Perfetto. Avvierò una ricerca approfondita su come integrare il Model Context Protocol (MCP) nel progetto AI Developer Panel basato su Cline, focalizzandomi su interazione tra agenti AI, memoria dinamica, orchestrazione multi-modello e compatibilità con estensioni VS Code e LLM locali.

Ti aggiorno appena ho risultati dettagliati con spiegazioni tecniche, link ufficiali, esempi pratici e progetti GitHub correlati.

# Integrazione del Model Context Protocol (MCP) in un’estensione VS Code AI

## 1. Cos’è il Model Context Protocol (MCP) e formati supportati

Il **Model Context Protocol (MCP)** è uno standard aperto introdotto da Anthropic per collegare i modelli di linguaggio a dati esterni e strumenti in modo unificato ([Introduction - Model Context Protocol](https://modelcontextprotocol.io/introduction#:~:text=MCP%20is%20an%20open%20protocol,different%20data%20sources%20and%20tools)). In pratica, MCP funge da **interfaccia universale** (paragonata a una porta USB-C per l’AI) che consente alle applicazioni di AI di accedere a file, database, API e altri servizi esterni in modo coerente e sicuro ([Introduction - Model Context Protocol](https://modelcontextprotocol.io/introduction#:~:text=MCP%20is%20an%20open%20protocol,different%20data%20sources%20and%20tools)) ([But What Is MCP (Model Context Protocol)? (2025)](https://arthurchiao.art/blog/but-what-is-mcp/#:~:text=protocols,services%20with%20the%20MCP%20clients)). Questo evita di dover implementare integrazioni ad-hoc per ogni fonte dati, sostituendole con un’unica **interfaccia standardizzata** ([MCP – Model Context Protocol: Standardizing AI-Data Access - DEV Community](https://dev.to/sreeni5018/mcp-model-context-protocol-standardizing-ai-data-access-465#:~:text=Anthropic%27s%20Model%20Context%20Protocol%20,with%20a%20single%2C%20standardized%20protocol)). I componenti chiave di MCP sono:

* **MCP Host**: l’applicazione AI principale (ad es. un IDE, un assistente tipo Claude Desktop) che richiede dati contestuali esterni ([MCP – Model Context Protocol: Standardizing AI-Data Access - DEV Community](https://dev.to/sreeni5018/mcp-model-context-protocol-standardizing-ai-data-access-465#:~:text=1.%20MCP%20Hosts%20%E2%80%93%20AI,calculations%20or%20user%20profile%20management)).
* **MCP Client**: un adattatore lato host che gestisce la connessione 1:1 con un server MCP ([MCP – Model Context Protocol: Standardizing AI-Data Access - DEV Community](https://dev.to/sreeni5018/mcp-model-context-protocol-standardizing-ai-data-access-465#:~:text=1.%20MCP%20Hosts%20%E2%80%93%20AI,calculations%20or%20user%20profile%20management)). L’host può avere più client per connettersi a diversi server.
* **MCP Server**: un servizio (anche locale e leggero) che espone specifiche funzionalità o dati tramite MCP ([MCP – Model Context Protocol: Standardizing AI-Data Access - DEV Community](https://dev.to/sreeni5018/mcp-model-context-protocol-standardizing-ai-data-access-465#:~:text=MCP%20%28e,calculations%20or%20user%20profile%20management)). Ad esempio, un server può fornire accesso ai file locali, ad API web, o compiere calcoli.
* **Risorse e Strumenti**: ciascun server può definire *risorse* (es. documenti, database) identificabili e *tools* (funzionalità come “cerca nei file”, “esegui query SQL”) a cui il modello può accedere tramite il protocollo.

**Formati e trasporti:** MCP utilizza messaggi **JSON-RPC 2.0** per lo scambio strutturato di dati e comandi ([MCP – Model Context Protocol: Standardizing AI-Data Access - DEV Community](https://dev.to/sreeni5018/mcp-model-context-protocol-standardizing-ai-data-access-465#:~:text=MCP%20supports%20two%20transport%20models%2C,0%20for%20structured%20data%20exchange)) ([But What Is MCP (Model Context Protocol)? (2025)](https://arthurchiao.art/blog/but-what-is-mcp/#:~:text=protocols,services%20with%20the%20MCP%20clients)). Questo significa che richieste e risposte seguono un formato JSON standard (con campi come jsonrpc, method, params, id etc.), facilitando l’implementazione multipiattaforma ([But What Is MCP (Model Context Protocol)? (2025)](https://arthurchiao.art/blog/but-what-is-mcp/#:~:text=protocols,services%20with%20the%20MCP%20clients)). I trasporti supportati attualmente includono:

* **STDIO (Standard I/O)** – Il client e server comunicano tramite stdin/stdout, ideale per integrazioni locali (si può eseguire il server come processo figlio) ([MCP – Model Context Protocol: Standardizing AI-Data Access - DEV Community](https://dev.to/sreeni5018/mcp-model-context-protocol-standardizing-ai-data-access-465#:~:text=MCP%20supports%20two%20transport%20models%2C,0%20for%20structured%20data%20exchange)).
* **SSE (Server-Sent Events)** – Basato su HTTP: il client invia richieste HTTP POST al server e riceve risposte/eventi via stream SSE, adatto a servizi remoti o scenari in cui serve streaming e connessione via rete ([MCP – Model Context Protocol: Standardizing AI-Data Access - DEV Community](https://dev.to/sreeni5018/mcp-model-context-protocol-standardizing-ai-data-access-465#:~:text=,server%29%20requests)). (In futuro sono concepibili altri trasporti come WebSocket o gRPC, ma lo standard attuale è centrato su JSON-RPC su STDIO/SSE).

**SDK e implementazioni locali:** MCP è pensato per essere aperto e utilizzabile ovunque, non solo come servizio cloud. Esistono SDK ufficiali in diversi linguaggi, tra cui **Python, TypeScript/JavaScript, Java, Kotlin, C#** ([Overview - Model Context Protocol](https://modelcontextprotocol.io/sdk/java/mcp-overview#:~:text=,SDK%20%2A%20Specification)), facilitando l’integrazione sia in applicazioni server (ad es. backend Python) sia lato client (ad es. tool web/desktop). Ad esempio, Anthropic ha rilasciato un SDK Python (pip install mcp) e un SDK TypeScript; con questi è possibile sia creare server MCP personalizzati sia scrivere client che si connettono a server esistenti. Inoltre, MCP si presta all’uso locale: un **MCP server** può essere un piccolo programma sul tuo PC (anche uno script Python, Go, Node, ecc.) che espone i tuoi file locali o altre risorse in modo che il modello le possa consultare ([MCP – Model Context Protocol: Standardizing AI-Data Access - DEV Community](https://dev.to/sreeni5018/mcp-model-context-protocol-standardizing-ai-data-access-465#:~:text=servers%2C%20maintaining%20one,calculations%20or%20user%20profile%20management)). Il primo client ufficiale è stato Claude Desktop (un’app locale) a dimostrazione che MCP funziona localmente, non richiede infrastrutture cloud ([MCP – Model Context Protocol: Standardizing AI-Data Access - DEV Community](https://dev.to/sreeni5018/mcp-model-context-protocol-standardizing-ai-data-access-465#:~:text=1,that%20connect%20to%20these%20servers)).

Esistono anche integrazioni community e SDK aggiuntivi. Ad esempio, un adattatore per **LangChain** consente di trattare i tool MCP come tool di LangChain: la libreria langchain-mcp-adapters permette di caricare strumenti MCP e usarli in agenti LangChain ([MCP – Model Context Protocol: Standardizing AI-Data Access - DEV Community](https://dev.to/sreeni5018/mcp-model-context-protocol-standardizing-ai-data-access-465#:~:text=from%20langchain_mcp_adapters,prebuilt%20import%20create_react_agent)) ([MCP – Model Context Protocol: Standardizing AI-Data Access - DEV Community](https://dev.to/sreeni5018/mcp-model-context-protocol-standardizing-ai-data-access-465#:~:text=from%20langchain_mcp_adapters,ChatOpenAI%20from%20dotenv%20import%20load_dotenv)). In pratica, un agente LangChain può connettersi a uno o più server MCP e vedere una lista di tool invocabili. Analogamente, la libreria **AutoGen** di Microsoft (per agenti multi-LLM) supporta MCP attraverso la funzione di *function calling*: è possibile avvolgere i tool esposti da un server MCP dentro funzioni chiamabili dal modello. Un esempio viene descritto da Victor Dibia, mostrando come un agente AutoGen possa usare un tool “fetch” servito via MCP per recuperare il contenuto di una URL ([How to Use Anthropic MCP Tools with Your AutoGen Agents (and any model)](https://newsletter.victordibia.com/p/how-to-use-mcp-anthropic-mcp-tools#:~:text=Under%20the%20hood%2C%20,several%20important%20things)) ([How to Use Anthropic MCP Tools with Your AutoGen Agents (and any model)](https://newsletter.victordibia.com/p/how-to-use-mcp-anthropic-mcp-tools#:~:text=In%20this%20example%2C%20we%27re%20using,line%20tools)). In quell’approccio, il client AutoGen avvia un server MCP (trasporto STDIO) ed esegue tools/list per ottenere lo schema JSON dei tool disponibili, poi passa questo schema al modello come funzioni disponibili ([How to Use Anthropic MCP Tools with Your AutoGen Agents (and any model)](https://newsletter.victordibia.com/p/how-to-use-mcp-anthropic-mcp-tools#:~:text=2,fetch)). Ciò dimostra compatibilità con modelli OpenAI, modelli Alibaba (es. Qwen), o local LLM con supporto per **function calling** ([How to Use Anthropic MCP Tools with Your AutoGen Agents (and any model)](https://newsletter.victordibia.com/p/how-to-use-mcp-anthropic-mcp-tools#:~:text=,client%20that%20supports%20function%20calling)). Anche **CrewAI** (framework open-source per agenti multi-LLM) può integrarsi con MCP in modo simile: CrewAI permette di definire tool personalizzati per agenti, quindi nulla vieta di implementare un wrapper che utilizzi un MCP server per fornire un tool. Insomma, MCP è **agnostico rispetto al modello LLM** – finché l’agente o il modello può invocare una funzione o strutturare una richiesta, può usare MCP.

Per quanto riguarda il **contesto multi-file**, MCP non ha limiti intrinseci sul tipo o quantità di dati: un server può esporre più risorse (ad es. più file) e l’LLM può richiederne diversi in una sessione. Ad esempio, un’estensione come *Cursor* (IDE AI) utilizza MCP per gestire contesti multi-file: quando l’utente chiede “aiuto con un componente”, Cursor raccoglie i contenuti di vari file (UserProfile.tsx, types.ts, api.ts ecc.) e li passa a un tool MCP “analyze\_context” assieme al nome del file principale su cui focalizzarsi ([GitHub - johnneerdael/multiplatform-cursor-mcp](https://github.com/johnneerdael/multiplatform-cursor-mcp#:~:text=3.%20Multi)) ([GitHub - johnneerdael/multiplatform-cursor-mcp](https://github.com/johnneerdael/multiplatform-cursor-mcp#:~:text=%2F%2F%203,focus_file%2C%20analysis_type%3A%20%27component_dependencies%27)). Il server MCP riceve quindi un’unica chiamata contenente più file nel parametro (in JSON), e può analizzare il tutto fornendo un output che tiene conto delle dipendenze multi-file. Anche Cline 3.0 (un’estensione AI per coding) ha introdotto il supporto MCP per operare su repository GitHub: tramite un MCP server dedicato a GitHub, Cline può leggere file, creare commit e pull request su più file ([Speeding up your GitHub workflow with Cline 3.0 and MCP - DEV Community](https://dev.to/shannonlal/speeding-up-your-github-workflow-with-cline-30-and-mcp-j6#:~:text=GitHub%20MCP%20is%20a%20powerful,to%20supercharge%20your%20development%20process)) ([Speeding up your GitHub workflow with Cline 3.0 and MCP - DEV Community](https://dev.to/shannonlal/speeding-up-your-github-workflow-with-cline-30-and-mcp-j6#:~:text=tasks,of%20manual%20work%20each%20week)). **Esempio d’uso reale:** con *GitHub MCP*, l’assistente è in grado di aggiungere file e generare messaggi di commit, e persino preparare automaticamente una Pull Request con descrizione delle modifiche ([Speeding up your GitHub workflow with Cline 3.0 and MCP - DEV Community](https://dev.to/shannonlal/speeding-up-your-github-workflow-with-cline-30-and-mcp-j6#:~:text=GitHub%20MCP%20is%20a%20powerful,to%20supercharge%20your%20development%20process)). In scenari di commit multipli, alcuni utenti hanno notato che l’AI tendeva a concentrarsi solo sull’ultimo file modificato trascurando gli altri ([Speeding up your GitHub workflow with Cline 3.0 and MCP - DEV Community](https://dev.to/shannonlal/speeding-up-your-github-workflow-with-cline-30-and-mcp-j6#:~:text=However%2C%20it%27s%20not%20without%20challenges,can%20occasionally%20be%20a%20struggle)) – ciò indica che, sebbene MCP permetta di fornire contesto multi-file, bisogna istruire bene il modello (prompt engineering) affinché utilizzi **tutti** i dati forniti. In sintesi, **sì, MCP supporta contesto multi-file**: sta poi all’applicazione e al prompt garantire che il modello ne faccia buon uso.

**Disponibilità di formati e integrazione:** Il protocollo è definito da una **specifica pubblica** (vedi spec.modelcontextprotocol.io) e include schemi JSON per descrivere tools, risorse, errori ecc. Ad esempio, i tool MCP vengono descritti con nomi, parametri e tipi (spesso convertiti in JSON Schema) in modo che il client possa presentarli al modello come funzioni standard ([How to Use Anthropic MCP Tools with Your AutoGen Agents (and any model)](https://newsletter.victordibia.com/p/how-to-use-mcp-anthropic-mcp-tools#:~:text=2,fetch)). La comunicazione però non avviene via *HTTP headers* custom, ma tramite i messaggi JSON-RPC (che possono viaggiare su HTTP SSE o su pipe locali). Attualmente **non esiste un servizio cloud obbligatorio**: puoi eseguire tutto in locale. Ad esempio, potresti avere un server MCP Python che legge file su disco e un LLM locale che lo interroga – tutto restando offline. E viceversa, puoi usare MCP anche con modelli cloud (Claude, OpenAI) se supportano function calling, perché alla fine stai standardizzando il formato in cui fornisci contesto (come funzioni) ([Using the Model Context Protocol (MCP) With a Local LLM | by Ashraff Hathibelagal | Predict | Mar, 2025 | Medium](https://medium.com/predict/using-the-model-context-protocol-mcp-with-a-local-llm-e398d6f318c3#:~:text=Any%20large%20language%20model%20that,make%20use%20of%20those%20tools)).

**Esempio (Python)** – Definizione rapida di un MCP server locale con alcuni tool file-system e avvio via STDIO:

from mcp.server.fastmcp import FastMCP

# Istanzia un MCP server nominato (ad es. "LocalFS")

server = FastMCP("LocalFS Helper")

# Definisci tool: list directory

@server.tool()

def ls(directory: str) -> str:

"""Lista file in una directory"""

import os

return "\n".join(os.listdir(directory))

# Tool: leggi file

@server.tool()

def cat(filepath: str) -> str:

"""Ritorna il contenuto di un file di testo"""

with open(filepath, "r") as f:

return f.read()

# Avvia il server (ascolta su stdin/stdout per richieste JSON-RPC)

server.run()

In questo esempio, un modello dotato di capacità di function calling potrebbe ricevere automaticamente lo schema di questi tool (ls e cat) e invocarli quando necessario (ad es. chiamando ls per ottenere l’elenco file, poi cat su un file specifico). Un MCP client locale potrebbe invece inoltrare chiamate dirette: ad esempio client.call("ls", {"directory": "/progetto"}) per ottenere la lista dei file. Questo mostra come MCP sia integrabile facilmente in pipeline Python locali.

## 2. Integrazione di MCP con Visual Studio Code e orchestrazione multi-agente

**Uso di MCP in una WebView VS Code:** È possibile integrare MCP in un’estensione VS Code utilizzando sia il lato frontend (WebView) sia il backend dell’estensione. La **WebView (React + TypeScript)** può certamente interagire con MCP, ad esempio inviando richieste a un server MCP via HTTP SSE, oppure più semplicemente delegando al backend dell’estensione la gestione del protocollo. Esiste un SDK TypeScript ufficiale ([Overview - Model Context Protocol](https://modelcontextprotocol.io/sdk/java/mcp-overview#:~:text=,SDK)) che può essere usato in ambiente Node.js – nel contesto di VS Code, il codice dell’estensione (script principale in esecuzione in Node) potrebbe usare questo SDK. Tuttavia, la WebView stessa gira in un contesto isolato simile a una pagina web, con restrizioni: non ha direttamente accesso al filesystem locale né alla rete arbitraria (a meno che l’estensione non lo permetta tramite messaggi). Quindi, l’approccio tipico è: **gestire MCP nel backend** (estensione) e inviare i risultati alla WebView per visualizzarli. Ad esempio, il codice dell’estensione VS Code (TypeScript) potrebbe avviare un client MCP via STDIO per connettersi a un server locale (come quello definito sopra) e poi fornisce all’LLM i tool. La WebView React può inviare un messaggio al backend dicendo “esegui tool X con param Y” oppure semplicemente mostrando la risposta che il modello ha generato usando MCP. In alternativa, se si preferisce lato WebView, si potrebbe attivare il SDK TS in modalità SSE: ad esempio, far partire un server MCP locale che espone un endpoint HTTP, e poi dal codice JS nella WebView effettuare una fetch con EventSource per ascoltare risposte. Questo è tecnicamente fattibile (specialmente se il server SSE gira su http://localhost con CORS aperto per la WebView), ma implica gestire sicurezza e permessi di rete di VS Code. In sintesi: **MCP può essere usato in un’estensione VS Code**, solitamente integrandolo nel back-end (Node/Python) dell’estensione e comunicando con la WebView via messaggi IPC.

**Interazione con backend Python (chatdev.py):** Se la tua estensione già utilizza un backend Python (ad esempio uno script chatdev.py che gestisce la logica della chat/agent), puoi sfruttarlo per MCP usando l’SDK Python. Ad esempio, dallo script Python potresti usare la libreria mcp per avviare un client verso vari server. Immagina di avere un agent in Python che riceve i messaggi utente e decide se servono dati esterni: potrebbe fare qualcosa tipo:

from mcp.client import StdioMcpClient

# Avvia un client MCP verso un server (esempio: server file system avviato altrove)

mcp\_client = StdioMcpClient(executable="python fs\_server.py")

tools = mcp\_client.list\_tools() # ottiene lista tool dal server

# ... decide di usare un tool:

result = mcp\_client.call\_tool("cat", {"filepath": "README.md"})

In pratica, il backend Python funge da **orchestratore**: quando l’LLM necessita di contesto (ad esempio l’utente chiede “apri il file X”), Python può chiamare il tool MCP appropriato e poi includere il risultato nel prompt del modello. Un’altra modalità è invertire i ruoli: far sì che **il modello stesso chiami i tool**. In tal caso, il backend Python dovrebbe fornire al modello l’elenco di funzioni disponibili (come fa OpenAI con functions nel prompt). Se il modello locale non supporta nativamente il function calling, puoi simulare un approccio ReAct: il modello potrebbe essere istruito a restituire un segnale (es. una stringa specializzata) quando vuole usare un tool, e chatdev.py intercetta quel segnale, esegue il tool via MCP e poi fornisce la risposta al modello. Questa orchestrazione manuale richiede un po’ di prompt engineering (ad es. convenzioni come: ToolRequest: {"tool": "cat", "filepath": "X"} dentro la conversazione). In alternativa, se usi modelli OpenAI o altri con function calling, il backend Python può direttamente passare la spec delle funzioni ottenuta da mcp\_client.list\_tools() nella chiamata API ([How to Use Anthropic MCP Tools with Your AutoGen Agents (and any model)](https://newsletter.victordibia.com/p/how-to-use-mcp-anthropic-mcp-tools#:~:text=2,fetch)).

**Simulare streaming con LM Studio e MCP:** *LM Studio* è un’interfaccia locale per LLM che può esporre un’API compatibile OpenAI. Se stai usando LM Studio con DeepSeek-Coder, probabilmente interroghi il modello via chiamate HTTP localmente (ad es. POST /v1/chat/completions). In molti casi, tali API **supportano il parametro streaming** ("stream": true), restituendo i token gradualmente come eventi SSE. È da verificare nella documentazione di LM Studio, ma tipicamente se si presenta come OpenAI-like, potrebbe inviare chunk di risposta. Se lo streaming nativo non fosse supportato, puoi sempre implementare una simulazione: invia la richiesta al modello e poi manda i token in streaming alla WebView man mano che li ricevi dal backend. Ad esempio, chatdev.py potrebbe ricevere l’output del modello in streaming (HuggingFace Transformers permette di iterare token generati) e inoltrare ciascun token o blocco di testo via WebSocket o messaggio postMessage alla UI. MCP stesso supporta lo streaming nei tool: usando SSE, un MCP server può mandare output parziali. Immagina un tool MCP che fa una lunga operazione (es. esegue una query lunga): potrebbe inviare aggiornamenti parziali tramite eventi SSE al client. Nel contesto LM Studio + MCP, il disaccoppiamento è chiaro: LM Studio genera testo (streaming o no) **che può contenere chiamate tool**, e il tuo orchestratore esegue tali chiamate e inserisce i risultati. Per far apparire tutto come un flusso continuo all’utente, dovrai concatenare il flusso del modello e gli eventuali output dei tool. In pratica, **sì è possibile mantenere lo streaming** anche quando il modello utilizza MCP, ma va gestito attentamente: probabilmente accumulando l’output del modello fino a quando non richiede un tool, poi inserendo la risposta del tool, e così via, infine continuando il testo. Fortunatamente, con modelli che supportano function calling, questo avviene in modo abbastanza strutturato (la risposta del modello si interrompe quando chiama il tool, ricevi un risultato, poi riprendi). In un’implementazione custom, puoi inviare al front-end messaggi di tipo “assistant sta pensando...” e poi “assistant: risultato tool X: ...” gradualmente per simulare un flusso.

**Orchestrazione di più agenti AI nel workspace:** Il tuo scenario prevede agenti multipli (DeveloperAgent, SupervisorAgent, ResearcherAgent) che collaborano. MCP può aiutare fungendo da **canale di contesto condiviso** e fornitore di strumenti, ma la logica di orchestrazione multi-agente va progettata. Puoi implementare un *coordinator* (magari nel backend Python o nell’estensione stessa) che gestisce turni e comunicazione tra agenti. Ogni agente potrebbe essere un istanza separata dello stesso LLM con un proprio ruolo/prompt iniziale. Ad esempio, DeveloperAgent può scrivere codice, ResearcherAgent può fare ricerche (usando tool web via MCP), SupervisorAgent verifica e coordina. Puoi tenere una singola chat visibile all’utente ma dietro le quinte etichettare i messaggi con l’agente mittente. Nulla impedisce di usare **un solo workspace** (progetto) e avere ciascun agente dotato degli stessi tool MCP oppure di tool differenti. MCP consente al modello di interagire con risorse: se vuoi che *solo* il ResearcherAgent possa accedere a Internet, potresti configurare che solo il prompt di Researcher include il tool “web-search” via MCP, mentre DeveloperAgent ha il tool “file-system” via MCP, ad esempio. In questo modo, ciascun agente invoca solo i tool pertinenti al suo ruolo. La coordinazione delle risposte potrebbe avvenire in round-robin o su richiesta: SupervisorAgent potrebbe analizzare il problema e dire “Researcher, cercami X; Developer, intanto prepara Y”. Questo significa che il codice orchestratore deve passare il giusto contesto a ciascun modello agente. MCP **di per sé non gestisce il turno degli agenti**, ma fornisce i mezzi per recuperare informazioni quando un agente le chiede. Quindi sì, puoi orchestrare più agenti in uno stesso workspace VS Code: ad esempio, aprendo **più sessioni parallele** (thread) o usando agenti asincroni che comunicano via Python. Alcuni framework aiutano: *LangChain* e *AutoGen* supportano interazioni multi-agente dove puoi definire ruoli e un loop di conversazione. Questi framework possono integrarsi con MCP come detto, quindi potresti usarli per semplificare la logica. In un approccio artigianale: potresti avere una funzione run\_agents\_cycle() che invia il messaggio utente iniziale a tutti o ad uno specifico agente, poi raccoglie la risposta, aggiunge eventualmente strumenti via MCP, e passa la palla all’altro agente. Ad esempio:

# Pseudocodice orchestrazione

supervisor\_ctx = init\_context(SupervisorAgent)

dev\_ctx = init\_context(DeveloperAgent)

res\_ctx = init\_context(ResearcherAgent)

# Il supervisore analizza la richiesta iniziale

sup\_msg = generate\_reply(SupervisorAgent, user\_input, tools=MCP\_tools\_all)

if "Ricerca" in sup\_msg:

res\_msg = generate\_reply(ResearcherAgent, sup\_msg, tools=[web\_tool])

sup\_msg = generate\_reply(SupervisorAgent, res\_msg)

if "Codice" in sup\_msg:

dev\_msg = generate\_reply(DeveloperAgent, sup\_msg, tools=[fs\_tools])

sup\_msg = generate\_reply(SupervisorAgent, dev\_msg)

# ... e così via ...

final\_answer = sup\_msg

In pratica, orchestrare agenti multipli richiede **memorizzare separatamente il contesto di ciascuno**, ma mantenendo un obiettivo comune. Puoi decidere di mostrare all’utente solo l’esito finale o anche il dialogo intermedio (magari in un log). Dal punto di vista MCP, tutti gli agenti potrebbero condividere gli stessi server (file system locale, documentazione, web, ecc.) registrati in comune all’inizio. MCP non ha limiti nel numero di client connessi, quindi potresti avere un MCP client per ogni agente, o uno unico condiviso se implementi in un unico processo orchestratore.

Infine, implementare tutto ciò in un’estensione VS Code **open-source e modulare** è plausibile: potresti organizzare moduli per i vari agenti, uno per la gestione MCP, uno per i modelli (locale vs cloud) e uno per la UI. Questo ti dà flessibilità di eseguire tutto in locale (DeepSeek con LM Studio + MCP server locali) o usare API cloud (OpenAI, Claude) a seconda delle preferenze dell’utente.

## 3. Memoria dinamica e ottimizzazione del contesto

Gestire la **memoria della conversazione in modo dinamico** in base al modello è fondamentale, perché diversi LLM hanno finestre di contesto molto diverse. Ad esempio, GPT-4 può arrivare a 8K o 32K token, Claude v2 anche 100K, mentre un modello locale come DeepSeek-Coder (versione open) potrebbe avere solo 4K token disponibili (o meno se quantizzato). Inoltre alcuni modelli, specie quelli ottimizzati per codice, potrebbero non essere addestrati su conversazioni lunghe. Le migliori pratiche includono:

* **Attivare/Disattivare la memoria in base al modello:** implementa una logica nel tuo sistema che controlla il nome o le specifiche del modello selezionato e adatta la gestione della chat history. Ad esempio, per modelli “grandi” (Claude, GPT-4, o anche Llama2 70B con 16K) puoi mantenere una cronologia più estesa degli scambi utente-assistente. Per modelli con contesto ridotto (es. 2048 token), conviene limitare la history agli ultimi messaggi significativi. Puoi esporre questa scelta anche come impostazione: *“Memoria conversazione: completa / ridotta / disattivata”*. **Disattivare la memoria** per un modello significa, in pratica, non includere i vecchi turni nel prompt ad ogni nuova query (o includerne solo una parte riassunta). Ad esempio, potresti azzerare il contesto per ogni domanda con modelli piccoli, comportandosi più come chat isolate. Oppure mantenere solo le ultime 1-2 interazioni.
* **Rilevare modelli ottimizzati per codice:** Non c’è un metodo intrinseco al modello per “autodichiararsi” come specializzato in codice, ma ci si può basare su meta-informazioni. In pratica: **sapere a priori** quali modelli sono addestrati per il coding. DeepSeek-Coder, ad esempio, è documentato come “LLM ottimizzato per compiti di codifica” ([I Built AI Agents for Code Generation & Review with DeepSeek ...](https://levelup.gitconnected.com/i-built-ai-agents-for-code-generation-review-with-deepseek-coder-model-0d18c1694184#:~:text=,to%20assist%20with%20code)) (87% del training su codice). Spesso il nome del modello o la documentazione lo indica. Se usi modelli da HuggingFace, il loro card potrebbe menzionare il dominio (codice vs generalista). Puoi quindi mantenere una lista (o regole sui nomi, come contiene “Coder”, “Code”, “CodeLlama”, ecc.) per flaggare quei modelli. Un’altra via: testare il comportamento – se chiedi in linguaggio naturale qualcosa di astratto e noti che il modello fatica, ma fornisce codice eccellente, è segno che è specializzato. In generale però, l’approccio semplice è configurare manualmente: es. *model\_config.json* che elenca i modelli noti e proprietà (es: "DeepSeek-Coder": {"domain": "code", "context": 4096}).
* **Strategie di memoria adattiva:** Per modelli con finestra ridotta, **compattare il contesto** è essenziale. Alcune tecniche:  
  + **Riassumere la cronologia**: dopo un tot di interazioni, produci un riassunto del dialogo e sostituisci i dettagli precedenti con tale riassunto nella prompt. ChatGPT e Claude internamente fanno qualcosa di simile, mantenendo un “indice di conversazione” ([Best Practices for Managing LLM Context Memory & Minimizing Token Usage in Long-Response Applications? : r/LLMDevs](https://www.reddit.com/r/LLMDevs/comments/1fcwq1f/best_practices_for_managing_llm_context_memory/#:~:text=%E2%80%A2)). Ad esempio, potresti avere una variabile long\_term\_memory in cui accumuli un riassunto progressivo delle vecchie turnazioni. Quando il contesto supera X token, aggiorni il riassunto e rimuovi i messaggi meno recenti.
  + **Memoria vettoriale (RAG)**: per informazioni importanti ma troppo corpose, estrai embedding e salva in un vettore database. Al bisogno, effettua una ricerca di similarità per recuperare le parti rilevanti e includile di nuovo nel prompt (magari condensate). Questo consente di **non tenere tutto sempre nel prompt**, ma solo ciò che serve per la query corrente.
  + **Eliminare ridondanze e meta-testo**: assicurati di non inviare ogni volta istruzioni ripetute. Ad esempio, se hai un lungo system prompt con regole, mantienilo fisso e non farlo crescere. Oppure, se l’utente dà sempre contesto uguale (es. descrizione di un progetto), valuta di metterlo in un file e farlo accedere via tool quando serve, invece di inserirlo ad ogni turno.
  + **Strutturare il prompt**: modelli orientati al codice spesso preferiscono input concisi. Mantieni magari separate sezioni (es: “Codice utente”, “Errore compilazione”, “Messaggio precedente”) e rimuovi conversazioni fuori contesto. Se il modello non è bravo nel small-talk, evitare di includere messaggi non correlati al codice.

Un esempio di approccio ibrido: potresti tenere attiva una *memoria a breve termine* (ultimi N scambi) e una *memoria a lungo termine* riassuntiva. Per modelli come DeepSeek, potresti **disabilitare la memoria a lungo termine** (non include un lungo riassunto, tanto si focalizza sul compito corrente) e limitarti alla breve (per coerenza con l’ultima domanda). Per un modello GPT-4, puoi permetterti di allegare sia la cronologia (anche 10-20 messaggi) sia magari note pertinenti dal passato.

Esistono strumenti e librerie che facilitano questo. Ad esempio, **LangChain** offre una classe di *memory* modulare dove puoi impostare una memoria di conversazione con window slinding (ultimi k tokens) o summary memory. Un’altra libreria, **MemoryGPT** (sperimentale), crea un indice vettoriale della chat. In mancanza di tool automatici, implementare manualmente con Python è fattibile: puoi usare tiktoken (per modelli OpenAI) o funzioni Huggingface per stimare il count di token e decidere quando troncare. Anche **OpenAI ChatCompletion API** restituisce spesso un token count; se fai routing su quella logica potresti usarla.

Riguardo DeepSeek in particolare: essendo un modello focalizzato sul codice, è probabile che abbia meno capacità di seguire contesti conversazionali lunghissimi. Quindi come *best practice* con modelli di coding:

* Mantieni il prompt il più **autonomo e specifico** possibile: includi esattamente il codice su cui lavorare e la richiesta, ed eventualmente brevi spiegazioni precedenti. Evita di lasciargli dedurre da conversazioni molto risalenti cosa fare.
* Se hai bisogno di contestualizzare con molta documentazione (es. più file), valuta di attivare quella funzionalità solo quando utilizzi un modello con contesto ampio (Claude 100k) o di suddividere il lavoro in parti per il modello locale.

In conclusione, **adatta la memoria al modello** usando impostazioni condizionali e tecniche di compressione del contesto. Questo garantirà risposte più pertinenti e ridurrà il rischio di sforare la finestra massima. Come discusso dalla community, anche servizi come ChatGPT/Claude usano riassunti e indici per non superare i limiti di token ([Best Practices for Managing LLM Context Memory & Minimizing Token Usage in Long-Response Applications? : r/LLMDevs](https://www.reddit.com/r/LLMDevs/comments/1fcwq1f/best_practices_for_managing_llm_context_memory/#:~:text=%E2%80%A2)), quindi seguire approcci simili nel tuo tool è consigliabile.

## 4. Supporto multi-modello e routing intelligente con MCP

In uno scenario avanzato, potresti voler usare **più modelli LLM** diversi nella stessa estensione: ad esempio il tuo sistema potrebbe avere accesso a Claude o GPT-4 via cloud, e a DeepSeek-Coder localmente via LM Studio, e scegliere di volta in volta quello più adatto al compito. MCP in sé è focalizzato sulla fornitura di **contesto e strumenti** ai modelli, mentre il **routing tra modelli** è qualcosa che spetta alla logica applicativa. Tuttavia, MCP può essere un **abilitatore** in questo senso: poiché standardizza l’interfacciamento con dati e tool, puoi riutilizzare gli stessi tool con qualunque modello. Così, indipendentemente che la query vada a Claude o a DeepSeek, potranno accedere agli stessi MCP server (file system, documentazione, etc.) allo stesso modo. Questo uniforma le capacità accessorie dei modelli.

**Instradamento tra modelli:** Puoi implementare un meccanismo di *routing* che analizza la richiesta utente e decide quale modello impiegare. Ad esempio:

* In base al **tipo di domanda**: se rilevi che l’utente chiede di generare codice C#, potresti invocare DeepSeek-Coder (specializzato in codice). Se invece chiede spiegazioni lunghe o sintesi di documenti, magari preferisci Claude che regge contesti lunghi.
* In base al **costo/velocità**: per domande semplici, usa il modello locale (gratuito e immediato), per questioni complesse offri l’opzione di un modello cloud più potente.
* In base alla **lunghezza del contesto necessario**: se l’utente ha selezionato 5 file da analizzare insieme e il totale supera i 4K token, un modello locale piccolo non basterà – qui il router potrebbe scegliere un modello con finestra maggiore.
* Con un **prompt di controllo**: esistono approcci dove un “modellino” o una logica valuta la query. Ad esempio, potresti utilizzare un modello di classificazione (anche una semplice regex o parola chiave) che categorizza il task come coding, planning, writing ecc., e mappa ciascuna categoria a un modello preferito.

Questo processo può essere codificato manualmente (if/else sul contenuto della domanda) oppure usando tool più sofisticati. *LangChain* ad esempio offre una catena chiamata **RouterChain** in cui un piccolo LLM legge la domanda e sceglie quale esperto (chain) usare. Potresti configurare delle prompt in cui un GPT-3.5 decide se “inoltrare a coderLLM o a chatLLM”. Dato che vuoi restare locale/open, puoi implementare qualcosa di simile con un modello leggero o regole fisse.

**Capacità specifiche via prompt (tool, memory, ruolo):** Quando usi modelli diversi, è utile comunicare a ciascun modello *chi è e cosa può fare*. In pratica, impostare il **prompt di sistema** adeguatamente:

* **Ruolo**: definisci nel prompt del modello la sua identità (es. “Sei DeveloperAgent, un assistente specializzato in C#. Rispondi con codice e spiegazioni tecniche.”). Questo aiuta a differenziare i comportamenti se usi più modelli in parallelo con ruoli diversi.
* **Tool disponibili**: attraverso MCP, hai la lista di strumenti e il modello deve sapere come invocarli. Se il modello supporta function calling, fornisci le funzioni (con nome, descrizione e schema parametri) direttamente nell’API call. Se non lo supporta, includi nel prompt istruzioni tipo: “Puoi utilizzare i seguenti strumenti quando necessario: 1) search(query) per cercare informazioni, 2) open\_file(path) per leggere file. Per usare uno strumento, rispondi con la sintassi: <ToolName>(parametri) e niente altro.”. Questo è un esempio di come via prompt si possono descrivere **capabilities**. Il protocollo MCP stesso non forza un formato di prompt, ma suggerisce di usare schemi consistenti. Spesso, un approccio è utilizzare uno **schema JSON**: ad esempio, dire al modello “quando vuoi usare uno strumento, restituisci un oggetto JSON con {\"jsonrpc\": \"2.0\", \"method\": \"nome\_tool\", \"params\": {...}}”. Così il client MCP può riconoscerlo facilmente ([A quick look at MCP with Large Language Models and Node.js](https://developers.redhat.com/blog/2025/01/22/quick-look-mcp-large-language-models-and-nodejs#:~:text=A%20quick%20look%20at%20MCP,support%20function%2Ftool%20calling%20and)).
* **Memoria**: se un modello dispone di una memoria esterna (ad es. un vettore di conoscenza), potresti anche considerare di renderglielo noto. In genere però la memory bank viene gestita dal sistema (pre-filtro dei documenti rilevanti) più che come strumento invocabile dal modello. Comunque, potresti avere un tool MCP tipo recall(query) che cerca nei tuoi appunti passati (memoria a lungo termine) e restituisce qualcosa; il modello potrebbe chiamarlo se addestrato a farlo.

MCP facilita la definizione di tali capacità perché ogni tool viene esposto con uno schema formale (nome, parametri, documentazione) ([How to Use Anthropic MCP Tools with Your AutoGen Agents (and any model)](https://newsletter.victordibia.com/p/how-to-use-mcp-anthropic-mcp-tools#:~:text=2,fetch)). Alcuni SDK MCP possono automaticamente generare uno **schema OpenAPI/JSON Schema** dalle funzioni, che puoi inserire nel prompt di sistema. Questo toglie a te la fatica di scrivere descrizioni manuali per ogni funzione. Ad esempio, il TypeScript SDK usa *Zod* per creare schemi che rappresentano esattamente i parametri attesi dal tool ([Model Context Protocol: TypeScript SDKs for the Agentic AI ecosystem](https://www.speakeasy.com/post/release-model-context-protocol#:~:text=Model%20Context%20Protocol%3A%20TypeScript%20SDKs,picture%20of%20the%20request%20format)). Integrazioni come mcp\_server\_tools di AutoGen hanno mostrato che si può convertire quei schemi direttamente in funzioni chiamabili dal modello ([How to Use Anthropic MCP Tools with Your AutoGen Agents (and any model)](https://newsletter.victordibia.com/p/how-to-use-mcp-anthropic-mcp-tools#:~:text=2,fetch)). Quindi definire le *capabilities via prompt* è un passaggio di “prompt engineering” combinato con le specifiche fornite dall’SDK MCP.

**Selezione automatica del modello “giusto”:** Non c’è una feature out-of-the-box di MCP che faccia il modello switching, poiché MCP si occupa di dati di contesto. Tuttavia, potresti usare MCP indirettamente anche qui: per esempio, un *MCP server* potrebbe essere un wrapper di un modello! Immagina di avere un server MCP chiamato “ClaudeServer” con un tool ask\_claude(prompt) -> answer. Il tuo agente locale (es. DeepSeek) potrebbe decidere di chiamare quel tool per delegare una domanda complessa a Claude e poi presentare la risposta. In pratica stai usando un modello come “oracolo esterno” tramite MCP. Non è l’uso più comune, ma concettualmente possibile (è simile a come un agente ReAct potrebbe usare una calcolatrice: qui la “calcolatrice” è un altro LLM). Altrimenti, la strada più semplice è far sì che il **coordinatore** scelga il modello in anticipo e poi attivi gli strumenti MCP per quel modello. Ad esempio: if user\_query.length > 1000: use Claude; else if "codice" in query: use DeepSeek; else: use GPT-4. Dopo aver scelto, la pipeline procede con uno dei modelli mentre MCP resta disponibile a contorno.

Ci sono alcuni limiti e complessità da tenere presenti:

* **Gestione delle identità e risposte**: con multi-modello e multi-agente, assicurati di mantenere chi ha detto cosa. Ogni modello avrà il suo stile; quando componi insieme, evita conflitti (es. due modelli che rispondono contemporaneamente). Può aiutare avere un turno ben definito o un supervisore come detto.
* **Overhead**: aggiungere MCP comporta orchestrazione aggiuntiva. Ogni chiamata tool è un round-trip extra. Su modelli locali molto veloci non è un problema, ma sulle API cloud considera la latenza.
* **Compatibilità**: non tutti i modelli open source supportano nativamente il function calling. Quindi, modelli come DeepSeek dovrai probabilmente guidarli con prompt speciali per usare MCP (a meno che non li affianchi con un agente esterno). Modelli come GPT-4 invece lo fanno nativamente, ma potresti non voler mandare i tuoi dati locali su cloud per questioni privacy – quindi magari useresti GPT-4 solo per compiti senza necessità di leggere file locali.
* **Sicurezza**: MCP dà accesso a risorse potenti (file system, esecuzione comandi). Quando integri questo in VS Code, considera di mettere paletti: es. chiedere conferma all’utente prima che l’AI esegua certi tool (la spec MCP prevede conferma opzionale dei tool da parte dell’utente) ([Model Context Protocol (MCP) Client Development Guide - GitHub](https://github.com/cyanheads/model-context-protocol-resources/blob/main/guides/mcp-client-development-guide.md#:~:text=Model%20Context%20Protocol%20,prompts%20as%20your%20application%20evolves)). Soprattutto se userai MCP con agenti autonomi, prevenire effetti indesiderati è cruciale.

In conclusione, **MCP** è un protocollo versatile che puoi integrare nella tua estensione per fornire contesto multi-file e strumenti al tuo AI, sia locale che cloud. Ti permette di standardizzare l’accesso ai dati (file, API) in modo indipendente dal modello utilizzato ([MCP – Model Context Protocol: Standardizing AI-Data Access - DEV Community](https://dev.to/sreeni5018/mcp-model-context-protocol-standardizing-ai-data-access-465#:~:text=Anthropic%27s%20Model%20Context%20Protocol%20,with%20a%20single%2C%20standardized%20protocol)). Può essere usato sia con UI WebView (tramite backend Node/Python) su VS Code, sia con orchestrazioni multi-agente complesse, restando locale-friendly. Assicurati di adattare la gestione del **contesto e memoria** a ciascun modello per ottenere il massimo da ognuno, e progetta la logica multi-modello/agente in modo che collaborino senza conflitto. Con riferimenti ufficiali e SDK disponibili (vedi la [documentazione ufficiale MCP][2], i repository GitHub e integrazioni in progetti esistenti), hai a disposizione gli strumenti per costruire un sistema AI open-source, modulare e potente basato su VS Code, MCP e LLM sia locali che cloud.

**Fonti:**

* Anthropic, *Introducing the Model Context Protocol (MCP)* – **Anthropic Blog** ([Introduction - Model Context Protocol](https://modelcontextprotocol.io/introduction#:~:text=MCP%20is%20an%20open%20protocol,different%20data%20sources%20and%20tools)) ([MCP – Model Context Protocol: Standardizing AI-Data Access - DEV Community](https://dev.to/sreeni5018/mcp-model-context-protocol-standardizing-ai-data-access-465#:~:text=Anthropic%27s%20Model%20Context%20Protocol%20,with%20a%20single%2C%20standardized%20protocol))
* *Model Context Protocol – Documentation & SDK* (sito ufficiale) ([MCP – Model Context Protocol: Standardizing AI-Data Access - DEV Community](https://dev.to/sreeni5018/mcp-model-context-protocol-standardizing-ai-data-access-465#:~:text=MCP%20supports%20two%20transport%20models%2C,0%20for%20structured%20data%20exchange)) ([Overview - Model Context Protocol](https://modelcontextprotocol.io/sdk/java/mcp-overview#:~:text=,SDK%20%2A%20Specification))
* Dev.to, *MCP – Standardizing AI-Data Access* (S. Ramadurai, 2024) ([MCP – Model Context Protocol: Standardizing AI-Data Access - DEV Community](https://dev.to/sreeni5018/mcp-model-context-protocol-standardizing-ai-data-access-465#:~:text=MCP%20supports%20two%20transport%20models%2C,0%20for%20structured%20data%20exchange)) ([MCP – Model Context Protocol: Standardizing AI-Data Access - DEV Community](https://dev.to/sreeni5018/mcp-model-context-protocol-standardizing-ai-data-access-465#:~:text=servers%2C%20maintaining%20one,calculations%20or%20user%20profile%20management))
* Dev.to, *GitHub workflow with Cline 3.0 and MCP* (S. Lal, 2024) ([Speeding up your GitHub workflow with Cline 3.0 and MCP - DEV Community](https://dev.to/shannonlal/speeding-up-your-github-workflow-with-cline-30-and-mcp-j6#:~:text=GitHub%20MCP%20is%20a%20powerful,to%20supercharge%20your%20development%20process)) ([Speeding up your GitHub workflow with Cline 3.0 and MCP - DEV Community](https://dev.to/shannonlal/speeding-up-your-github-workflow-with-cline-30-and-mcp-j6#:~:text=However%2C%20it%27s%20not%20without%20challenges,can%20occasionally%20be%20a%20struggle))
* Victor Dibia, *Using Anthropic MCP Tools with AutoGen Agents* (2023) ([How to Use Anthropic MCP Tools with Your AutoGen Agents (and any model)](https://newsletter.victordibia.com/p/how-to-use-mcp-anthropic-mcp-tools#:~:text=2,fetch)) ([How to Use Anthropic MCP Tools with Your AutoGen Agents (and any model)](https://newsletter.victordibia.com/p/how-to-use-mcp-anthropic-mcp-tools#:~:text=,client%20that%20supports%20function%20calling))
* Medium, *Using MCP with a Local LLM* (A. Hathibelagal, 2025) ([Using the Model Context Protocol (MCP) With a Local LLM | by Ashraff Hathibelagal | Predict | Mar, 2025 | Medium](https://medium.com/predict/using-the-model-context-protocol-mcp-with-a-local-llm-e398d6f318c3#:~:text=Any%20large%20language%20model%20that,make%20use%20of%20those%20tools)) (esempio di server locale con tool)
* CrewAI Community Forum – *Loading Multiple LLMs with LM Studio* ([Loading Multiple LLM for multiple Agent - CrewAI Community Support - CrewAI](https://community.crewai.com/t/loading-multiple-llm-for-multiple-agent/3613#:~:text=try%3A%20llm_llama%20%3D%20LLM%28model%3D%22openai%2Fmeta,mlx%22%2C%20base_url%3D%22http%3A%2F%2Flocalhost%3A1234%2Fv1%22%2C%20api_key%3D%22fsdf)) (esempio multi-modello con LM Studio)
* Reddit r/LLMDevs – *Managing LLM Context Memory* (discussione 2023) ([Best Practices for Managing LLM Context Memory & Minimizing Token Usage in Long-Response Applications? : r/LLMDevs](https://www.reddit.com/r/LLMDevs/comments/1fcwq1f/best_practices_for_managing_llm_context_memory/#:~:text=%E2%80%A2)) (best practice su riassunti e memory)