# Architectural Specification: Google Jules Model Context Protocol (MCP) Server Integration

## 1. Introduction: The Convergence of Agentic Coding and Standardized Protocols

The software development landscape is currently undergoing a foundational shift, moving from a paradigm of interactive code completion—where the developer leads and the AI assists—to one of agentic autonomy, where the AI leads execution under developer supervision. This transition is epitomized by Google’s Jules, an experimental coding agent designed to autonomously navigate repositories, plan architectural modifications, and execute code changes [1, 2]. Unlike synchronous chatbots that generate code snippets in isolation, Jules operates within a persistent "Session" model, executing multi-step workflows that involve environment setup, dependency analysis, plan generation, and pull request (PR) creation [2, 3].

However, the efficacy of such autonomous agents is frequently bottlenecked by the fragmentation of developer toolchains. An agent might exist within a web interface, while the developer operates within an Integrated Development Environment (IDE) like VS Code or Cursor. To bridge this operational gap, the Model Context Protocol (MCP) has emerged as the industry standard for interoperability. MCP acts as a universal "USB-C" for AI applications, establishing a standardized contract between "Hosts" (clients like Claude Desktop or IDEs) and "Servers" (providers of tools and context) [4, 5]. By encapsulating the Jules API within an MCP server, organizations can embed Google’s autonomous coding capabilities directly into their local development environments, creating a unified interface for task delegation, schedule management, and architectural review.

This report provides an exhaustive architectural specification for creating a Google Jules MCP Server. It evaluates every exposed endpoint of the Jules v1alpha API [6], mapping them to MCP primitives (Tools, Resources, and Prompts). Furthermore, it addresses the significant architectural challenge of "Scheduling." While the Jules web interface supports scheduled tasks [7], the current public API reference [6] focuses primarily on immediate session creation. Consequently, this report details the design of a **Client-Side Scheduler Subsystem** within the MCP server to satisfy the requirement for programmatic task scheduling, effectively polyfilling the feature for API consumers. The resulting architecture offers a robust, secure, and extensible foundation for integrating Jules into high-velocity engineering workflows.

## 2. Theoretical Framework: The Model Context Protocol (MCP) and Asynchronous Agency

Before dissecting the specific endpoints of the Jules API, it is essential to establish the theoretical model of the MCP server that will house them. The integration of an asynchronous agent (Jules) into a protocol often used for synchronous tool execution (MCP) presents unique state management challenges.

### 2.1 The MCP Topology: Clients, Hosts, and Servers

The Model Context Protocol operates on a client-server architecture designed to decouple the intelligence of the Large Language Model (LLM) from the mechanics of tool execution.

* **The MCP Host/Client:** This is the application where the user interacts (e.g., Claude Desktop, Cursor, or a custom IDE plugin). The Host is responsible for managing the connection, obtaining user authorization for sensitive actions, and rendering the output [8, 9].
* **The MCP Server:** This is the lightweight process that the Host spawns or connects to. In this specific architecture, the MCP Server is a Python application utilizing the mcp SDK [10, 11]. Its responsibility is to translate the JSON-RPC messages from the Host into HTTP requests destined for the Jules API.

### 2.2 Asynchrony and the "Task" Pattern

A critical divergence exists between the typical MCP tool lifecycle and the Jules operational model. Standard MCP tools are often synchronous functions: a calculator adds two numbers and returns the result immediately [12]. Jules, however, performs long-running operations. A request to "Refactor the authentication middleware" triggers a cloning process, a planning phase, and potentially minutes or hours of execution [2, 13].

If the MCP server were to block the connection while waiting for Jules to finish, it would likely trigger a timeout in the Host application [14]. Therefore, this specification adopts the **Asynchronous Task Pattern** [15, 16].

1. **Initiation:** The MCP Tool create\_task triggers the Jules session and immediately returns a "Session ID" and a status of QUEUED or PLANNING.
2. **Polling:** The MCP Server provides a secondary tool, get\_task\_status, or exposes a dynamic Resource jules://session/{id}/status, allowing the Host to poll for updates.
3. **Interruptibility:** The architecture must support the ability to pause or provide feedback during the AWAITING\_PLAN\_APPROVAL state, requiring a stateful understanding of the Jules workflow [17].

### 2.3 Security and Consequential Operations

The Jules API allows for the modification of source code and the creation of Pull Requests, actions that are inherently consequential. The MCP specification includes a mechanism to flag such tools [8]. All tools that map to Jules endpoints capable of mutation—specifically create\_session (when automationMode is active) and approvePlan—must be annotated with isConsequential: true. This forces the Host application to present a confirmation dialog to the human user, ensuring a "human-in-the-loop" security model [8, 18].

## 3. The Google Jules API Architecture: A Deep Analysis of v1alpha

The Jules API v1alpha is a RESTful interface built around three core resource concepts: **Sources**, **Sessions**, and **Activities** [3, 19]. Understanding the interdependencies of these resources is prerequisite to mapping them to MCP Tools.

### 3.1 Authentication and Transport Security

Access to the Jules API is gated via an API Key, which must be passed in the X-Goog-Api-Key HTTP header [20, 21].

* **Mechanism:** The user generates keys via the Jules settings page (jules.google.com/settings) [21].
* **MCP Implementation:** The MCP server must strictly avoid hardcoding credentials. Instead, it should initialize by reading the JULES\_API\_KEY from the environment variables [22, 23]. This aligns with MCP best practices for configuration management, allowing the server to be deployed in different environments (local dev, CI/CD) with different credentials.

### 3.2 The Resource Hierarchy

The API follows a hierarchical resource structure typical of Google APIs:

* **Sources (/v1alpha/sources):** Represents the connected repositories. These are the "targets" of any coding work [6].
* **Sessions (/v1alpha/sessions):** The operational container. A session ties a *prompt* to a *source* [3].
* **Activities (/v1alpha/sessions/{id}/activities):** The granular event log within a session. This includes plan generation, user messages, agent messages, and artifacts like diffs [17, 19].

This hierarchy dictates the discovery flow for the MCP server: it must first discover Sources to obtain the sourceId required to create a Session, and then monitor Activities to report progress.

## 4. Source Discovery and Context Management

The first functional requirement for the MCP server is to enable the LLM to understand *where* it can operate. The Sources resource provides this context.

### 4.1 Endpoint Evaluation: GET /sources

* **Function:** Retrieves a list of repositories that the user has connected to Jules via the GitHub integration [20].
* **Response Payload:** A list of Source objects, each containing:
  + name: The resource identifier (e.g., sources/github/user/repo).
  + githubRepo: Details including owner, repo name, default branch [17].
* **Constraint:** Users cannot connect new repositories via the API; this must be done in the Jules web UI [24]. The MCP server can only *list* existing connections.

### 4.2 MCP Integration: The list\_repositories Tool

While this could be exposed as a Resource, exposing it as a Tool allows the Agent to dynamically query availability during a conversation.

**Tool Definition:**

* **Name:** jules\_list\_repositories
* **Description:** "Lists all GitHub repositories currently connected to the Jules platform. Use this tool to find the valid 'source\_name' required to start a new coding session."
* **Input Schema:** {} (No parameters required).
* **Output Schema:** A JSON-formatted string listing the logical names and GitHub URLs of the repositories.

Implementation Logic:

The server creates a singleton HTTP client using httpx. Upon invocation, it performs a GET request to https://jules.googleapis.com/v1alpha/sources. The raw JSON response is parsed to extract only the relevant fields (name, githubRepo.owner, githubRepo.repo) to minimize context window usage for the LLM.

### 4.3 Contextual Optimization: jules://sources Resource

In addition to the tool, the server should expose a read-only Resource: jules://sources.

* **Utility:** This allows the user to "pin" the list of available repositories to the context window in their IDE (e.g., Cursor or VS Code) [9].
* **Mechanism:** When the Host requests read\_resource("jules://sources"), the server fetches the latest list and returns it as a formatted Markdown table.

**Table 1: Proposed Markdown Output for jules://sources**

| **Repository Name** | **Branch** | **Source ID (Required for Tasks)** |
| --- | --- | --- |
| my-app-backend | main | sources/github/org/my-app-backend |
| frontend-react | dev | sources/github/org/frontend-react |

This dual-mode exposure (Tool and Resource) ensures that both the Agent (via Tool) and the User (via Resource) have visibility into the available workspace.

## 5. Session Orchestration: Mapping Sessions to MCP Tools

The Sessions resource is the command center of the Jules API. It is where tasks are defined, initiated, and managed. This section details the comprehensive mapping of session creation and management to MCP tools.

### 5.1 Endpoint Evaluation: POST /sessions (Create Session)

This is the most complex endpoint in the API, requiring a multipart payload configuration.

* **Parameters:**
  + prompt (Required): The natural language instruction [19].
  + sourceContext (Required): A nested object containing the source ID and githubRepoContext (starting branch) [3, 20].
  + automationMode (Optional): An enum AUTO\_CREATE\_PR or AUTOMATION\_MODE\_UNSPECIFIED. This determines if Jules stops at the plan or proceeds to a PR [17].
  + requirePlanApproval (Optional): A boolean. If true, the session halts at AWAITING\_PLAN\_APPROVAL [24].
* **Response:** A Session object containing the id, name, and initial state (usually QUEUED or PLANNING) [17].

### 5.2 MCP Integration: The jules\_start\_task Tool

This tool serves as the primary entry point for the agent to delegate work to Jules.

**Tool Definition:**

* **Name:** jules\_start\_task
* **Description:** "Initiates a new asynchronous coding task in a specific repository. Returns a Session ID immediately. You must poll for status updates."
* **Input Schema:**  
  JSON  
  {  
   "type": "object",  
   "properties": {  
   "repository\_name": {  
   "type": "string",  
   "description": "The name of the repository (e.g., 'owner/repo'). The tool will resolve this to a source ID."  
   },  
   "instruction": {  
   "type": "string",  
   "description": "Detailed natural language prompt describing the coding task."  
   },  
   "branch": {  
   "type": "string",  
   "default": "main",  
   "description": "The starting branch for the task."  
   },  
   "auto\_create\_pr": {  
   "type": "boolean",  
   "default": false,  
   "description": "If true, Jules will automatically create a Pull Request upon completion."  
   },  
   "require\_approval": {  
   "type": "boolean",  
   "default": true,  
   "description": "If true, the task pauses for manual plan approval."  
   }  
   },  
   "required": ["repository\_name", "instruction"]  
  }
* **IsConsequential:** true (Triggers side effects/costs).

**Implementation Logic:**

1. **Resolution:** The tool first calls the internal \_get\_source\_id\_by\_name helper (using the cached GET /sources data) to translate "owner/repo" into the opaque sources/github/... string required by the API.
2. **Payload Construction:** It constructs the JSON body, mapping auto\_create\_pr to the AUTO\_CREATE\_PR enum if true [17].
3. **Execution:** Sends the POST request.
4. **Error Handling:** Catches 404 (Source not found) or 401 (Auth failed) errors and returns structured error messages to the LLM [25].

### 5.3 Endpoint Evaluation: GET /sessions/{id} and GET /sessions

To monitor progress, the agent needs visibility into the session's lifecycle.

* **State Machine:** The session transitions through states: QUEUED -> PLANNING -> AWAITING\_PLAN\_APPROVAL -> IN\_PROGRESS -> COMPLETED (or FAILED) [17].

### 5.4 MCP Integration: The jules\_get\_task\_status Tool

* **Name:** jules\_get\_task\_status
* **Description:** "Checks the current state of a running Jules session."
* **Input Schema:** {"session\_id": "string"}.
* **Output:** Returns a concise status report.
  + If state is AWAITING\_PLAN\_APPROVAL, the tool output *must* explicitly instruct the LLM: "The plan is ready. You should now retrieve the plan details and ask the user for approval."
  + If state is COMPLETED, it returns the SessionOutput which includes the Pull Request URL [17].

### 5.5 Endpoint Evaluation: POST /sessions/{id}:approvePlan

This endpoint is the "Execute" button for the planned changes.

* **Function:** Signals Jules to proceed from the planning phase to code generation/modification [20, 26].
* **Prerequisite:** The session must be in AWAITING\_PLAN\_APPROVAL state.

### 5.6 MCP Integration: The jules\_approve\_plan Tool

* **Name:** jules\_approve\_plan
* **Description:** "Approves the proposed plan for a session. This will trigger code modification."
* **Input Schema:** {"session\_id": "string"}.
* **IsConsequential:** true **(CRITICAL)**.
* **Security Context:** This annotation ensures that if an autonomous agent tries to call this tool, the user is prompted: "The agent wants to approve the plan for Session X. Allow?" [8, 14].

## 6. The Scheduling Subsystem: Architecting "Jules on Cron"

The original request explicitly requires the capability to create **schedules**. However, a rigorous analysis of the v1alpha API documentation reveals a discrepancy: while the Jules web interface supports "Scheduled Tasks" [7], the REST API reference does *not* document an endpoint for creating scheduled triggers (e.g., POST /scheduledTasks) [6]. The API currently allows only for the creation of immediate Sessions [19].

To satisfy the user requirement for "creating schedules in full," the MCP server must architect a **Client-Side Scheduler**. The MCP server itself will act as the cron daemon, triggering the standard createSession endpoint at user-defined intervals.

### 6.1 Architectural Design: The Persistent Scheduler

To implement reliability, the scheduler cannot simply be a transient generic Python loop, as the schedule would vanish if the server restarts.

* **Library Selection:** The implementation will utilize APScheduler (Advanced Python Scheduler) with a persistent job store (SQLAlchemy/SQLite) [27].
* **Storage:** A local SQLite database (jules\_schedule.db) will be created in the MCP server's configuration directory. This ensures that scheduled tasks persist across reboots of the MCP server or the host IDE.

### 6.2 The jules\_schedule\_task Tool

This new tool polyfills the missing API functionality.

**Tool Definition:**

* **Name:** jules\_schedule\_task
* **Description:** "Schedules a recurring Jules coding task. The task will be executed automatically by this server according to the specified cron schedule."
* **Input Schema:**  
  JSON  
  {  
   "type": "object",  
   "properties": {  
   "repository\_name": { "type": "string" },  
   "instruction": { "type": "string" },  
   "cron\_expression": {  
   "type": "string",  
   "description": "Standard cron string (e.g., '0 9 \* \* 1' for every Monday at 9am)."  
   },  
   "task\_name": { "type": "string", "description": "A unique identifier for this schedule." }  
   },  
   "required": ["repository\_name", "instruction", "cron\_expression", "task\_name"]  
  }

### 6.3 The jules\_list\_schedules and jules\_delete\_schedule Tools

To provide full CRUD (Create, Read, Update, Delete) capabilities over schedules:

* jules\_list\_schedules: Queries the local SQLite database to show all active jobs and their next run times.
* jules\_delete\_schedule: Removes a job from the APScheduler registry.

### 6.4 Execution Flow: The "Wake-Up" Cycle

1. **Registration:** The user invokes jules\_schedule\_task.
2. **Persistence:** The MCP server serializes the repository\_name and instruction and saves the job to SQLite.
3. **Trigger:** When the cron\_expression is satisfied, APScheduler invokes a callback function within the MCP server.
4. **API Call:** The callback function executes the exact logic of jules\_start\_task (resolves source -> POST /sessions).
5. **Logging:** The resulting session\_id is logged to a local file resource (jules://schedules/history), allowing the user to audit what tasks were autonomously started.

This architecture effectively creates a "Headless Jules Orchestrator," granting the user the power of scheduled maintenance (e.g., "Check for deprecated dependencies every Sunday") even without explicit support in the alpha API.

## 7. Activity Streams and Observability: Implementing Activities

Once a session is running (either manually or via schedule), the "Activity" resource becomes the primary window into the agent's cognition.

### 7.1 Endpoint Evaluation: GET /sessions/{id}/activities

* **Payload:** Returns a list of Activity objects. Each activity has an originator (USER, AGENT, SYSTEM) and a payload type [17].
* **Key Activity Types:**
  + PlanGenerated: Contains the Plan object with steps.
  + Artifact: Contains outputs like ChangeSet (git patches) or BashOutput.
  + AgentMessaged: Textual communication from Jules.

### 7.2 Handling Rich Data Artifacts

The Jules API returns high-density data that can overwhelm an LLM's context window. The MCP server must process this data intelligently.

The Plan Artifact:

When an activity of type PlanGenerated is detected, the MCP server should format the steps into a clean, numbered list.

* **MCP Tool:** jules\_get\_session\_plan
* **Function:** Fetches activities, filters for the latest planGenerated event, and returns the formatted text.

## **Format:** Plan for Session

* 1. **Analyze Dependencies:** Check package.json for outdated versions.
  2. **Modify Code:** Update utils.js to handle new query format.
  3. **Verify:** Run npm test.

The Diff Artifact (ChangeSet / GitPatch):

The Artifact type contains gitPatch with unidiffPatch strings [17].

* **Challenge:** Sending raw JSON containing escaped newline characters for a diff is hard for humans to read.
* **Solution:** The MCP server should parse the unidiffPatch and serve it as a *Resource*.
* **Resource URI:** jules://sessions/{id}/diff
* **Content:** The raw, unescaped diff text. This allows the Host application to potentially use its native diff viewer (if supported) or simply display the diff as a text file for the user to review before calling jules\_approve\_plan.

### 7.3 Interactive Feedback Loop: POST /sendMessage

If the plan is unsatisfactory, or if the state is AWAITING\_USER\_FEEDBACK, the user must intervene.

* **MCP Tool:** jules\_send\_feedback
* **Mapping:** Wraps the POST /v1alpha/sessions/{id}:sendMessage endpoint [28].
* **Usage:** The LLM client calls this tool with {"message": "Please add a unit test for the edge case of empty strings."}. This pushes a UserMessaged activity to the session, prompting Jules to revise its plan or continue execution [17].

## 8. Human-in-the-Loop and Security Governance

Integrating an autonomous coding agent into a local environment introduces significant risk. The MCP server serves as the governance layer, enforcing safety checks that might be optional in the raw API.

### 8.1 Plan Approval Enforcement

Although the API allows requirePlanApproval to be set to false, the MCP server implementation will enforce a "Safety by Default" policy.

* **Policy:** The jules\_start\_task tool defaults require\_approval to true.
* **Workflow:**
  1. Agent starts task.
  2. Agent polls status -> AWAITING\_PLAN\_APPROVAL.
  3. Agent fetches plan via jules\_get\_session\_plan.
  4. Agent presents plan to User.
  5. User confirms.
  6. Agent calls jules\_approve\_plan.

This workflow prevents the "Runaway Agent" scenario where a scheduled task silently breaks the build or introduces vulnerabilities.

### 8.2 Environment Variable Security

Jules tasks often require environment variables (API keys, database URLs) to run tests [22, 29].

* **API Feature:** The v1alpha API allows defining environment variables for tasks.
* **MCP Handling:** The server configuration should allow an env\_allowlist. When jules\_start\_task is called, the MCP server checks this allowlist and injects the permitted local environment variables into the session payload. This prevents accidental leakage of sensitive local secrets (like AWS\_SECRET\_KEY) to the remote Jules VM unless explicitly authorized.

## 9. Implementation Guide: Python/FastMCP Specification

The following section outlines the structural implementation of the server using Python.

### 9.1 Dependency Architecture

Python

# requirements.txt  
mcp[cli]>=1.0.0  
httpx>=0.24.0  
apscheduler>=3.10.0  
sqlalchemy>=2.0.0 # For persistent schedule storage  
pydantic>=2.0.0 # For strict schema validation

### 9.2 The JulesClient Wrapper

To abstract the REST complexity, a dedicated client class is required.

Python

import httpx  
from typing import Optional, Dict, Any  
  
class JulesClient:  
 BASE\_URL = "https://jules.googleapis.com/v1alpha"  
  
 def \_\_init\_\_(self, api\_key: str):  
 self.headers = {"X-Goog-Api-Key": api\_key, "Content-Type": "application/json"}  
 self.client = httpx.Client(base\_url=self.BASE\_URL, headers=self.headers, timeout=30.0)  
  
 def create\_session(self, prompt: str, source\_id: str,   
 branch: str = "main", auto\_pr: bool = False,   
 require\_approval: bool = True) -> Dict[str, Any]:  
 payload = {  
 "prompt": prompt,  
 "sourceContext": {  
 "source": source\_id,  
 "githubRepoContext": {"startingBranch": branch}  
 },  
 "automationMode": "AUTO\_CREATE\_PR" if auto\_pr else "AUTOMATION\_MODE\_UNSPECIFIED",  
 "requirePlanApproval": require\_approval  
 }  
 resp = self.client.post("/sessions", json=payload)  
 resp.raise\_for\_status()  
 return resp.json()  
   
 #... Implementation of get\_session, list\_activities, approve\_plan...

### 9.3 The Server Entry Point

Using FastMCP simplifies the tool registration [11].

Python

from mcp.server.fastmcp import FastMCP  
from apscheduler.schedulers.background import BackgroundScheduler  
  
# Initialize Server  
mcp = FastMCP("Google Jules MCP")  
  
# Initialize Scheduler  
scheduler = BackgroundScheduler()  
scheduler.add\_jobstore('sqlalchemy', url='sqlite:///jules\_schedule.db')  
scheduler.start()  
  
@mcp.tool()  
def jules\_start\_task(repository\_name: str, instruction: str) -> str:  
 """Starts a new coding task."""  
 # Logic to resolve repo name and call client.create\_session  
 pass  
  
@mcp.tool()  
def jules\_schedule\_task(repository\_name: str, instruction: str, cron\_expression: str) -> str:  
 """Schedules a recurring task."""  
 # Logic to add job to scheduler  
 scheduler.add\_job(  
 func=trigger\_jules\_session,   
 trigger='cron',   
 args=[repository\_name, instruction],  
 # parsing logic for cron\_expression  
 )  
 return f"Task scheduled with cron: {cron\_expression}"  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 mcp.run()

## 10. Conclusion

The architecture proposed in this report transforms Google Jules from a siloed web application into an integral component of the local development ecosystem. By meticulously mapping the v1alpha API endpoints (Sources, Sessions, Activities) to MCP primitives, we create a system where an AI agent in the IDE can seamlessly "phone a friend" (Jules) to handle heavy-lifting tasks asynchronously.

Crucially, the integration of a **Client-Side Scheduler** directly addresses the gap in the current API specification regarding programmatic scheduling. This innovation allows development teams to define "self-healing" repository workflows—such as nightly builds, dependency audits, and automated refactoring—that run autonomously, yet remain fully governable through the MCP's security protocols. This implementation not only satisfies the requirements of the original request but establishes a forward-looking blueprint for the integration of remote agentic services into local tools via the Model Context Protocol.