-an-mcp-server-a-developers-guide
5. What Is the Model Context Protocol (MCP) and How It Works https://www.descope.com/learn/post/mcp
6. Introducing the Model Context Protocol https://www.anthropic.com/news/model-context-protocol
7. Introducing apps in ChatGPT and the new Apps SDK https://openai.com/index/introducing-apps-in-chatgpt/
8. MCP server: A step-by-step guide to building from scratch https://composio.dev/blog/mcp-server-step-by-step-guide-to-building-from-scrtch
9. openai/openai-apps-sdk-examples https://github.com/openai/openai-apps-sdk-examples
10. OpenAI Platform: MCP Documentation https://platform.openai.com/docs/mcp
11. Getting Started: Running AI Models Locally with Ollama https://medium.com/@technobios/getting-started-running-ai-models-locally-with-ollama-41c2411c0833

12. Figma API Essential Guide - Rollout https://rollout.com/integration-guides/figma/api-essentials
13. Apps SDK https://developers.openai.com/apps-sdk/
14. OpenAI Apps SDK: How Developers Bring Services Into ... https://skywork.ai/blog/openai-apps-sdk-chatgpt-integration/
15. MCP authentication and authorization implementation guide https://stytch.com/blog/MCP-authentication-and-authorization-guide
16. Inside OpenAI's Apps SDK: Web Architecture Explained https://thenewstack.io/openai-launches-apps-sdk-for-chatgpt-a-new-app-platform/
17. Introduction to MCP authentication https://workos.com/blog/introduction-to-mcp-authentication
18. MCP developer guide | Visual Studio Code Extension API https://code.visualstudio.com/api/extension-guides/ai/mcp
19. Guide to authentication for the OpenAI Apps SDK - Stytch https://stytch.com/blog/guide-to-authentication-for-the-openai-apps-sdk/
20. Model Context Protocol (MCP) | Cursor Docs https://cursor.com/docs/context/mcp
21. Using Ollama for Image Generation: Create Stunning AI Art! https://www.arsturn.com/blog/using-ollama-for-image-generation-your-guide-to-creating-stunning-ai-art
22. ChatGPT Apps SDK: Proven 2025 Guide For Building Apps https://binaryverseai.com/chatgpt-apps-sdk-guide-build-apps-tutorial/
23. MCP Authentication & Authorization: Guide for 2025 https://www.infisign.ai/blog/what-is-mcp-authentication-authorization
24. How do I get Figma API to work with the Google App-script ... https://stackoverflow.com/questions/58792786/how-do-i-get-figma-api-to-work-with-the-google-app-script-api
25. Connect Figma To Google Sheets [Integration] https://apipheny.io/figma-api-google-sheets/
26. How to Integrate Google Cloud AI with Figma https://www.omi.me/blogs/ai-integrations/how-to-integrate-google-cloud-ai-with-figma?srsltid=AfmBOoraSLUAwbYYgSiTLf-aeWfMEq724AIod4S0HP7rjpRQ9MZra_9E
27. How to make next-level Figma plugins: auth, routing, ... https://evilmartians.com/chronicles/how-to-make-next-level-figma-plugins-auth-routing-storage-and-more
28. Security Best Practices https://modelcontextprotocol.io/specification/2025-06-18/basic/security_best_practices
29. Securing MCP Servers: A Comprehensive Guide to ... https://www.infracloud.io/blogs/securing-mcp-servers/
30. An Introduction to MCP and Authorization https://auth0.com/blog/an-introduction-to-mcp-and-authorization/

31. Design MCP Authorization to Securely Expose APIs https://curity.io/resources/learn/design-mcp-authorization-apis/
32. MCP Authorization Patterns for Upstream API Calls https://www.solo.io/blog/mcp-authorization-patterns-for-upstream-api-calls
33. Examples - OpenAI Agents SDK https://openai.github.io/openai-agents-python/examples/
34. Open Protocols for Agent Interoperability Part 2 - AWS https://aws.amazon.com/blogs/opensource/open-protocols-for-agent-interoperability-part-2-authentication-on-mcp/
35. Examples https://developers.openai.com/apps-sdk/build/examples/
36. What is MCP Server Authentication? https://www.truefoundry.com/blog/mcp-server-authentication
37. Building ChatGPT Apps with OpenAI Apps SDK - Cursor IDE https://www.cursor-ide.com/blog/openai-apps-sdk
38. GenAi to generate images locally and completely offline https://corradoignoti.medium.com/genai-to-generate-images-locally-and-completely-offline-b7009af9ff89
39. This app makes using Ollama local AI on MacOS devices ... https://www.zdnet.com/article/this-app-makes-using-ollama-local-ai-on-macos-devices-so-easy/
40. Draw Things Installation and Usage Tutorial https://blog.tangwudi.com/en/technology/homedatacenter12805/
41. Securing OpenAI's MCP Integration: From API Keys to ... https://medium.com/@richardhightower/securing-openais-mcp-integration-from-api-keys-to-enterprise-authentication-cafe40c049c1
42. SSE MCP Server with JWT Authentication https://glama.ai/mcp/servers/@anisirji/mcp-server-remote-setup-with-jwt-auth
43. How to build MCP server with Authentication in Python ... https://medium.com/@miki_45906/how-to-build-mcp-server-with-authentication-in-python-using-fastapi-8777f1556f75
44. How to build an MCP server with JWT verification https://www.linkedin.com/posts/ceposta_mcp-authorization-security-activity-7345830085599338496-EH0M
45. MCP OAuth 2.1 - A Complete Guide https://dev.to/composiodev/mcp-oauth-21-a-complete-guide-3g91