Perfetto, mi metto subito al lavoro!

Realizzerò un report approfondito su come integrare il Model Context Protocol (MCP) di Anthropic nel progetto AI Developer Panel basato su Cline, con particolare attenzione all’ambiente locale (DeepSeekCoder via LM Studio) e compatibilità con altri protocolli (OpenAI Tool Use, LangGraph, CrewAI). Il report includerà:

* Introduzione tecnica al MCP
* Use case reali nel contesto del tuo progetto (inclusi Telegram e .cs live editing)
* Step di implementazione tecnica per WebView (React + TypeScript) e backend (Python + Node.js)
* Confronto con protocolli alternativi e possibili combinazioni
* Struttura JSON proposta per la comunicazione contestuale
* Considerazioni su sicurezza, performance e compatibilità
* Guida per sviluppatori (ITA + EN)
* Report PDF da salvare in docs/prompts/
* Mappa architetturale (Mermaid + PNG)

Ti aggiorno appena è tutto pronto!

# **Implementazione del Model Context Protocol (MCP) nell'AI Developer Panel / *Implementing the Model Context Protocol (MCP) in the AI Developer Panel***

## **Introduzione e Contesto del Progetto / *Introduction & Project Context***

L’**AI Developer Panel** è un’estensione avanzata per Visual Studio Code, basata su Cline, che funge da assistente AI multi-agente durante lo sviluppo software. Il pannello include agenti AI con ruoli specifici – uno *sviluppatore* che scrive codice, un *supervisore* che controlla/guida il processo e un *tester* che verifica il codice – collaborando tra loro. L’estensione supporta backend LLM sia locali che remoti (ad es. modelli eseguiti in locale tramite LM Studio con il modello *DeepSeekCoder* o tramite Ollama, nonché modelli cloud come Claude di Anthropic o OpenAI GPT). Attualmente, il pannello offre funzionalità come: editing di file C# (.cs), conservazione della cronologia di chat e contesto, esecuzione di compilazione C# e test, gestione avanzata dei prompt con injection di documentazione e supporto per uno storico di conversazione, e integrazione con **Telegram** per permettere interazioni con l’AI anche fuori dall’IDE (un canale di chat remoto). La configurazione generale risiede in un file JSON (es: config/config.json), dove sono definiti i provider di modelli e varie opzioni.

**Obiettivo:** Integrare il *Model Context Protocol (MCP)* di Anthropic all’interno di questo pannello AI, per migliorare l’accesso dinamico al contesto e agli strumenti esterni da parte degli agenti AI. In particolare, l’integrazione MCP mira a consentire: **(1)** l’accesso in tempo reale da parte degli agenti a repository di codice e documentazione esterni (come basi di codice esterne, knowledge base, API, ecc.), **(2)** la connessione e coordinazione tra agenti locali e remoti (ad esempio un agente nel pannello VS Code che comunica con un agente su Telegram o utilizza servizi remoti), e **(3)** l’invio efficiente del contesto dei file aperti in VS Code direttamente ai modelli AI tramite MCP (anziché includere interamente il testo dei file nei prompt, sfruttando invece richieste contestuali on-demand). In sintesi, MCP dovrebbe fungere da “collante” standard tra il pannello AI e fonti di informazione esterne, migliorando la cooperazione multi-agente e la rilevanza delle risposte AI.

Nel seguito presentiamo un’analisi tecnica e architetturale dettagliata su come implementare MCP nel progetto, seguita da linee guida applicative pratiche per gli sviluppatori (in entrambe le lingue, italiano e inglese). Viene inoltre fornita una **mappa architetturale** in formato diagramma (vedi sezione dedicata) e un confronto con approcci alternativi (OpenAI Tools, LangChain/LangGraph, AutoGen, CrewAI). Infine, proponiamo una struttura JSON standardizzata per orchestrare la comunicazione tra gli agenti e le loro fonti di contesto.

\*The **AI Developer Panel** is an advanced Visual Studio Code extension (based on Cline) that serves as a multi-agent AI assistant for software development. It includes AI agents with specific roles – a *developer* agent writing code, a *supervisor* agent overseeing and guiding, and a *tester* agent verifying code – all working collaboratively. The extension supports both local and remote LLM backends (e.g. local models via LM Studio running the *DeepSeekCoder* model, or via Ollama, as well as cloud models like Anthropic’s Claude or OpenAI’s GPT). Currently, the panel provides features such as: editing of C# source files (.cs), maintaining file edit history and context, executing C# builds and tests, advanced prompt management with documentation injection and conversational memory, and integration with **Telegram** to enable AI interactions outside the IDE (i.e. a remote chat channel). Configuration is managed through a JSON file (e.g. config/config.json), which defines model providers and various options.

**Goal:** Integrate Anthropic’s *Model Context Protocol (MCP)* into this AI panel to enhance the agents’ ability to dynamically access context and external tools. In particular, MCP integration should enable: **(1)** real-time access by the agents to external code repositories and documentation (such as external codebases, knowledge bases, APIs), **(2)** connectivity and coordination between local agents and remote ones (for example, an agent in the VS Code panel communicating with an agent via Telegram or utilizing remote services), and **(3)** efficient delivery of VS Code file context to the AI models via MCP (instead of always injecting entire file contents into prompts, leveraging on-demand contextual requests). In essence, MCP will act as a standardized “glue” between the AI panel and external information sources, improving multi-agent collaboration and the relevance of AI responses.

The following sections provide a detailed technical and architectural analysis of how to implement MCP in the project, along with practical developer guidelines (presented in both Italian and English). We include an **architectural diagram** illustrating the design, and a comparison with alternative approaches (OpenAI Tools, LangChain/LangGraph, AutoGen, CrewAI). Finally, we propose a standard JSON structure to orchestrate communication between the agents and their context sources.\*

## **Che cos’è il Model Context Protocol (MCP)? / *What is the Model Context Protocol (MCP)?***

Il **Model Context Protocol (MCP)** di Anthropic è uno standard aperto pensato per uniformare il modo in cui le applicazioni forniscono contesto e strumenti ai modelli linguistici (LLM). In altre parole, MCP definisce un’interfaccia comune attraverso cui un modello può richiedere e ottenere dati o eseguire azioni esterne in modo sicuro e strutturato. È spesso descritto con l’analogia di una “porta USB-C per le applicazioni AI” – così come USB-C fornisce un connettore universale per diversi dispositivi, MCP funge da interfaccia unificata per collegare i modelli AI a svariate fonti di dati e tool esterni ([Introduction - Model Context Protocol](https://modelcontextprotocol.io/introduction#:~:text=MCP%20is%20an%20open%20protocol,different%20data%20sources%20and%20tools)) ([Introduction - Model Context Protocol](https://modelcontextprotocol.io/introduction#:~:text=LLMs,different%20data%20sources%20and%20tools)).

In concreto, MCP adotta un’architettura **client-server**: da un lato c’è un’applicazione host (ad esempio Claude Desktop, un IDE come VS Code con Cline, o altri strumenti AI) che funge da *MCP host*, dall’altro ci sono uno o più *MCP server* esterni che espongono funzionalità o dati ([Introduction - Model Context Protocol](https://modelcontextprotocol.io/introduction#:~:text=At%20its%20core%2C%20MCP%20follows,can%20connect%20to%20multiple%20servers)). Il pannello AI agisce da host (cliente MCP) e può connettersi a diversi server MCP in parallelo. Ogni **MCP server** è un piccolo programma (potenzialmente eseguito localmente o in remoto) che offre uno specifico insieme di funzionalità al modello – ad esempio accesso al file system, interrogazioni a database, chiamate API web, ricerca tra la documentazione, memorizzazione di conoscenza, ecc. – il tutto attraverso endpoint standardizzati. In pratica, un MCP server si comporta come un’API che il modello può utilizzare indirettamente ([MCP Overview | Cline](https://docs.cline.bot/mcp-servers/mcp#:~:text=MCP%20servers%20act%20as%20intermediaries,that%20an%20LLM%20can%20use)). I server MCP definiscono **tool** e **risorse** disponibili: i *tool* sono operazioni eseguibili (funzioni che possono modificare lo stato o effettuare calcoli/azioni), mentre le *risorse* sono dati accessibili in sola lettura (simili a file, documenti o query predefinite) ([MCP Overview | Cline](https://docs.cline.bot/mcp-servers/mcp#:~:text=MCP%20servers%20define%20a%20set,a%20wide%20range%20of%20capabilities)).

Il flusso di interazione tipico è il seguente ([MCP Overview | Cline](https://docs.cline.bot/mcp-servers/mcp#:~:text=Here%27s%20how%20MCP%20works%3A)):

* L’host (il pannello AI) scopre le capacità dei server MCP connessi, ottenendo la lista dei tool e delle risorse che essi mettono a disposizione (tramite chiamate come tools/list e resources/list).
* Durante una conversazione o attività, il modello LLM può decidere di utilizzare un tool o accedere a una risorsa. Nel contesto di MCP, idealmente il modello **genera una richiesta strutturata** indicando quale tool invocare e con quali parametri.
* Il client MCP (integrato nell’host) cattura questa richiesta e la invia al server MCP appropriato (ad esempio, un comando JSON-RPC tools/call verso il server in questione) ([Function Calling vs. Model Context Protocol (MCP): What You Need to Know - DEV Community](https://dev.to/fotiecodes/function-calling-vs-model-context-protocol-mcp-what-you-need-to-know-4nbo#:~:text=,%7D)).
* Il server esegue l’azione richiesta (ad es. legge un file, esegue una query, invia un messaggio) in un ambiente isolato e sicuro, e restituisce il risultato al client in formato strutturato (spesso sotto forma di contenuto testuale o dati JSON).
* L’host quindi fornisce quel risultato al modello come parte della conversazione, permettendo al modello di utilizzarlo per generare la risposta finale all’utente.

MCP offre diversi **benefici chiave**:

* **Standardizzazione e Interoperabilità:** Il protocollo definisce formati unificati (basati su JSON-RPC) per richieste e risposte di tool/risorse, indipendenti dal vendor del modello ([Function Calling vs. Model Context Protocol (MCP): What You Need to Know - DEV Community](https://dev.to/fotiecodes/function-calling-vs-model-context-protocol-mcp-what-you-need-to-know-4nbo#:~:text=Purpose%20Converts%20user%20prompts%20into,Ensures%20interoperability%20across%20multiple%20tools)). Ciò significa che una volta implementato MCP, l’estensione potrà lavorare con vari modelli (Claude, OpenAI, modelli open-source) e vari tool, senza dover gestire formati proprietari diversi per ciascun caso.
* **Sicurezza e Controllo:** I server MCP operano isolando credenziali e dati sensibili. Ogni invocazione di tool può richiedere un’esplicita approvazione dell’utente, prevenendo che il modello esegua azioni non desiderate ([MCP Overview | Cline](https://docs.cline.bot/mcp-servers/mcp#:~:text=database%20queries)). Ad esempio, se il modello tenta di accedere a un database aziendale tramite MCP, l’utente può essere interpellato prima di concedere l’accesso. Questo è importante in un contesto IDE, per evitare modifiche distruttive al codice o fughe di dati.
* **Dati sempre aggiornati e contestuali:** Diversamente dal solo contesto statico inserito nel prompt, MCP consente di recuperare informazioni *on-demand*. Il modello può quindi interrogare in tempo reale una documentazione aggiornata, o ottenere l’ultima versione di un file di progetto, anche se questi non erano inizialmente nel prompt. In altre parole, MCP trasforma un LLM statico in un agente dinamico capace di **two-way integration** con l’ambiente circostante ([Introducing the Model Context Protocol - Anthropic](https://www.anthropic.com/news/model-context-protocol#:~:text=The%20Model%20Context%20Protocol%20is,powered%20tools)) (non solo l’utente fornisce input al modello, ma il modello può richiedere ulteriori input dal mondo esterno durante la sua elaborazione).

Possiamo paragonare il ruolo di MCP rispetto al classico *function calling* di OpenAI: la chiamata di funzione tradizionale (ad esempio la feature “functions” delle API OpenAI) è centrata sull’LLM che produce un invocazione strutturata a una funzione – in pratica è il modo in cui il modello “ordina un’azione” da compiere. MCP invece copre l’intero ciclo di esecuzione: definisce come queste richieste vengono trasmesse a tool effettivi, eseguite e restituite in modo consistente ([Function Calling vs. Model Context Protocol (MCP): What You Need to Know - DEV Community](https://dev.to/fotiecodes/function-calling-vs-model-context-protocol-mcp-what-you-need-to-know-4nbo#:~:text=Feature%20Function%20Calling%20MCP%20,Ensures)). In altri termini, *“il Function Calling riguarda il chiedere al modello di fare qualcosa, mentre MCP si occupa dell’esecuzione effettiva di quel qualcosa”* ([Function Calling vs. Model Context Protocol (MCP): What You Need to Know - DEV Community](https://dev.to/fotiecodes/function-calling-vs-model-context-protocol-mcp-what-you-need-to-know-4nbo#:~:text=Essentially%2C%20Function%20Calling%20is%20about,driven%20software%20automation%20runs%20efficiently)). Ciò rende MCP un componente fondamentale per costruire agenti avanzati su larga scala, dove serve garantire che i modelli possano usare strumenti in modo affidabile e replicabile.

***In sintesi***, adottando MCP nel pannello AI, doteremo i nostri agenti di un “canale universale” per interagire con file, servizi e tool esterni al volo, mantenendo il tutto coerente, controllato e compatibile con diversi modelli. Nel prossimo paragrafo vedremo come questo si traduce nell’architettura concreta dell’AI Developer Panel.

\*The **Model Context Protocol (MCP)** by Anthropic is an open standard designed to standardize how applications supply context and tools to language models (LLMs). In other words, MCP defines a common interface through which a model can request and obtain external data or perform external actions in a secure, structured way. It’s often described by analogy as a “USB-C port for AI applications” – just as USB-C offers a universal connector for many devices, MCP serves as a unified interface to connect AI models to all sorts of data sources and external tools ([Introduction - Model Context Protocol](https://modelcontextprotocol.io/introduction#:~:text=MCP%20is%20an%20open%20protocol,different%20data%20sources%20and%20tools)) ([Introduction - Model Context Protocol](https://modelcontextprotocol.io/introduction#:~:text=LLMs,different%20data%20sources%20and%20tools)).

In practice, MCP uses a **client-server** architecture: on one side is a host application (e.g. Claude Desktop, an IDE like VS Code with Cline, or other AI tools) acting as the *MCP host*, and on the other side are one or more external *MCP servers* that expose functionality or data ([Introduction - Model Context Protocol](https://modelcontextprotocol.io/introduction#:~:text=At%20its%20core%2C%20MCP%20follows,can%20connect%20to%20multiple%20servers)). Our AI panel will act as the host (MCP client) and can connect to multiple MCP servers in parallel. Each **MCP server** is a small program (running locally or remotely) that offers a specific set of capabilities to the model – for example, access to the file system, database queries, web API calls, documentation search, knowledge storage, etc. – all through standardized endpoints. Essentially, an MCP server behaves like an API that the model can use indirectly ([MCP Overview | Cline](https://docs.cline.bot/mcp-servers/mcp#:~:text=MCP%20servers%20act%20as%20intermediaries,that%20an%20LLM%20can%20use)). The MCP servers define available **tools** and **resources**: *tools* are executable actions (functions that may modify state or perform computations/actions), whereas *resources* are read-only data accesses (akin to files, documents, or predefined queries) ([MCP Overview | Cline](https://docs.cline.bot/mcp-servers/mcp#:~:text=MCP%20servers%20define%20a%20set,a%20wide%20range%20of%20capabilities)).

The typical interaction flow is as follows ([MCP Overview | Cline](https://docs.cline.bot/mcp-servers/mcp#:~:text=Here%27s%20how%20MCP%20works%3A)):

* The host (the AI panel) discovers the capabilities of connected MCP servers, retrieving the list of tools and resources they provide (via calls like tools/list and resources/list).
* During a conversation or task, the LLM may decide it needs to use a tool or fetch a resource. In an MCP-enabled setup, the model ideally **produces a structured request** indicating which tool to invoke and with what parameters.
* The MCP client (built into the host) intercepts this request and forwards it to the appropriate MCP server (e.g. sending a JSON-RPC tools/call request to that server) ([Function Calling vs. Model Context Protocol (MCP): What You Need to Know - DEV Community](https://dev.to/fotiecodes/function-calling-vs-model-context-protocol-mcp-what-you-need-to-know-4nbo#:~:text=,%7D)).
* The server executes the requested action (e.g. reads a file, runs a query, sends a message) in an isolated, secure environment, then returns the result in a structured format (often as text content or JSON data).
* The host then supplies that result back to the model as part of the conversation, allowing the model to use it to produce its final answer to the user.

MCP provides several **key benefits**:

* **Standardization and Interoperability:** The protocol defines unified formats (JSON-RPC based) for tool/resource requests and responses, independent of any particular model vendor ([Function Calling vs. Model Context Protocol (MCP): What You Need to Know - DEV Community](https://dev.to/fotiecodes/function-calling-vs-model-context-protocol-mcp-what-you-need-to-know-4nbo#:~:text=Purpose%20Converts%20user%20prompts%20into,Ensures%20interoperability%20across%20multiple%20tools)). This means once we implement MCP, the extension can work with various models (Claude, OpenAI, open-source LLMs) and various tools without handling different proprietary formats for each case.
* **Security and Control:** MCP servers operate in a way that isolates credentials and sensitive data. Each tool invocation can require explicit user approval, preventing the model from performing unintended actions ([MCP Overview | Cline](https://docs.cline.bot/mcp-servers/mcp#:~:text=database%20queries)). For example, if the model tries to access a company database via MCP, the user can be prompted before granting access. This is important in an IDE context to avoid destructive code changes or data leaks.
* **Up-to-date, Contextual Data:** Unlike static context that is all pre-injected into the prompt, MCP allows retrieving information *on demand*. The model can query documentation in real time or get the latest version of a project file, even if those were not initially in the prompt. In other words, MCP turns a static LLM into a dynamic agent capable of **two-way integration** with the surrounding environment ([Introducing the Model Context Protocol - Anthropic](https://www.anthropic.com/news/model-context-protocol#:~:text=The%20Model%20Context%20Protocol%20is,powered%20tools)) – not only does the user give input to the model, but the model can request additional input from the outside world during its reasoning.

We can compare MCP’s role to classical OpenAI-style *function calling*: a standard function-call feature (e.g. OpenAI API “functions”) centers on the LLM producing a structured invocation of a function – it’s essentially how the model “orders a task to be done.” MCP, by contrast, covers the entire execution cycle: it defines how those requests are transmitted to actual tools, executed, and returned consistently ([Function Calling vs. Model Context Protocol (MCP): What You Need to Know - DEV Community](https://dev.to/fotiecodes/function-calling-vs-model-context-protocol-mcp-what-you-need-to-know-4nbo#:~:text=Feature%20Function%20Calling%20MCP%20,Ensures)). In other words, *“Function Calling is about ordering the task, while MCP is responsible for executing the task”* ([Function Calling vs. Model Context Protocol (MCP): What You Need to Know - DEV Community](https://dev.to/fotiecodes/function-calling-vs-model-context-protocol-mcp-what-you-need-to-know-4nbo#:~:text=Essentially%2C%20Function%20Calling%20is%20about,driven%20software%20automation%20runs%20efficiently)). This makes MCP crucial for building advanced, scalable agents where we need to ensure models can use tools reliably and repeatably.

***In summary***, by adopting MCP in the AI panel, we equip our agents with a “universal channel” to interact with files, services, and external tools on the fly – all in a consistent, controlled manner that’s compatible with different models. In the next section, we’ll see how this translates into the concrete architecture of the AI Developer Panel.\*

## **Architettura di Integrazione MCP nell’AI Developer Panel / *MCP Integration Architecture in the AI Developer Panel***

Per integrare MCP nel progetto, dobbiamo estendere l’architettura esistente del pannello AI in modo che gli agenti possano sfruttare i server MCP mantenendo la priorità sulle soluzioni **locali**. La figura seguente illustra a alto livello la nuova architettura con MCP:

flowchart LR

subgraph VSCode Extension - AI Developer Panel

direction TB

Agents["Agenti AI (Sviluppatore, Supervisore, Tester)"]

ContextManager["Gestore Contesto & Prompt"]

TelegramInt["Integrazione Telegram"]

end

subgraph LLM Backends

LocalLM["Modello Locale<br/>(LM Studio / Ollama)"]

RemoteLM["Modello Remoto<br/>(Claude 3, OpenAI)"]

end

subgraph MCP Servers

FileSrv["Server MCP File System"]

DocSrv["Server MCP Documentazione"]

MemSrv["Server MCP Memoria"]

RepoSrv["Server MCP Repository"]

end

User(["Utente Sviluppatore"]) -->|Richieste<br/>nell'IDE| Agents

User -->|Chat Telegram| TelegramInt

TelegramInt -->|Prompt condivisi| Agents

Agents -->|richiede contesto| ContextManager

ContextManager -->|via MCP| FileSrv

ContextManager -->|via MCP| DocSrv

ContextManager -->|via MCP| MemSrv

ContextManager -->|via MCP| RepoSrv

Agents -->|query LLM| LocalLM

Agents -->|query LLM| RemoteLM

LocalLM -->|risposta| Agents

RemoteLM -->|risposta| Agents

flowchart LR

subgraph VSCode Extension - AI Dev Panel

direction TB

AgentsEng["AI Agents (Developer, Supervisor, Tester)"]

ContextEng["Context Manager & Prompt Handler"]

TelegramEng["Telegram Bot Integration"]

end

subgraph LLM Backends

LocalLLM["Local LLM<br/>(LM Studio / Ollama)"]

RemoteLLM["Remote LLM<br/>(Claude 3, OpenAI)"]

end

subgraph MCP Servers (Tools & Data)

FileServer["File System MCP Server"]

DocServer["Documentation MCP Server"]

MemoryServer["Memory MCP Server"]

RepoServer["Git/Repo MCP Server"]

end

DevUser(["Developer (User)"]) -->|IDE goal/prompt| AgentsEng

DevUser -->|Telegram chat| TelegramEng

TelegramEng -->|distributed prompt| AgentsEng

AgentsEng -->|request context| ContextEng

ContextEng -->|via MCP| FileServer

ContextEng -->|via MCP| DocServer

ContextEng -->|via MCP| MemoryServer

ContextEng -->|via MCP| RepoServer

AgentsEng -->|LLM query| LocalLLM

AgentsEng -->|LLM query| RemoteLLM

LocalLLM -->|LLM response| AgentsEng

RemoteLLM -->|LLM response| AgentsEng

*Nella mappa architetturale sopra, il pannello AI in VS Code (riquadro grigio) agisce come host MCP, collegato a vari server MCP (riquadro blu) e ai modelli LLM locali/remoti (riquadro giallo). L’utente interagisce tramite VS Code e Telegram. / The architecture diagram above shows the AI panel in VS Code (gray cluster) acting as the MCP host, connected to various MCP servers (blue cluster) and to local/remote LLMs (yellow cluster). The user interacts via VS Code and Telegram.*

Come mostrato, l’estensione VS Code (AI Developer Panel) comprende:

* **Agenti AI (Developer, Supervisor, Tester):** istanze logiche (o processi) che incarnano ruoli differenti ma cooperano sul medesimo compito. Ad esempio, di fronte a un *goal* dell’utente (“correggi un bug in FactionManager.cs”), l’agente *Developer* proporrà modifiche al codice, l’agente *Tester* potrà eseguire test unitari o analisi statiche, e l’agente *Supervisor* monitorerà il processo, eventualmente fornendo istruzioni aggiuntive o valutando le soluzioni. Questi agenti condividono il contesto e si scambiano messaggi tramite il gestore centrale, e ora – grazie a MCP – possono anche condividere l’accesso a strumenti esterni.
* **Gestore del Contesto & Prompt:** un modulo centrale (parte dell’estensione) che si occupa di raccogliere il contesto rilevante (file aperti, memoria conversazionale, documentazione pertinente, ecc.), assemblare i prompt per i modelli e instradare le richieste agli LLM o ai server MCP. Questo componente implementerà il client MCP: ovvero, userà l’SDK o le API MCP per connettersi ai vari server, inviare richieste e attendere risposte. Inoltre, è responsabile di mantenere allineati gli agenti con la stessa visione del contesto.
* **Integrazione Telegram:** un modulo che collega l’estensione a Telegram (probabilmente tramite le API Bot di Telegram). Questo consente di ricevere prompt o comandi da un utente via chat Telegram e inoltrarli agli agenti come se fossero input dell’utente, e viceversa di inviare le risposte degli agenti all’utente remoto. Nel diagramma, vediamo che l’utente può interagire sia dall’IDE direttamente sia via Telegram; in entrambi i casi i messaggi confluiscono agli agenti AI attraverso il gestore contesto. L’integrazione con Telegram è importante per permettere *distributed AI interaction*, ad esempio un supervisore umano può controllare l’agente via smartphone o gli agenti possono notificare l’utente su Telegram quando un lungo processo è completo. (Nota: Telegram non è gestito da MCP di per sé, essendo una via di input/output separata; tuttavia, potremmo anche implementare un *MCP server Telegram* dedicato per far sì che gli agenti possano autonomamente inviare messaggi Telegram come tool, vedere dettaglio più avanti).

**LLM Backends (Locali e Remoti):** rappresentano i modelli di linguaggio effettivi che generano testo e ragionamento. In linea con l’obiettivo di priorità locale, useremo preferenzialmente un modello locale caricabile via LM Studio o Ollama (ad esempio un modello Llama 2 o CodeLlama specializzato nel code-completion, come DeepSeekCoder se disponibile). Questo garantisce che buona parte delle operazioni (in particolare l’analisi del codice e le generazioni di risposta) avvengano in locale, preservando privacy e velocità. Tuttavia, l’architettura consente anche di utilizzare modelli remoti su richiesta: ad esempio Anthropic Claude (es. *claude-3-opus*) o OpenAI GPT-4, tramite le rispettive API cloud. Il sistema di orchestrazione dovrebbe essere agnostico rispetto al backend: il *Gestore Contesto* invierà prompt e riceverà risposte dal modello configurato (in base a config.json l’utente può selezionare il modello). La differenza chiave è che con alcuni modelli avanzati (es. GPT-4 con function calling, o Claude) la formattazione delle richieste MCP potrebbe essere semplificata (perché il modello è già predisposto a output strutturati), mentre con modelli open-source potrebbe servire più guida (prompt engineering) per far sì che producano correttamente le chiamate ai tool.

**MCP Servers (strumenti e dati):** questi sono servizi esterni che metteremo a disposizione dell’AI tramite MCP. Nel diagramma ne sono indicati quattro tipi, corrispondenti ai requisiti del progetto:

1. **File System Server:** permette di esplorare e leggere (o eventualmente anche modificare, se autorizzato) il file system locale del progetto. Questo è cruciale per dare accesso ai contenuti dei file di codice sorgente. Ad esempio, l’agente può utilizzare un tool read\_file esposto da questo server per ottenere il contenuto di FactionManager.cs su richiesta, invece di doverlo avere tutto in prompt sin dall’inizio. Un server MCP filesystem di riferimento è già disponibile open-source ([Example Servers - Model Context Protocol](https://modelcontextprotocol.io/examples#:~:text=Data%20and%20file%20systems)); esso supporta operazioni come elencare directory, leggere file con controllo di accesso, ecc. (Cline stesso integra un *MCP server Filesystem* di default in alcuni casi).
2. **Documentation Server:** fornisce accesso a documentazione esterna (documenti, guide, API references). A seconda della fonte, questo potrebbe implementarsi in vari modi. Se la documentazione è disponibile online (es. documentazione Microsoft .NET o altri framework), potremmo usare un server MCP di tipo *web fetch* o *search*. Ad esempio, esiste un MCP server *Brave Search* per eseguire ricerche web ([Example Servers - Model Context Protocol](https://modelcontextprotocol.io/examples#:~:text=%2A%20Brave%20Search%20,automation%20and%20web%20scraping%20capabilities)) e uno *Fetch* per recuperare contenuti web in modo pulito ([Example Servers - Model Context Protocol](https://modelcontextprotocol.io/examples#:~:text=%2A%20Brave%20Search%20,automation%20and%20web%20scraping%20capabilities)). In alternativa, se possediamo documentazione in locale (ad es. una cartella docs/ nel progetto, o manuali in PDF/Markdown), possiamo indicizzare quel corpus e accedervi tramite un server (magari usando un server MCP “Memory” come descritto di seguito o creando un apposito server per query).
3. **Memory Server:** un componente per la **memoria conversazionale estesa o conoscenza persistente**. Il pannello già mantiene uno storico di conversazione base, ma con MCP possiamo avere un *Memory Server* che funge da knowledge base a lungo termine, magari strutturato come un grafo di conoscenze o un vettore semantico. Nel repository ufficiale è presente un server “Memory” basato su grafi di conoscenza ([Example Servers - Model Context Protocol](https://modelcontextprotocol.io/examples#:~:text=%2A%20Google%20Maps%20,based%20persistent%20memory%20system)). Questo server potrebbe consentire di salvare e cercare informazioni contestuali (ad es. “che decisione è stata presa riguardo al design del modulo X?” oppure “qual è l’architettura del sistema?”) che persistono tra sessioni. Gli agenti (specialmente il Supervisore) potrebbero interrogare questa memoria per mantenere coerenza di alto livello.
4. **Repository/Git Server:** per accedere a repository esterni o funzioni di versioning. Se il progetto ha un repository Git (locale o remoto su GitHub/GitLab), un *MCP server Git* può permettere agli agenti di effettuare operazioni come cercare commit, analizzare diff, o aprire altri file del repository non aperti in editor ([Example Servers - Model Context Protocol](https://modelcontextprotocol.io/examples#:~:text=Development%20tools)). Ad esempio, se il bug in FactionManager.cs è dovuto a un cambiamento recente, l’agente può usare il tool git\_log o git\_diff per trovare il commit problematico. Esistono server predefiniti per Git, GitHub e GitLab ([Example Servers - Model Context Protocol](https://modelcontextprotocol.io/examples#:~:text=Development%20tools)).

Oltre a questi, potremmo considerare altri server MCP utili, come:

* **Test runner**: un server che esegue test o codice in sandbox (es. esiste E2B per eseguire codice in sandbox cloud ([Example Servers - Model Context Protocol](https://modelcontextprotocol.io/examples#:~:text=%2A%20Browserbase%20,the%20Qdrant%20vector%20search%20engine)), o un server Docker ([Example Servers - Model Context Protocol](https://modelcontextprotocol.io/examples#:~:text=A%20growing%20ecosystem%20of%20community,servers%20extends%20MCP%E2%80%99s%20capabilities))). Ciò può essere utile per far sì che l’agente Tester esegua realmente il progetto e catturi output/errori.
* **Telegram Server (ipotetico)**: non c’è (ancora) un server MCP per Telegram pubblico, ma potremmo crearne uno custom. Un *Telegram MCP server* potrebbe esporsi tool tipo send\_message o read\_updates per permettere agli agenti di inviare/leggere messaggi via Telegram come parte del loro flusso. In mancanza, questa funzione la gestiamo esternamente tramite il modulo integrazione Telegram, ma teniamo a mente che è possibile incapsularla in MCP per uniformità.

**Coordinamento locale vs remoto:** Un principio chiave dell’implementazione è che l’integrazione MCP avviene principalmente a livello locale, all’interno dell’estensione VS Code. Ciò significa che anche i server MCP idealmente gireranno localmente (o nella rete locale) per motivi di prestazioni e privacy. Fortunatamente, molti MCP server (file system, git, memory, etc.) sono pensati per essere eseguiti localmente accanto al client. Il pannello potrebbe avviare questi server come sottoprocessi all’avvio (o al bisogno) e comunicare con loro via **STDIO** – che è uno dei transport standard di MCP ([GitHub - modelcontextprotocol/typescript-sdk: The official Typescript SDK for Model Context Protocol servers and clients](https://github.com/modelcontextprotocol/typescript-sdk#:~:text=The%20Model%20Context%20Protocol%20allows,specification%2C%20making%20it%20easy%20to)) ([GitHub - modelcontextprotocol/typescript-sdk: The official Typescript SDK for Model Context Protocol servers and clients](https://github.com/modelcontextprotocol/typescript-sdk#:~:text=import%20,zod)). Ad esempio, il *TypeScript SDK* ufficiale supporta un StdioServerTransport per facilitare comunicazioni host-server via stdin/stdout ([GitHub - modelcontextprotocol/typescript-sdk: The official Typescript SDK for Model Context Protocol servers and clients](https://github.com/modelcontextprotocol/typescript-sdk#:~:text=import%20,zod)) ([GitHub - modelcontextprotocol/typescript-sdk: The official Typescript SDK for Model Context Protocol servers and clients](https://github.com/modelcontextprotocol/typescript-sdk#:~:text=%2F%2F%20Add%20an%20addition%20tool,)). Questo è molto utile nel contesto VS Code extension: possiamo includere o installare i server MCP come pacchetti NPM o binari, poi il client MCP (nel pannello) li lancia e dialoga con loro senza necessità di rete.

Allo stesso tempo, **supporto remoto**: se una determinata funzionalità richiede necessariamente un servizio remoto (ad es. accedere a un API online, compiere una ricerca web, o usare un modello AI remoto), MCP lo consente. Possiamo avere un MCP server che funge da *proxy* verso servizi remoti. Ad esempio, il server *Brave Search* richiede l’API di Brave e quindi una connessione internet; potremmo avviarlo comunque localmente ma esso farà chiamate esterne. L’architettura prevede che in questi casi i dati sensibili (API key) siano confinati nel server MCP stesso, che come detto è isolato e può essere autorizzato specificamente. In sintesi, la logica e il coordinamento restano locali, ma quando **MCP “lo richiede”**, si può coinvolgere un servizio remoto. Ciò soddisfa la condizione di dare priorità all’integrazione locale, riservandosi però la possibilità di attingere al cloud se e solo se necessario (ad es. usare Claude via API se il modello locale non è sufficiente per un compito, oppure interrogare un sito web per la documentazione). Il design modulare dei server MCP facilita questo: possiamo decidere quali server attivare in base alla modalità (offline/pure-local vs online).

### **Flusso di lavoro con MCP integrato**

Con l’architettura delineata, ecco come si svolgerebbe il flusso di lavoro quando l’utente utilizza il pannello AI per un compito (come *“correggi bug in FactionManager.cs”*):

1. **Input dell’utente:** L’utente formula la richiesta, ad esempio digitando in VS Code oppure inviando un messaggio su Telegram (“Trova e correggi il bug nel file *FactionManager.cs* che causa un errore NullReference”). Il messaggio arriva agli agenti AI.
2. **Raccolta contesto iniziale:** Il *Gestore Contesto* esamina la richiesta e determina quali fonti contestuali attivare. In base alla configurazione context\_sources (vedi sezione successiva per JSON), potrebbe ad esempio rilevare che serve il contenuto del file FactionManager.cs (*source:* “files”), lo storico delle interazioni recenti (*source:* “memory”), e eventuali note di documentazione sull’uso di *FactionManager* (*source:* “docs”).  
   * Senza MCP, tipicamente qui si sarebbe letto il file dal disco e inserito tutto il suo contenuto direttamente nel prompt. Con MCP, invece di iniettare l’intero file, possiamo inserire nel prompt un riferimento o un breve estratto e lasciare al modello la possibilità di chiedere dettagli via tool. Ad esempio, potremmo dire nel prompt iniziale qualcosa come: *“Il file FactionManager.cs è disponibile come risorsa file://project/FactionManager.cs (200 linee)”*. In aggiunta, se il bug genera un errore a runtime, il tester agent potrebbe avere log o stacktrace disponibili (anche questi caricati magari in memory server).
   * Il contesto iniziale fornito quindi può essere più leggero, includendo solo ciò che riteniamo strettamente necessario (es. firma di classi/metodi, messaggi di errore), sapendo che grazie a MCP il modello potrà approfondire chiamando tool.
3. **Coordinamento multi-agente:** L’agente *Developer* prende l’iniziativa sulla base del goal. Potrebbe analizzare la descrizione del bug e decidere di leggere il codice di FactionManager.cs per individuare la causa. Invece di doverlo già avere in prompt, l’agente (tramite il modello LLM) può formulare internamente una richiesta MCP: ad esempio il modello potrebbe generare un output strutturato o un trigger nel suo testo tipo: *“[Call: read\_file, Target: FactionManager.cs]”*. La sintassi esatta dipende da come alleniamo/istruiamo il modello (potrebbe essere un JSON o un semplice comando in markdown, a seconda delle convenzioni scelte), ma in ogni caso il *Gestore Contesto* riconosce l’intenzione del modello di invocare un tool.  
   * Il Gestore allora effettua la chiamata reale al server MCP File System: tools/call con nome tool read\_file e parametro path = FactionManager.cs ([Function Calling vs. Model Context Protocol (MCP): What You Need to Know - DEV Community](https://dev.to/fotiecodes/function-calling-vs-model-context-protocol-mcp-what-you-need-to-know-4nbo#:~:text=,%7D)). Il server riceve e legge il file dal disco (supponendo sia nel workspace VS Code) e restituisce il contenuto (o magari solo uno snippet se il protocollo supporta paginazione). Nota: conviene che il server supporti anche la possibilità di richiedere solo parti di file (es. range di righe) per efficienza, specialmente se i file sono grandi.
   * Una volta ottenuto il contenuto, il Gestore lo fornisce al modello. Spesso questo avviene inserendo la risposta come messaggio dell’assistente o come parte del prompt successivo.
   * L’agente Developer (cioè il modello) ora ha il codice e può individuare la riga col bug. Poniamo che scopra che c’è una variabile non inizializzata. A questo punto può direttamente proporre una correzione.
   * Il Tester agent potrebbe parallelamente, una volta disponibile il codice, eseguire mentalmente (o tramite un tool) il metodo incriminato con alcuni scenari. Se integrassimo un runner, potrebbe fare [Call: run\_tests, Target: FactionManagerTests]. Se non, può almeno analizzare staticamente.
   * Il Supervisor agent monitora queste azioni. Può intervenire se, ad esempio, l’agente Developer sta facendo una modifica dubbia. Il Supervisor potrebbe consultare la **Memory** per recuperare se in passato c’erano decisioni su come gestire certe entità Faction. Per farlo, può usare un tool tipo memory\_search fornito dal Memory Server, con query “Faction rules”. La risposta potrebbe ricordare che “le fazioni non vanno create se null” ecc., informazione che poi il Supervisor comunica al Developer agent per guidarlo.
4. **Proposta di soluzione e validazione:** L’agente Developer con tutte le informazioni propone la correzione al codice. Questo può concretizzarsi nel pannello con una diff o direttamente modificando il file (Cline di solito può applicare patch al file). Prima di applicare automaticamente, il *Supervisor* o l’utente umano potrebbe verificare. Il Tester agent potrebbe far girare test unitari (se automation), oppure almeno simulare gli effetti dopo patch.  
   * Se c’è integrazione con Git, potremmo anche automaticamente creare un commit con la fix, o almeno mostrare il diff e chiedere conferma.
   * L’utente infine vede la correzione suggerita. Se è soddisfatto, può applicarla.
5. **Iterazione tramite Telegram (opzionale):** Supponiamo che l’utente avesse lanciato la richiesta e poi si sia allontanato dall’IDE. Tramite la connessione Telegram, l’agente potrebbe inviare un messaggio al termine: “Ho trovato il bug e proposto una correzione, attendendo conferma.” L’utente su Telegram può rispondere “Applica la correzione e rifai il build”, questo input rientra nel loop e così via.

In questo scenario integrato, notiamo come MCP ha permesso agli agenti di:

* **Accedere al codice sorgente necessario on-demand**, senza dover inondare ogni prompt di tutto il repository. Ciò mantiene i prompt più concisi e sotto il limite di token del modello, demandando ai tool MCP l’accesso ai dettagli quando servono.
* **Consultare fonti esterne** (memoria di progetto, documentazione online) in modo interattivo. Se il modello avesse incertezze su un certo metodo o API, potrebbe chiamare un tool search\_docs che effettua una ricerca nella doc e ritorna risultati pertinenti, che poi il modello sintetizza.
* **Automatizzare azioni ripetitive**: come cercare nel repository tutti i riferimenti a una funzione, o eseguire test, il tutto con comandi standardizzati.

Per implementare tutto ciò, dal punto di vista dello sviluppatore, sono necessarie sia **componenti software** (client MCP nell’estensione, server MCP configurati) sia **impostazioni di configurazione** adeguate. Vediamo questi aspetti nel dettaglio nelle sottosezioni seguenti.

\*To integrate MCP into the project, we need to extend the existing architecture of the AI panel so that the agents can leverage MCP servers while maintaining a **local-first** approach. The figure above illustrates the high-level architecture after introducing MCP.

As shown, the VS Code extension (AI Developer Panel) consists of:

* **AI Agents (Developer, Supervisor, Tester):** logical (or process-based) instances that embody different roles but cooperate on the same task. For example, given a user’s goal (“fix a bug in FactionManager.cs”), the *Developer* agent will propose code changes, the *Tester* agent might run unit tests or static analysis, and the *Supervisor* agent oversees the process, potentially providing additional instructions or evaluating solutions. These agents share context and exchange messages via a central coordinator, and now – thanks to MCP – they can also share access to external tools.
* **Context Manager & Prompt Handler:** a central module (part of the extension) responsible for gathering relevant context (open files, conversation history, pertinent documentation, etc.), assembling prompts for the models, and routing requests either to LLMs or to MCP servers. This component will implement the MCP client logic: i.e., it will use the MCP SDK or API to connect to various servers, send requests, and handle responses. It’s also responsible for keeping the agents in sync with a unified view of context.
* **Telegram Integration:** a module linking the extension to Telegram (likely via Telegram’s bot API). This enables receiving prompts or commands from a user over a Telegram chat and forwarding them to the agents as if they were user inputs, and likewise sending agent responses back to the remote user. In the diagram, the user can interact from the IDE or via Telegram; in both cases, messages flow into the AI agents through the context manager. Telegram integration is important for allowing *distributed AI interaction* – e.g., a human supervisor might monitor the agent via smartphone, or agents might notify the user on Telegram when a long process completes. (Note: Telegram is not itself handled through MCP since it’s a separate I/O channel; however, we could implement a dedicated *Telegram MCP server* so that agents themselves could autonomously send Telegram messages as a tool – more on that later.)

**LLM Backends (Local & Remote):** these are the actual language model engines generating text and reasoning. Following the local-priority goal, we will prefer a local model loaded via LM Studio or Ollama (e.g. a Llama 2 or CodeLlama-based model specialized for coding, like DeepSeekCoder if available). This ensures most operations (especially code analysis and response generation) occur locally, preserving privacy and speed. However, the architecture also allows using remote models when needed: for instance, Anthropic Claude (e.g. *claude-3-opus*) or OpenAI’s GPT-4 via their cloud APIs. The orchestration logic should be agnostic to which backend is in use: the Context Manager will send prompts and receive completions from whichever model is configured (the user can select the model via config.json). The main difference is that with some advanced models (like GPT-4 with function calling, or future versions of Claude) formatting MCP requests might be simpler (since the model can output structured calls natively), whereas with open-source models we might need more prompt guidance to get them to produce proper tool-call outputs.

**MCP Servers (Tools & Data):** these are external services we will make available to the AI via MCP. In the diagram, four types are shown, corresponding to project requirements:

1. **File System Server:** allows browsing and reading (and potentially writing, if permitted) the project’s local file system. This is crucial to give the AI access to source code content. For example, the agent can use a read\_file tool exposed by this server to fetch the contents of FactionManager.cs on demand, rather than having to include it entirely in the prompt from the start. A reference filesystem MCP server is available open-source ([Example Servers - Model Context Protocol](https://modelcontextprotocol.io/examples#:~:text=Data%20and%20file%20systems)); it supports operations like listing directories, reading files with access controls, etc. (Cline itself integrates a Filesystem MCP server by default in some scenarios).
2. **Documentation Server:** provides access to external documentation (docs, guides, API references). Depending on the source, this could be implemented in different ways. If documentation is available online (e.g. Microsoft .NET docs or other library docs), we might use a *web search* or *fetch* MCP server. For instance, there’s a *Brave Search* MCP server for web search ([Example Servers - Model Context Protocol](https://modelcontextprotocol.io/examples#:~:text=%2A%20Brave%20Search%20,automation%20and%20web%20scraping%20capabilities)) and a *Fetch* server to retrieve web content in a cleaned format ([Example Servers - Model Context Protocol](https://modelcontextprotocol.io/examples#:~:text=%2A%20Brave%20Search%20,automation%20and%20web%20scraping%20capabilities)). Alternatively, if we have documentation locally (say a docs/ folder in the project, or manuals in PDF/Markdown), we could index that corpus and access it via a server (perhaps using a “Memory” server with embedded docs, see below, or creating a custom docs server for search).
3. **Memory Server:** a component for **extended conversational memory or persistent knowledge**. The panel already keeps a basic chat history, but with MCP we can have a dedicated *Memory Server* acting as a long-term knowledge base, possibly structured as a knowledge graph or vector store. The official repository includes a “Memory” server for knowledge graph-based persistent memory ([Example Servers - Model Context Protocol](https://modelcontextprotocol.io/examples#:~:text=%2A%20Google%20Maps%20,based%20persistent%20memory%20system)). This server would allow storing and querying contextual information (e.g. “what decision was made about the design of module X?” or “what is the system architecture?”) that persists across sessions. Agents (especially the Supervisor) could query this memory to maintain high-level coherence.
4. **Repository/Git Server:** to access external repositories or version control functions. If the project uses Git (locally or remote on GitHub/GitLab), a *Git MCP server* allows agents to perform operations like searching commits, examining diffs, or opening other files not currently in the editor ([Example Servers - Model Context Protocol](https://modelcontextprotocol.io/examples#:~:text=Development%20tools)). For example, if the bug in FactionManager.cs was introduced by a recent commit, the agent could call a git\_log or git\_diff tool to identify the problematic change. There are existing servers for Git, GitHub, and GitLab ([Example Servers - Model Context Protocol](https://modelcontextprotocol.io/examples#:~:text=Development%20tools)).

In addition, we might consider other useful MCP servers such as:

* **Test Runner:** a server to execute tests or code in a sandbox (for example, there’s E2B for running code in secure sandboxes ([Example Servers - Model Context Protocol](https://modelcontextprotocol.io/examples#:~:text=%2A%20Browserbase%20,the%20Qdrant%20vector%20search%20engine)), or a Docker server for container tasks ([Example Servers - Model Context Protocol](https://modelcontextprotocol.io/examples#:~:text=A%20growing%20ecosystem%20of%20community,servers%20extends%20MCP%E2%80%99s%20capabilities))). This would allow the Tester agent to actually run the project and capture output/errors.
* **Telegram Server (hypothetical):** there isn’t a public Telegram MCP server yet, but we could build one. A Telegram server could expose tools like send\_message or read\_updates to let agents send/read Telegram messages as part of their workflow. In its absence, we handle that via the Telegram integration module externally, but it’s worth noting we could encapsulate it in MCP for uniformity.

**Local vs Remote coordination:** A key principle is that MCP integration is handled primarily locally, within the VS Code extension. This means even the MCP servers ideally run locally (or on the LAN) for performance and privacy reasons. Fortunately, many MCP servers (filesystem, git, memory, etc.) are designed to run alongside the client locally. The panel could launch these servers as subprocesses on startup (or on demand) and communicate with them via **STDIO** – which is a standard transport for MCP ([GitHub - modelcontextprotocol/typescript-sdk: The official Typescript SDK for Model Context Protocol servers and clients](https://github.com/modelcontextprotocol/typescript-sdk#:~:text=The%20Model%20Context%20Protocol%20allows,specification%2C%20making%20it%20easy%20to)) ([GitHub - modelcontextprotocol/typescript-sdk: The official Typescript SDK for Model Context Protocol servers and clients](https://github.com/modelcontextprotocol/typescript-sdk#:~:text=import%20,zod)). For example, the official *TypeScript SDK* supports a StdioServerTransport to easily connect a server over stdin/stdout ([GitHub - modelcontextprotocol/typescript-sdk: The official Typescript SDK for Model Context Protocol servers and clients](https://github.com/modelcontextprotocol/typescript-sdk#:~:text=import%20,zod)) ([GitHub - modelcontextprotocol/typescript-sdk: The official Typescript SDK for Model Context Protocol servers and clients](https://github.com/modelcontextprotocol/typescript-sdk#:~:text=%2F%2F%20Add%20an%20addition%20tool,)). This fits the VS Code extension context well: we can include or install the MCP servers as NPM packages or binaries, then have the MCP client (in the panel) spawn them and talk to them without network overhead.

At the same time, we ensure **remote support** when needed: if a particular capability inherently requires an external service (e.g. querying an online API, performing a web search, or using a remote AI model), MCP accommodates that. We can have an MCP server that acts as a *proxy* to a remote service. For instance, the Brave Search server needs Brave’s API and thus internet access; we could still run it locally but it will make external calls. The architecture anticipates that in such cases, sensitive data (API keys) remain confined to the MCP server itself, which as noted is isolated and can be individually authorized. In summary, our logic and coordination remain local, but when **MCP “requires it”**, we can involve cloud services. This meets the goal of prioritizing local integration while retaining the option to tap into cloud resources if absolutely necessary (e.g. using Claude via API if the local model isn’t sufficient for a task, or fetching documentation from a website). The modular design of MCP servers makes this straightforward: we decide which servers to activate depending on the mode (offline/pure-local vs online-enabled).

### **Workflow with MCP Integrated**

With the above architecture, here’s how a user workflow would play out when using the AI panel for a task (like *“fix a bug in FactionManager.cs”*):

1. **User Input:** The user provides a request, e.g. by typing in VS Code or sending a Telegram message (“Find and fix the bug in *FactionManager.cs* causing a NullReference error”). The message reaches the AI agents.
2. **Initial Context Gathering:** The *Context Manager* examines the request and determines what contextual sources to activate. Based on the context\_sources configuration (see JSON structure later), it might see that the open file FactionManager.cs should be included (*source:* “files”), conversation history should be included (*source:* “memory”), and any documentation notes about FactionManager should be fetched (*source:* “docs”).  
   * Without MCP, typically we’d read the file from disk and insert its entire content into the prompt. With MCP, instead of injecting the whole file, we can insert a reference or summary and let the model call for details via tools. For example, we might include in the initial prompt a note like: *“The file FactionManager.cs is available as resource file://project/FactionManager.cs (200 lines).”* Additionally, if the bug produces a runtime error, the tester agent might have a log or stack trace available (perhaps loaded in the memory server).
   * So the initial context provided can be lighter, containing just what we deem immediately necessary (e.g. relevant class/method signatures, error messages), knowing that thanks to MCP the model can drill down by calling tools when needed.
3. **Multi-Agent Coordination:** The *Developer* agent takes the lead given the goal. It might analyze the bug description and decide to read the FactionManager.cs code to find the cause. Instead of needing that code already in the prompt, the agent (via the model) can internally formulate an MCP request – for example, the model might produce a structured output or trigger phrase like: *“[Call: read\_file, Target: FactionManager.cs]”*. The exact syntax depends on how we prompt/format it (it could be a JSON block or a simple special command in the text, depending on conventions), but in any case the *Context Manager* will recognize the model’s intent to invoke a tool.  
   * The manager then makes the real call to the File System MCP server: a tools/call with name read\_file and parameter path = FactionManager.cs ([Function Calling vs. Model Context Protocol (MCP): What You Need to Know - DEV Community](https://dev.to/fotiecodes/function-calling-vs-model-context-protocol-mcp-what-you-need-to-know-4nbo#:~:text=,%7D)). The server receives that, reads the file from disk (assuming it’s in the VS Code workspace), and returns the content (maybe chunked if needed). Note: ideally, the server should support requesting file segments (e.g. specific line ranges) for efficiency, especially if files are large.
   * Once the content is obtained, the manager provides it back to the model. Often, this is done by inserting the content as the assistant’s message or part of the next prompt turn.
   * The Developer agent’s model now has the code and can locate the bug line. Suppose it finds an uninitialized variable causing the null reference. At this point, it can directly propose a fix.
   * The Tester agent, in parallel, once the code is available might mentally (or via a tool) execute the problematic method. If we integrated a test runner, it could do [Call: run\_tests, Target: FactionManagerTests]. If not, it can at least simulate or do static checks.
   * The Supervisor agent monitors these actions. It may intervene if, say, the Developer agent’s fix seems incorrect. The Supervisor might consult the **Memory** to retrieve if there were past decisions about how factions should be handled. To do this, it could use a memory\_search tool provided by the Memory server, with a query like “Faction rules”. The response might remind that “factions should not be created if null,” etc., information which the Supervisor then communicates to the Developer agent to guide the fix.
4. **Solution Proposal and Validation:** The Developer agent, armed with all information, proposes a code fix. This could materialize in the panel as a diff or direct edit to the file (Cline typically can apply patches to the code). Before automatically applying, the *Supervisor* or a human user might review it. The Tester agent might run tests on the fixed code (if automated), or at least reason about the outcome after the patch.  
   * If integrated with Git, we could even automatically create a commit for the fix or at least show the diff and ask for confirmation.
   * The user finally sees the suggested fix. If satisfied, they apply it (or instruct the agent to apply it).
5. **Iteration via Telegram (optional):** Suppose the user launched the request then stepped away from the IDE. Thanks to the Telegram connection, the agent could send a message upon completion: “I found the bug and proposed a fix, awaiting your confirmation.” The user on Telegram could reply “Go ahead and apply the fix and re-run the build,” which enters the loop and so on.

In this integrated scenario, note how MCP enabled the agents to:

* **Access needed source code on-demand** without flooding every prompt with the entire repository. This keeps prompts concise and within the model’s token limit, delegating details to MCP tools when needed.
* **Consult external sources** (project memory, online documentation) interactively. If the model is unsure about an API or method, it can call a search\_docs tool that searches the docs and returns relevant snippets which the model then incorporates.
* **Automate repetitive actions**: such as searching the repo for all references to a function, or running tests, all via standardized commands.

To implement all this, from a developer’s perspective, we need both **software components** (the MCP client in the extension, the configured MCP servers) and **appropriate configuration settings**. We will detail these aspects in the following subsections.\*

## **Dettagli di Implementazione e Linee Guida / *Implementation Details and Guidelines***

**1. Utilizzo dell’SDK MCP e integrazione nel pannello:** Per implementare il client MCP nell’estensione VS Code, possiamo sfruttare l’SDK ufficiale Model Context Protocol disponibile per TypeScript ([GitHub - modelcontextprotocol/typescript-sdk: The official Typescript SDK for Model Context Protocol servers and clients](https://github.com/modelcontextprotocol/typescript-sdk#:~:text=The%20Model%20Context%20Protocol%20allows,specification%2C%20making%20it%20easy%20to)). Dato che l’estensione Cline è basata su Node.js/TypeScript, questo SDK è ideale. Esso fornisce classi e metodi per gestire la connessione ai server, l’invio di richieste JSON-RPC e la ricezione delle risposte, occupandosi anche di dettagli come il framing dei messaggi e la concurrency. Ad esempio, l’SDK espone la classe McpClient (o metodi simili) che possiamo inizializzare indicando il transport (STDIO, HTTP SSE, ecc.) e poi usare per invocare tool o leggere risorse. Dal README del SDK TS: *“Questo SDK implementa l’intera specifica MCP, rendendo facile costruire client MCP che si connettano a qualunque server MCP, nonché creare server MCP che espongano risorse, prompt e tool”* ([GitHub - modelcontextprotocol/typescript-sdk: The official Typescript SDK for Model Context Protocol servers and clients](https://github.com/modelcontextprotocol/typescript-sdk#:~:text=The%20Model%20Context%20Protocol%20allows,specification%2C%20making%20it%20easy%20to)).

* **Avvio dei server MCP:** Per ogni fonte di contesto prevista, decidiamo quale server MCP impiegare. Molti server community sono open-source su GitHub (vedi l’elenco ufficiale su modelcontextprotocol.io o repositori “awesome-mcp” ([Adding MCP Servers from GitHub | Cline](https://docs.cline.bot/mcp-server-from-github#:~:text=,MCP%20servers%20on%20GitHub%20include))). Cline offre un meccanismo integrato per *clonare e buildare* server MCP direttamente da un repository GitHub ([Adding MCP Servers from GitHub | Cline](https://docs.cline.bot/mcp-server-from-github#:~:text=There%20are%20multiple%20places%20online,to%20find%20MCP%20servers)). Possiamo sfruttare questa capacità: ad esempio, aggiungere nel file di config un riferimento al repository del File System server e del Git server. Cline li clonerà (in una cartella temporanea) e compilerà (sono spesso Node or Python projects) e li renderà disponibili. In alternativa, se preferiamo maggiore controllo, possiamo aggiungerli come dipendenze NPM del nostro pannello (se esistono pacchetti pubblicati) o includere binari precompilati.
* Una volta che i server sono disponibili, l’estensione li lancia. Possiamo farlo all’attivazione dell’estensione, oppure in modo pigro la prima volta che un agente ne richiede l’uso. Avviare un server comporta tipicamente eseguire un comando (ad es. node filesystem-server.js). Per convogliare la comunicazione, passiamo i pipe STDIN/STDOUT al processo figlio così che l’SDK MCP possa dialogare.
* Configurare il **transport**: con STDIO, l’SDK TS consente di creare un StdioServerTransport e connetterlo al processo figlio in pochi passi ([GitHub - modelcontextprotocol/typescript-sdk: The official Typescript SDK for Model Context Protocol servers and clients](https://github.com/modelcontextprotocol/typescript-sdk#:~:text=async%20%28uri%2C%20,name%7D%21%60%20%7D%5D%20%7D%29)) ([GitHub - modelcontextprotocol/typescript-sdk: The official Typescript SDK for Model Context Protocol servers and clients](https://github.com/modelcontextprotocol/typescript-sdk#:~:text=%2F%2F%20Add%20an%20addition%20tool,)). In alternativa, alcuni server supportano --port per avviarsi come server HTTP SSE; in tal caso useremmo un HttpClientTransport indicando l’endpoint. STDIO è comunque preferibile per la semplicità all’interno di VS Code.
* **Registrazione dei tool e risorse:** Una volta connesso il client a un server, è opportuno chiamare l’endpoint di listing (tools/list, resources/list) per sapere quali nomi di tool e risorse sono disponibili. L’SDK probabilmente lo fa automaticamente o fornisce un metodo. Il pannello potrebbe poi usare queste info per arricchire le capacità dell’agente. Cline ad esempio integra i nomi dei tool disponibili nel prompt invisibile per il modello, in modo che sappia di poterli usare. Potremmo replicare questo: es. nell’istruzione di sistema al modello includere una lista tipo: *“Strumenti disponibili: read\_file(path), search\_code(query), memory\_search(query), send\_telegram(message)…”*. Questo aiuta soprattutto modelli non esplicitamente addestrati su MCP a capire cosa può fare.

Vale la pena menzionare che Anthropic Claude potrebbe già essere addestrato a usare MCP (essendo creato dalla stessa azienda): in Claude Desktop, ad esempio, l’uso di MCP è nativo. Per i nostri scopi, comunque, formuleremo i prompt in modo consistente per tutti i modelli.

**2. Prompt engineering per tool use:** Per modelli OpenAI GPT-4/GPT-3.5, potremmo usare il meccanismo *function calling* interno: definire ciascun tool MCP come una “function” nell’API OpenAI. Però, tenendo conto che vogliamo indipendenza dal vendor e uniformità via MCP, è meglio evitare di mischiare i due livelli. Invece, usiamo un semplice protocollo testuale. Ad esempio, possiamo adottare la convenzione che se il modello vuole chiamare un tool, produrrà un JSON del tipo:

{"tool": "<nome\_tool>", "arguments": { <parametri> }}

Oppure un markdown speciale, tipo: Tool[<nome\_tool>]: <args…>. Scegliamo un formato e istruiamo il modello di conseguenza nel prompt di sistema (e.g. *“Quando hai bisogno di usare uno strumento, rispondi esclusivamente con un JSON contenente 'tool' e 'arguments'.”*). Il Gestore Contesto intercetterà output che matchano questa struttura prima di inoltrare la risposta all’utente, e li interpreterà come chiamate MCP. Questo approccio è simile a quello di LangChain Agents dove il modello produce un “action” e l’agente esecutore lo realizza.

* In ambienti Cline, esiste già una gestione simile: Cline in modalità “Act” fa sì che il modello emetta comandi (per terminale, per editor, ecc.). Possiamo riutilizzare quell’idea per i tool MCP. Se l’estensione è un fork di Cline, integrarsi nel suo ciclo *Plan/Act* può essere opportuno: i MCP tool diventano possibili “azioni” nell’**Act Phase** del piano dell’agente.
* Per modelli come Claude, che in contesti come Claude Desktop già riconoscono i tool MCP, il compito è semplificato. Per modelli Llama2-based, potremmo dover iterare un po’ per ottenere la formattazione giusta (ad es. incluse parole chiave come “MCP” nel prompt).
* **Approccio con approvazione utente:** Il design dovrebbe includere un passo di conferma quando un tool potenzialmente pericoloso sta per essere usato. Ad esempio, se il modello tenta di chiamare delete\_file o di invocare un’API sensibile, potremmo mettere in pausa e mostrare un pop-up in VS Code: “L’agente vuole eseguire X, consentire?”. Questo è in linea con la filosofia MCP che prevede “human-in-the-loop” per azioni critiche ([MCP Overview | Cline](https://docs.cline.bot/mcp-servers/mcp#:~:text=database%20queries)). Possiamo implementarlo intercettando quelle chiamate specifiche prima di eseguirle.

**3. Struttura di Configurazione (config/config.json):** Per rendere l’integrazione flessibile, estendiamo il file di config con parametri relativi a MCP. Ad esempio, potremmo aggiungere una sezione mcp con la lista dei server attivi e come avviarli, oppure più semplicemente indicare le *fonti di contesto* e lasciare che il codice decida i server:

{

// ... altre configurazioni ...

"context\_sources": ["docs", "files", "memory", "telegram"],

"context\_protocol": "MCP",

"model": "claude-3-opus",

"mcp\_servers": {

"files": "github:anthropics/cli-mcp-filesystem",

"docs": "github:myorg/mcp-docs-search",

"memory": "github:anthropics/cli-mcp-memory",

"repo": "github:anthropics/cli-mcp-git"

}

}

Nel sopra esempio, context\_sources definisce le tipologie attive, e mcp\_servers mappa ciascuna fonte a un implementazione (ipotetica via URL GitHub). Il client al lancio itererebbe su questa mappa per avviare i server e connettersi. La chiave context\_protocol: MCP può servire per abilitare/disabilitare in blocco l’uso di MCP – se settata a “none” o diverso, il pannello potrebbe ricadere sul vecchio comportamento (iniettare contesto statico). Questo fallback è utile per compatibilità o debugging.

Nel JSON fornito nell’obiettivo:

{

"context\_sources": ["docs", "files", "memory", "telegram"],

"user\_goal": "correggi bug in FactionManager.cs",

"model": "claude-3-opus",

"context\_protocol": "MCP"

}

vediamo un esempio di input strutturato. In realtà, user\_goal sarebbe la richiesta utente corrente (non un’impostazione persistente, ma piuttosto una rappresentazione del prompt). Un design elegante potrebbe essere che il *Telegram bot* o un’interfaccia UI invii al pannello richieste già formattate in questo JSON, cosicché l’orchestratore sappia immediatamente cosa fare: abilitare i context sources indicati, usare quell’obiettivo e modello. È un’idea interessante per uniformare le chiamate tra agenti: ad esempio il Supervisor potrebbe generare un nuovo sub-task per il Developer formattato come JSON di questo tipo. In generale, per la configurazione possiamo attenersi a specificare le fonti, e il resto (user\_goal, model) saranno parametri variabili per ogni esecuzione.

**4. Esempi di codice e risorse utili:** Per guidare l’implementazione, è consigliabile studiare gli esempi esistenti:

* *Esempio di server+client TS:* il repository *MCP Server-Client Example* su GitHub dimostra come implementare un server MCP e un client TS che si parlano via stdio ([GitHub - JoeBuildsStuff/mcp-server-client](https://github.com/joeBlockchain/mcp-server-client#:~:text=MCP%20Server)) ([GitHub - JoeBuildsStuff/mcp-server-client](https://github.com/joeBlockchain/mcp-server-client#:~:text=npm%20run%20start%3Aserver)). Si tratta di un progetto minimale che espone un file di esempio come risorsa e mostra il ciclo di richiesta dal client. Questo può servire come base per capire l’integrazione.
* *Cline documentation:* la documentazione ufficiale di Cline fornisce linee guida per estendere l’estensione con MCP. In particolare, la sezione *Building MCP Servers with Cline* descrive come si può istruire Cline (anche in linguaggio naturale) per configurare un server ([MCP Overview | Cline](https://docs.cline.bot/mcp-servers/mcp#:~:text=Building%20MCP%20Servers)) e come poi eseguirlo. Anche se nel nostro caso faremo a mano, è utile leggere come Cline automatizza il processo, per replicarne i passi.
* *SDK docs:* leggere la *specifica MCP* e la sezione *Writing MCP Clients* dell’SDK TypeScript. Qui troveremo l’uso di classi come McpClient o Protocol<ClientRequest,…> e come registrare gli handler ([Core architecture - Model Context Protocol](https://modelcontextprotocol.io/docs/concepts/architecture#:~:text=MCP%20follows%20a%20client,where)) ([Core architecture - Model Context Protocol](https://modelcontextprotocol.io/docs/concepts/architecture#:~:text=,tools%2C%20and%20prompts%20to%20clients)). Dato che implementiamo all’interno di VS Code extension, è importante gestire bene i cicli asincroni e assicurare che i processi server vengano terminati in chiusura dell’IDE.

**5. Testing e debug:** Una volta implementato, testare a fondo:

* Provare l’avvio dei server e verificare che compaiano come processi figli dell’estensione, e che rispondano a ping (magari c’è un health tool in alcuni).
* Simulare prompt dove il modello deve chiamare i tool: ad esempio chiedere “Apri il file X e riassumine il contenuto”. Vedere se l’agente produce la chiamata e se la risposta viene correttamente incorporata. Utilizzare l’**Inspector** MCP (strumento di debug menzionato da Anthropic ([Introduction - Model Context Protocol](https://modelcontextprotocol.io/introduction#:~:text=,Inspector))) se disponibile, per monitorare i messaggi scambiati.
* Valutare le prestazioni: l’uso di MCP aggiunge overhead di latenza (una chiamata extra round-trip). Per mitigare, assicurarsi che quando possibile si utilizzino *risorse* statiche invece di chiamate frammentate: ad esempio, se sappiamo che un file piccolo è sempre necessario, possiamo ancora mandarlo in un colpo solo. MCP è più utile per dati grandi o condizionali.
* Controllare che la memoria conversazionale rimanga coerente: quando il modello riceve un output da un tool, questo dovrebbe essere anche incluso nel log della conversazione se vogliamo che i turni successivi lo ricordino. In pratica, ogni tool call+risposta può essere trattata come un turno di dialogo aggiuntivo.

Seguendo questi passi, l’AI Developer Panel avrà un’integrazione MCP robusta, capace di interfacciarsi con repository e documentazione in modo standard. Gli agenti multi-ruolo potranno collaborare ancora più efficacemente, ciascuno sfruttando i tool di cui ha bisogno (ad es. il Tester chiamerà i suoi tool, il Developer i suoi, etc.) ma condividendo la stessa infrastruttura.

* **Riepilogo Implementazione:**
  + Integrare l’SDK MCP (TypeScript) nell’estensione; configurare MCP client nel Context Manager.
  + Procurare i server MCP necessari (file, docs, memory, git, etc.) – via clone da GitHub o pacchetti – e farli partire (preferibilmente con STDIO).
  + Al lancio, connettere il client a ciascun server, eventualmente aggiornare la lista strumenti disponibili.
  + Adattare il prompt di sistema/inizio per informare il modello sugli strumenti (o addestrare gli agenti a scoprirli via listing).
  + Implementare il loop di parsing dell’output del modello: se rilevata richiesta tool, bloccare risposta utente, eseguire MCP call, dare output al modello, e solo quando il modello fornisce una risposta finale (non-tool) allora inoltrarla all’utente.
  + Aggiungere controlli di sicurezza sulle tool call (conferme, limiti di tempo, ecc.).
  + Testare con esempi pratici e aggiustare le istruzioni fornite al modello finché l’utilizzo dei tool diventa affidabile.
* **(English Translation of key implementation steps)**:  
  + **Use the MCP SDK**: Integrate the official TypeScript MCP SDK in the extension ([GitHub - modelcontextprotocol/typescript-sdk: The official Typescript SDK for Model Context Protocol servers and clients](https://github.com/modelcontextprotocol/typescript-sdk#:~:text=The%20Model%20Context%20Protocol%20allows,specification%2C%20making%20it%20easy%20to)), creating an MCP client to manage JSON-RPC calls and connections.
  + **Launch MCP servers** for each context source (file, docs, memory, repo). Possibly leverage Cline’s ability to auto-clone and build servers from GitHub ([Adding MCP Servers from GitHub | Cline](https://docs.cline.bot/mcp-server-from-github#:~:text=There%20are%20multiple%20places%20online,to%20find%20MCP%20servers)), or include them as dependencies. Use STDIO transport for local servers (via StdioServerTransport in the SDK) ([GitHub - modelcontextprotocol/typescript-sdk: The official Typescript SDK for Model Context Protocol servers and clients](https://github.com/modelcontextprotocol/typescript-sdk#:~:text=async%20%28uri%2C%20,name%7D%21%60%20%7D%5D%20%7D%29)). Ensure to gather the available tools/resources from each server after connecting.
  + **Prompt the model about tools**: Include in the system or context prompt a description of available MCP tools so the model knows what it can do (especially needed for models that don’t have innate MCP knowledge).
  + **Parse model outputs for tool calls**: Implement a loop where if the model’s output matches the format of a tool request (e.g. a JSON with "tool": ...), intercept it, call the corresponding MCP server via tools/call, get the result, and feed it back into the model. Only when the model produces a final answer (normal text) should it be shown to the user. This mediator loop is analogous to agent frameworks in LangChain, but here using MCP as execution layer.
  + **User approval and safety**: Insert user approval steps for sensitive actions. The extension can prompt the user if a tool call is deemed risky (like file delete or external API with costs).
  + **Extend config**: use config.json to specify which context sources/MCP servers are enabled. The example given can be adopted, where context\_sources list drives which servers to load and context\_protocol: "MCP" toggles the feature. The JSON structure example from the user is more for messaging, but we incorporate that format concept in orchestrating tasks.
  + **Testing**: Try out scenarios where agents use tools, ensure the interactions are correctly handled and debug with available tools (Anthropic’s MCP Inspector, etc.). Optimize as needed, and maintain a fallback to non-MCP mode if the user chooses (for example, if context\_protocol is set to none or context\_sources is empty, the panel could revert to embedding context).

By following these guidelines, the AI Developer Panel will gain a robust MCP integration, enabling standardized interfacing with code repositories and documentation. The multi-role agents will collaborate more effectively, each leveraging the tools they need (e.g. Tester calls testing tools, Developer calls code tools, etc.) while sharing the same infrastructure.\*

## **Confronto con Altre Soluzioni (OpenAI Tools, LangChain/LangGraph, AutoGen, CrewAI) / *Comparison with Alternative Approaches***

**OpenAI Tool Use (Function Calling e Plugins):** Prima dell’avvento di MCP, uno dei modi principali per far interagire i modelli con strumenti era tramite le *function calls* di OpenAI o il sistema di plugin di ChatGPT. Con la function calling, ad esempio, GPT-4 può essere istruito con alcune funzioni predefinite (in JSON schema) e il modello restituirà un oggetto strutturato quando ritiene di dover invocare quella funzione. Questa tecnologia ha concetti simili a MCP – infatti, la combinazione function calling + esecuzione lato applicazione ricorda molto la coppia “LLM genera richiesta + MCP la esegue”. La differenza è che OpenAI’s function calling è **specifica del vendor** e limitata ai modelli OpenAI (GPT-4, GPT-3.5 con supporto funzioni) e a quelli che implementano schema simili. Inoltre, non definisce uno standard per come eseguire le funzioni: spetta allo sviluppatore implementare l’esecuzione di ciascuna funzione nell’app. MCP invece standardizza anche la fase di esecuzione tramite un protocollo comune (JSON-RPC su vari trasporti) ([Function Calling vs. Model Context Protocol (MCP): What You Need to Know - DEV Community](https://dev.to/fotiecodes/function-calling-vs-model-context-protocol-mcp-what-you-need-to-know-4nbo#:~:text=Feature%20Function%20Calling%20MCP%20,Ensures)). Un altro limite dell’approccio OpenAI puro è che ogni volta bisogna ri-definire tutte le funzioni al modello, e formati diversi per modelli diversi (ad es. la sintassi di function call in GPT differisce da quella di, diciamo, PaLM di Google) ([Function Calling vs. Model Context Protocol (MCP): What You Need to Know - DEV Community](https://dev.to/fotiecodes/function-calling-vs-model-context-protocol-mcp-what-you-need-to-know-4nbo#:~:text=Each%20model%20formats%20function%20calls,LLMs%20by%20handling%20these%20variations)). MCP elimina questa frammentazione imponendo un formato unico per le richieste (simile per tutti i modelli, lato client) e fungendo da ponte tra modelli e strumenti.

Vale la pena sottolineare che MCP ed OpenAI function calling non si escludono a vicenda, anzi possono lavorare insieme: un possibile scenario è usare function calling per “catturare” l’intento del modello (ad es. definire una singola funzione call\_tool(tool\_name, args)), e poi all’interno di quella implementazione usare MCP per inoltrare al tool vero. In letteratura, è stato notato che *“Function Calling converte i prompt in chiamate strutturate, mentre MCP assicura che quelle chiamate vengano eseguite in modo consistente e scalabile”* ([Function Calling vs. Model Context Protocol (MCP): What You Need to Know - DEV Community](https://dev.to/fotiecodes/function-calling-vs-model-context-protocol-mcp-what-you-need-to-know-4nbo#:~:text=Feature%20Function%20Calling%20MCP%20,Ensures)). Quindi, la nostra scelta di implementare MCP direttamente ci dà massima flessibilità (slegandoci dal dover usare per forza GPT-4), a costo di dover gestire noi il ciclo di chiamata.

Rispetto ai **Plugins di OpenAI/ChatGPT**, MCP è più generale. I plugin richiedono di implementare un web server con specifiche OpenAI e registrarlo presso ChatGPT: ciò li rende utili solo in quell’ecosistema e per quel modello. MCP invece è concepito per essere usato con qualsiasi modello compatibile, e all’interno di applicazioni controllate dall’utente (come la nostra estensione VS Code) senza necessità di autorizzare esternamente un plugin. In sintesi, MCP è **open-standard e self-hosted** dove gli OpenAI Tools sono **proprietari e cloud-hosted**. Per un progetto come il nostro, che enfatizza l’integrazione locale e la privacy, MCP offre un controllo maggiore: i dati del repository e i documenti non devono lasciare la macchina locale (possono essere serviti via MCP in locale), cosa che invece con plugin cloud sarebbe rischioso.

**LangChain Agents / LangGraph:** LangChain è una popolare libreria Python per costruire catene di LLM e agenti che usano tool. Offre diversi “agent frameworks” in cui il modello può decidere un’azione (da un insieme di tool Python definiti) e la libreria esegue quell’azione, rientrando il risultato al modello, in un loop fino ad una soluzione. Il nostro approccio con MCP è concettualmente simile, ma differisce nell’implementazione e nella portata:

* LangChain è pensata per Python, e tipicamente viene usata in applicazioni Python (script, backend server, Jupyter). Nel contesto VS Code extension (Node.js), usarla direttamente sarebbe scomodo (richiederebbe avviare un processo Python o un server). Invece, MCP con l’SDK TS si integra nativamente nel nostro stack.
* LangChain define i tool come semplici funzioni Python che l’agente chiama. Ciò va bene per prototipi, ma su larga scala ci si scontra di nuovo col problema di dover reimplementare tool per ogni ambiente e modello. MCP fornisce un ecosistema già fatto di server riutilizzabili. Ad esempio, invece di scrivere da zero una funzione Python per leggere file e magari dover replicare logica di sicurezza, possiamo usare il server FileSystem robusto e testato ([Example Servers - Model Context Protocol](https://modelcontextprotocol.io/examples#:~:text=Data%20and%20file%20systems)).
* **LangGraph** in particolare è un’estensione di LangChain orientata a flussi a grafo (DAG) e multi-actor ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=LangGraph%20is%20an%20open,by%20the%20long%20history%20of)). Può orchestrare step complessi con nodi che rappresentano compiti specifici, integrando memoria avanzata e strumenti. In altri termini, LangGraph fornisce un controllo di flusso più dichiarativo. Nel nostro caso, potremmo non averne bisogno esplicito: la natura stessa del nostro sistema è multi-agente e i ruoli definiscono in parte un flusso (dev -> test -> review). Implementando MCP, manteniamo la logica di orchestrazione per lo più custom nell’estensione. Se avessimo voluto delegare il coordinamento a un framework esterno, LangGraph sarebbe un candidato ma al costo di aumentare la complessità (introducendo la necessità di modellare tutto il processo come un grafo e integrare quell’esecuzione col nostro UI).
* LangChain/LangGraph offrono **integrazione con molti tool** (web search, DB, etc.), ma in pratica la maggior parte di questi tool sono implementati come wrapper attorno a API specifiche (ad esempio, una classe Tool che chiama serpapi per la ricerca web). Con MCP possiamo raggiungere le stesse API ma tramite un canale unificato. Va detto però che LangChain e MCP non sono incompatibili: potremmo creare un *MCP Tool for LangChain* per usare server MCP dentro LangChain. Viceversa, potremmo ispirarci a come LangChain formatta i prompt per spingere il modello a scegliere un tool (il cosiddetto prompt “ReAct”). Dato che stiamo integrando in un prodotto (VS Code extension), probabilmente preferiamo controllare noi il prompt per i tool usage.

In definitiva, **LangChain/LangGraph** sono ottimi per costruire agenti in ambienti centralizzati (scripting, notebooks). **MCP** è migliore per applicazioni distribuite/modulari e per standardizzare l’interfaccia con i tool. La nostra scelta di MCP si allinea con l’idea di mantenere l’integrazione *all’interno dell’IDE e della sua tecnologia*, e di non dipendere da un ecosistema Python esterno. Inoltre, frameworks come LangGraph potrebbero essere overkill per un progetto singolo: preferiamo implementare la logica multi-agente su misura, mentre LangGraph brilla se devi creare molti flussi complessi in generale.

Una nota: LangGraph e CrewAI integrano anch’essi tool multipli e memoria estesa ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=LangGraph%3A%20LangGraph%20integrates%20seamlessly%20with,the%20functionality%20of%20their%20agents)) ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=LangGraph%3A%20LangGraph%20provides%20built,revisit%20and%20analyze%20previous%20states)), il che significa che anche loro hanno dovuto risolvere problemi simili. Ma dal confronto pubblicato, **“LangGraph e CrewAI hanno un vantaggio grazie alla perfetta integrazione con LangChain, che offre una vasta gamma di tool, e un sistema di memoria completo”** ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=match%20at%20L224%20,the%20addition%20of%20custom%20tools)) ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=Crew%20AI%3A%20Crew%20AI%20offers,recall%20important%20information%20across%20multiple)). Con MCP, stiamo fornendo al nostro sistema un’analoga gamma di tool (attraverso i server) e anche opzioni di memoria (via Memory server). Quindi stiamo ottenendo benefici simili, ma in un modo più personalizzato e integrato nell’IDE.

**AutoGen di Microsoft:** AutoGen è un framework open-source di Microsoft per creare sistemi multi-agente conversazionali e collaborativi. L’idea di base di AutoGen è modellare il workflow come una conversazione tra agenti specialisti ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=Autogen)) ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=match%20at%20L338%20Autogen%3A%20Autogen,including%20sequential%20and%20nested%20chats)). Ad esempio, c’è un “Assistant” e un “Developer” che parlano tra loro per risolvere un problema. AutoGen supporta *various tools (code executors, function callers, ecc.)* e mette un’enfasi sulla semplicità di configurazione – è modulare e relativamente facile da estendere ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=Autogen%20supports%20various%20tools%2C%20including,modular%20and%20easy%20to%20maintain)) ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=match%20at%20L192%20Autogen%3A%20Autogen,It%20abstracts%20much%20of%20the)). In un certo senso, il nostro AI Developer Panel è un’implementazione concreta di concetti di AutoGen: abbiamo diversi agenti (sviluppatore, tester, supervisore) che comunicano. La differenza è che noi li stiamo integrando in un contesto IDE con controlli specifici. AutoGen, essendo pensato come libreria, richiederebbe di scrivere codice per definire questi agenti e le regole di interazione. Inoltre, AutoGen è anch’esso Python-based. Se avessimo voluto costruire il nostro pannello da zero senza Cline, magari avremmo potuto usare AutoGen come base per la logica multi-agente, ma data l’infrastruttura esistente, è più efficiente estendere quella con MCP che non rimpiazzarla.

In termini di **tool use**, AutoGen consente agli agenti di usare strumenti, ma normalmente appoggiandosi a function calling o integrazioni custom. Non offre uno standard come MCP; dovremmo comunque pluggare l’uso di file/doc come abbiamo fatto, solo che lo faremmo nel contesto AutoGen. D’altro canto, un vantaggio di AutoGen è la gestione di conversazioni nested e sub-tasks in modo abbastanza pulito, e una documentazione ricca di esempi ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=match%20at%20L301%20Autogen%3A%20Autogen,of%20key%20concepts%20and%20features)). Il nostro progetto però è focalizzato su sviluppo codice, quindi la struttura dei ruoli è già chiara e fissa, non creiamo/distruggiamo agenti arbitrariamente a runtime (che è una cosa che AutoGen supporta, ad esempio potresti spin-off un agente “Debugger” temporaneo).

**CrewAI:** CrewAI è un altro framework multi-agente orientato alla collaborazione in team di AI. Si distingue perché enfatizza la definizione di *ruoli* per ciascun agente e la possibilità di lavorare come team con obiettivi comuni ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=Crew%20AI)). In effetti il nostro scenario è proprio un team di agenti con ruoli (dev, tester, supervisor). CrewAI fornisce strutture per delega di compiti tra agenti, gestione flessibile dei task e integra strettamente **LangChain** per i tool e la memoria ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=match%20at%20L221%20Crew%20AI%3A,tailored%20to%20their%20specific%20needs)). Dal confronto emerso: CrewAI ha ottime capacità di memoria (breve, lungo termine, entità) e supporto per output strutturato (es. può far sì che gli agenti condividano informazioni in JSON con Pydantic) ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=Crew%20AI%3A%20Crew%20AI%20offers,recall%20important%20information%20across%20multiple)) ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=Crew%20AI%3A%20Crew%20AI%20supports,to%20meet%20their%20specific%20requirements)). Questo riduce miscommunication tra agenti.

Il nostro pannello, arricchito con MCP, raggiunge obiettivi simili:

* Ruoli ben definiti (hard-coded in a sense) come CrewAI;
* Memoria condivisa se usiamo il Memory server, coprendo contesto breve (chat) e lungo (knowledge base), equiparabile alla “comprehensive memory system” di CrewAI ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=Crew%20AI%3A%20Crew%20AI%20offers,recall%20important%20information%20across%20multiple));
* Possibilità di strutturare le comunicazioni: volendo, gli agenti nostri potrebbero scambiarsi JSON (magari usando il Memory server come bacheca, o tramite la stessa orchestrazione in codice). CrewAI lo formalizza di più, ma nulla ci vieta di implementare regole: es. potremmo decidere che il Tester riporta bug come un oggetto strutturato che il Developer agent interpreta.

Un punto di forza di CrewAI è la semplicità di onboarding: “straightforward to get started” con esempi e documentazione chiara ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=match%20at%20L198%20Crew%20AI%3A,approach%20to%20defining%20and%20managing)) ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=Documentation)). Nel nostro caso, però, abbiamo esigenze su misura (integrazione IDE, Telegram, etc.), quindi un framework generico potrebbe non coprire tutto senza pesanti personalizzazioni.

**Sintesi del confronto:** L’integrazione MCP che stiamo adottando posiziona il progetto in modo flessibile tra queste soluzioni. MCP fornisce uno **standard aperto** e implementazioni pronte per strumenti comuni (vs. dover scrivere plugin specifici come con OpenAI). A differenza di LangChain/LangGraph, la nostra soluzione non richiede dipendenze esterne pesanti e rimane interamente nel dominio JavaScript/Node dell’estensione, pur beneficiando di concetti analoghi (loop percepisci-azione). Rispetto a frameworks multi-agente come AutoGen e CrewAI, la nostra implementazione è più focalizzata: quei framework sono generici e orientati a supportare qualunque configurazione di agenti, mentre noi sappiamo esattamente i ruoli e le interazioni che vogliamo, e possiamo codificarle in modo deterministico usando MCP come mezzo per le azioni. In cambio, otteniamo un sistema più controllabile e integrato con l’IDE.

Va notato che essendo MCP relativamente nuovo, la community sta ancora costruendo best practice; ma le prospettive sono promettenti dato che molti vendor e open-source adopters stanno contribuendo (c’è un repository “official servers” e uno “community/awesome” in crescita ([Adding MCP Servers from GitHub | Cline](https://docs.cline.bot/mcp-server-from-github#:~:text=,MCP%20servers%20on%20GitHub%20include))). Adottare MCP ora dà al nostro progetto un’impostazione futura-proof: se un domani OpenAI o altri supportassero nativamente MCP, il nostro pannello ne beneficerebbe immediatamente. Invece legarsi a soluzioni proprietarie rischia di dover essere rivisto se cambiano API o policy.

In conclusione, **MCP** si distingue come la scelta giusta per un tool di sviluppo come il nostro, dove la **modularità, la privacy e il controllo locale** sono fondamentali. Gli alternative (OpenAI Tools, LangChain, AutoGen, CrewAI) hanno punti di forza ma avrebbero richiesto compromessi su uno di questi aspetti – ad esempio l’uso intensivo del cloud, o l’introduzione di un’ulteriore piattaforma nel progetto. Con MCP integriamo invece gli strumenti direttamente dove servono, parlando la lingua comune degli agenti.

\***OpenAI Tool Use (Function Calling & Plugins):** Prior to MCP, a primary way to enable model-tool interaction was via OpenAI’s function calling or ChatGPT’s plugin system. With function calling, e.g. GPT-4 can be given some predefined functions (with JSON schema), and the model will return a structured object when it decides to invoke that function. This technology shares similar concepts with MCP – indeed, coupling function calling with application-side execution is much like the pairing of “LLM generates request + MCP executes it.” The difference is that OpenAI’s function calling is **vendor-specific** and limited to OpenAI models (GPT-4, GPT-3.5 with functions) and those that adopt similar schema. It also doesn’t define a standard for how to execute the functions: the developer must implement each function’s execution within their app. MCP, on the other hand, standardizes the execution phase as well via a common protocol (JSON-RPC over various transports) ([Function Calling vs. Model Context Protocol (MCP): What You Need to Know - DEV Community](https://dev.to/fotiecodes/function-calling-vs-model-context-protocol-mcp-what-you-need-to-know-4nbo#:~:text=Feature%20Function%20Calling%20MCP%20,Ensures)). Another limitation of the pure OpenAI approach is that one must redefine all needed functions for each model and handle different formats for different models (e.g. how GPT formats a function call vs how perhaps Google’s PaLM would) ([Function Calling vs. Model Context Protocol (MCP): What You Need to Know - DEV Community](https://dev.to/fotiecodes/function-calling-vs-model-context-protocol-mcp-what-you-need-to-know-4nbo#:~:text=Each%20model%20formats%20function%20calls,LLMs%20by%20handling%20these%20variations)). MCP eliminates that fragmentation by enforcing one format for requests (the client side is the same for all models) and acting as a bridge between models and tools.

It’s worth noting that MCP and OpenAI function calling are not mutually exclusive; they can complement each other: one possible scenario is using function calling to *capture* the model’s intent (e.g. define a single function call\_tool(tool\_name, args)), and then inside that function’s implementation use MCP to dispatch to the actual tool. As one source explains, *“Function-calling translates prompts into actionable instructions, while MCP ensures those instructions are executed reliably and at scale.”* ([Function Calling vs. Model Context Protocol (MCP): What You Need to Know - DEV Community](https://dev.to/fotiecodes/function-calling-vs-model-context-protocol-mcp-what-you-need-to-know-4nbo#:~:text=Feature%20Function%20Calling%20MCP%20,Ensures)). So, our choice to implement MCP directly gives us maximum flexibility (not being forced to use GPT-4), at the cost of handling the call loop ourselves.

Compared to **OpenAI Plugins**, MCP is more general. Plugins require implementing a web service with OpenAI’s specific schema and registering it with ChatGPT; that makes them useful only in that ecosystem and with that model. MCP is designed to work with any compatible model and within user-controlled applications (like our VS Code extension) without needing external plugin approval. In short, MCP is **open-standard and self-hosted**, whereas OpenAI’s tools are **proprietary and cloud-hosted**. For a project like ours that emphasizes local integration and privacy, MCP offers greater control: repository code and documents don’t need to leave the local machine (they can be served via MCP locally), which would be a risk with cloud plugins.

**LangChain Agents / LangGraph:** LangChain is a popular Python library to build LLM chains and agents that use tools. It provides various “agent frameworks” where the model can decide an action (from a set of Python-defined tools) and the library executes that action, feeding the result back to the model, looping until done. Our MCP-based approach is conceptually similar, but differs in implementation and scope:

* LangChain is designed for Python and typically used in Python applications (scripts, backend servers, notebooks). In a VS Code extension context (Node.js), using it directly is awkward (would require running a Python process or server). MCP with the TS SDK integrates natively into our stack instead.
* LangChain defines tools simply as Python functions that the agent can call. That’s fine for prototypes, but at scale you again face implementing tools for each environment and model. MCP provides an ecosystem of already-built, reusable tool servers. For example, instead of writing from scratch a Python function to read files (and having to handle security logic), we can use a robust, tested FileSystem server ([Example Servers - Model Context Protocol](https://modelcontextprotocol.io/examples#:~:text=Data%20and%20file%20systems)).
* **LangGraph**, in particular, is an extension of LangChain oriented toward graph (DAG) workflows and multi-actor scenarios ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=LangGraph%20is%20an%20open,by%20the%20long%20history%20of)). It can orchestrate complex steps with nodes representing tasks, integrating advanced memory and human-in-loop options. In other words, LangGraph gives more declarative flow control. In our case, we may not need that explicitly: the nature of our system is already multi-agent and the roles define a sort of flow (dev -> test -> review). By implementing MCP, we keep the orchestration logic mostly custom within the extension. If we wanted to delegate coordination to an external framework, LangGraph could be an option but at the cost of increased complexity (introducing the need to model our whole process as a graph and integrate that execution with our UI).
* LangChain/LangGraph offer **integrations with many tools** (web search, databases, etc.), but in practice most of these are implemented as wrappers around specific APIs (e.g. a Tool that calls SerpAPI for web search). With MCP, we can reach those same APIs but through a unified channel. It’s worth noting LangChain and MCP are not incompatible: one could create an *MCP Tool for LangChain* to use MCP servers within a LangChain agent. Conversely, we might take inspiration from how LangChain formats prompts to coax the model into choosing a tool (the so-called ReAct prompting). Since we’re integrating into a product (VS Code extension), we likely prefer to craft our prompt logic ourselves for tool usage.

In summary, **LangChain/LangGraph** excel at building agents in a standalone environment (scripts or backend). **MCP** is better for integrated, modular applications and for standardizing the interface to tools. Our choice of MCP aligns with keeping integration *inside the IDE and its technology* and not depending on an external Python ecosystem. Moreover, frameworks like LangGraph might be overkill for a single-use-case project: we prefer to implement our multi-agent logic tailored to our needs, whereas LangGraph shines if you need to create many complex workflows generally.

One note: LangGraph and CrewAI also incorporate multiple tools and advanced memory ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=LangGraph%3A%20LangGraph%20integrates%20seamlessly%20with,the%20functionality%20of%20their%20agents)) ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=LangGraph%3A%20LangGraph%20provides%20built,revisit%20and%20analyze%20previous%20states)), meaning they have addressed similar problems. But from published comparisons, **“LangGraph and Crew have an edge due to their seamless integration with LangChain, offering a comprehensive range of tools, and a comprehensive memory system”** ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=match%20at%20L224%20,the%20addition%20of%20custom%20tools)) ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=Crew%20AI%3A%20Crew%20AI%20offers,recall%20important%20information%20across%20multiple)). With MCP, we are equipping our system with an analogous range of tools (via servers) and memory options (via a Memory server). So we gain similar benefits, but in a more IDE-integrated, bespoke way.

**Microsoft AutoGen:** AutoGen is Microsoft’s open-source framework for multi-agent conversational collaboration. Its core idea is to model workflows as conversations between specialized agents ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=Autogen)) ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=match%20at%20L338%20Autogen%3A%20Autogen,including%20sequential%20and%20nested%20chats)). For example, they demonstrate an “Assistant” and a “Developer” agent chatting to solve tasks. AutoGen supports *various tools (code executors, function callers, etc.)* and emphasizes ease of configuration – it’s modular and fairly easy to extend ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=Autogen%20supports%20various%20tools%2C%20including,modular%20and%20easy%20to%20maintain)) ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=match%20at%20L192%20Autogen%3A%20Autogen,It%20abstracts%20much%20of%20the)). In a sense, our AI Developer Panel is a concrete implementation of AutoGen-like concepts: we have multiple agents (developer, tester, supervisor) communicating. The difference is we are integrating them into an IDE context with specific controls. AutoGen, as a library, would require writing code to define these agents and interactions. Also, AutoGen is Python-based. If we were building our panel from scratch outside VS Code, we might have considered AutoGen as a basis for multi-agent logic, but given the existing infrastructure, it’s more efficient to extend it with MCP rather than replace it.

In terms of **tool use**, AutoGen allows agents to use tools, but typically leveraging function calling or custom integration. It doesn’t offer a standard like MCP; we would still need to wire up file/doc access similar to how we did, just within AutoGen’s framework. On the other hand, one advantage of AutoGen is handling nested conversations and sub-tasks elegantly, and having extensive documentation with examples ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=match%20at%20L301%20Autogen%3A%20Autogen,of%20key%20concepts%20and%20features)). Our project, however, focuses on code development, so the roles and their interplay are already clearly defined and static – we are not dynamically spawning and retiring agents on the fly (which AutoGen can do, e.g. spawn a “Debugger” agent temporarily).

**CrewAI:** CrewAI is another multi-agent framework aiming at role-based AI team collaboration. It emphasizes assigning specific roles and goals to each agent and enabling them to operate as a cohesive unit ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=Crew%20AI)). Our scenario indeed has a team of role-based agents. CrewAI provides structures for flexible task management, autonomous delegation among agents, and tightly integrates **LangChain** for tools and memory ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=match%20at%20L221%20Crew%20AI%3A,tailored%20to%20their%20specific%20needs)). From comparisons: CrewAI has strong memory capabilities (short, long-term, entity memory) and supports structured outputs (e.g. agents can share info as JSON via Pydantic models) ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=Crew%20AI%3A%20Crew%20AI%20offers,recall%20important%20information%20across%20multiple)) ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=Crew%20AI%3A%20Crew%20AI%20supports,to%20meet%20their%20specific%20requirements)). This reduces miscommunication between agents.

Our panel, enhanced with MCP, achieves similar goals:

* Clearly defined roles (hard-coded, similar to CrewAI’s concept of role assignment);
* Shared memory if we use the Memory server, covering short-term (chat) and long-term (knowledge base) context, comparable to CrewAI’s “comprehensive memory system” ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=Crew%20AI%3A%20Crew%20AI%20offers,recall%20important%20information%20across%20multiple));
* Ability to structure communication: if needed, our agents could exchange JSON too (perhaps using the Memory server as a bulletin board, or via our orchestration logic). CrewAI formalizes it more, but nothing prevents us from implementing conventions, e.g. we could decide the Tester reports findings as structured data that the Developer agent parses.

One strength of CrewAI is ease of use: “very straightforward to get started” with how-to guides and examples ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=match%20at%20L198%20Crew%20AI%3A,approach%20to%20defining%20and%20managing)) ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=Documentation)). In our case, however, we have custom needs (IDE integration, Telegram, etc.), so a general framework might not cover everything without heavy customization.

**Comparison summary:** The MCP integration we’re adopting positions our project flexibly among these solutions. MCP provides an **open standard** and ready implementations for common tools (versus having to write specific plugins as with OpenAI). Unlike LangChain/LangGraph, our solution requires no heavy external dependencies and remains entirely within the extension’s JavaScript/Node domain, while still benefiting from similar agent/tool orchestration concepts. Compared to multi-agent frameworks like AutoGen and CrewAI, our implementation is more focused: those frameworks are generic and meant to support any agent configuration, whereas we know exactly what roles and interactions we want and can code them deterministically using MCP as the medium for actions. In return, we get a system that’s more controllable and tightly integrated with the IDE.

It’s worth noting that since MCP is relatively new, the community is still converging on best practices; but prospects are promising given many vendors and open-source contributors are adopting it (there’s an official servers repo and a growing “awesome” community list ([Adding MCP Servers from GitHub | Cline](https://docs.cline.bot/mcp-server-from-github#:~:text=,MCP%20servers%20on%20GitHub%20include))). Adopting MCP now gives our project a future-proof setup: if tomorrow OpenAI or others natively support MCP, our panel would benefit immediately. In contrast, tying to proprietary solutions might require rework if APIs or policies change.

In conclusion, **MCP** stands out as the right choice for a development tool like ours, where **modularity, privacy, and local control** are paramount. The alternatives (OpenAI Tools, LangChain, AutoGen, CrewAI) each have strengths but would have forced trade-offs in one of those aspects – e.g. heavier cloud reliance or introducing another platform into the mix. With MCP we integrate the tools directly where we need them, speaking the agents’ common language in a standardized way.\*

## **Riferimenti Utili / *Key References and Resources***

* **Introduzione a MCP – Anthropic**: *Introducing the Model Context Protocol* – Panoramica del protocollo e motivazioni ([Introduction - Model Context Protocol](https://modelcontextprotocol.io/introduction#:~:text=MCP%20is%20an%20open%20protocol,different%20data%20sources%20and%20tools)) ([Introduction - Model Context Protocol](https://modelcontextprotocol.io/introduction#:~:text=LLMs,different%20data%20sources%20and%20tools)).
* **Documentazione Ufficiale MCP** (spec e SDK): disponibile su *modelcontextprotocol.io*, incluse guide per sviluppare server e client MCP ([GitHub - modelcontextprotocol/typescript-sdk: The official Typescript SDK for Model Context Protocol servers and clients](https://github.com/modelcontextprotocol/typescript-sdk#:~:text=The%20Model%20Context%20Protocol%20allows,specification%2C%20making%20it%20easy%20to)) ([GitHub - modelcontextprotocol/typescript-sdk: The official Typescript SDK for Model Context Protocol servers and clients](https://github.com/modelcontextprotocol/typescript-sdk#:~:text=import%20,zod)).
* **Repository GitHub MCP**:  
  + *Model Context Protocol Specification* ([Introduction - Model Context Protocol](https://modelcontextprotocol.io/introduction#:~:text=At%20its%20core%2C%20MCP%20follows,can%20connect%20to%20multiple%20servers)) e
  + *TypeScript SDK* ufficiale ([GitHub - modelcontextprotocol/typescript-sdk: The official Typescript SDK for Model Context Protocol servers and clients](https://github.com/modelcontextprotocol/typescript-sdk#:~:text=The%20Model%20Context%20Protocol%20allows,specification%2C%20making%20it%20easy%20to)) (contiene esempi di quickstart per server e client).
* **Esempi di Server MCP**: Elenco di server di riferimento (FileSystem, Git, GitHub, Memory, ecc.) sul sito MCP ([Example Servers - Model Context Protocol](https://modelcontextprotocol.io/examples#:~:text=Data%20and%20file%20systems)) ([Example Servers - Model Context Protocol](https://modelcontextprotocol.io/examples#:~:text=Development%20tools)).
* **Cline Docs – MCP**: Sezioni su come Cline integra MCP, tra cui *MCP Servers* ([MCP Overview | Cline](https://docs.cline.bot/mcp-servers/mcp#:~:text=Integration%20with%20Cline)) ([MCP Overview | Cline](https://docs.cline.bot/mcp-servers/mcp#:~:text=Using%20MCP%20Servers)) e guida *Adding MCP Servers from GitHub* ([Adding MCP Servers from GitHub | Cline](https://docs.cline.bot/mcp-server-from-github#:~:text=There%20are%20multiple%20places%20online,to%20find%20MCP%20servers)).
* **Confronto Function Calling vs MCP**: Articolo di approfondimento ([Function Calling vs. Model Context Protocol (MCP): What You Need to Know - DEV Community](https://dev.to/fotiecodes/function-calling-vs-model-context-protocol-mcp-what-you-need-to-know-4nbo#:~:text=Feature%20Function%20Calling%20MCP%20,Ensures)) ([Function Calling vs. Model Context Protocol (MCP): What You Need to Know - DEV Community](https://dev.to/fotiecodes/function-calling-vs-model-context-protocol-mcp-what-you-need-to-know-4nbo#:~:text=Essentially%2C%20Function%20Calling%20is%20about,driven%20software%20automation%20runs%20efficiently)) (Dev.to) che spiega differenze di ruolo tra i due.
* **Framework Multi-Agente**:  
  + Confronto *LangGraph vs Autogen vs CrewAI* (Galileo AI) ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=LangGraph%3A%20LangGraph%20integrates%20seamlessly%20with,the%20functionality%20of%20their%20agents)) ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=Crew%20AI%3A%20Crew%20AI%20offers,recall%20important%20information%20across%20multiple)),
  + Blog Microsoft su *AutoGen* ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=Autogen)) ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=match%20at%20L338%20Autogen%3A%20Autogen,including%20sequential%20and%20nested%20chats)),
  + Documentazione CrewAI (crew.ai) e discussione Reddit comparativa ([CrewAI vs AutoGen for Code Execution AI Agents - Reddit](https://www.reddit.com/r/AutoGenAI/comments/1asf2zj/crewai_vs_autogen_for_code_execution_ai_agents/#:~:text=CrewAI%20vs%20AutoGen%20for%20Code,an%20edge%20in%20language%20understanding)).
* **Code Example MCP TS**: *MCP Server-Client Example* su GitHub ([GitHub - JoeBuildsStuff/mcp-server-client](https://github.com/joeBlockchain/mcp-server-client#:~:text=MCP%20Server)) ([GitHub - JoeBuildsStuff/mcp-server-client](https://github.com/joeBlockchain/mcp-server-client#:~:text=npm%20run%20start%3Aserver)) – dimostrazione minima di client e server in TypeScript.

**Nota conclusiva:** Implementare MCP aggiunge complessità iniziale, ma allunga di molto le potenzialità del pannello AI. Una volta configurati i server e istruito il modello, gli agenti potranno sfruttare un “ecosistema” di risorse molto più ricco rispetto al solo prompt statico. Ciò si tradurrà in una capacità di comprendere e modificare progetti più grandi, di mantenere conoscenza tra sessioni e di interagire con altri sistemi (come issue tracker, ecc. potenzialmente via MCP) senza dover codificare caso per caso. In un ambiente come Visual Studio Code, questa integrazione promette di fare del pannello AI un assistente ancora più **autonomo, collaborativo e contestualmente consapevole**.

\*\*\* ([Introduction - Model Context Protocol](https://modelcontextprotocol.io/introduction#:~:text=MCP%20is%20an%20open%20protocol,different%20data%20sources%20and%20tools)) ([MCP Overview | Cline](https://docs.cline.bot/mcp-servers/mcp#:~:text=MCP%20servers%20act%20as%20intermediaries,that%20an%20LLM%20can%20use)) ([Cline - AI Autonomous Coding Agent for VS Code](https://cline.bot/#:~:text=Extends%20your%20reach%20with%20MCP,external%20databases%20and%20live%20docs)) ([Cline - AI Autonomous Coding Agent for VS Code](https://cline.bot/#:~:text=Access%20top,while%20keeping%20your%20code%20secure)) ([Function Calling vs. Model Context Protocol (MCP): What You Need to Know - DEV Community](https://dev.to/fotiecodes/function-calling-vs-model-context-protocol-mcp-what-you-need-to-know-4nbo#:~:text=Feature%20Function%20Calling%20MCP%20,Ensures)) ([Function Calling vs. Model Context Protocol (MCP): What You Need to Know - DEV Community](https://dev.to/fotiecodes/function-calling-vs-model-context-protocol-mcp-what-you-need-to-know-4nbo#:~:text=Essentially%2C%20Function%20Calling%20is%20about,driven%20software%20automation%20runs%20efficiently)) ([GitHub - modelcontextprotocol/typescript-sdk: The official Typescript SDK for Model Context Protocol servers and clients](https://github.com/modelcontextprotocol/typescript-sdk#:~:text=The%20Model%20Context%20Protocol%20allows,specification%2C%20making%20it%20easy%20to)) ([GitHub - modelcontextprotocol/typescript-sdk: The official Typescript SDK for Model Context Protocol servers and clients](https://github.com/modelcontextprotocol/typescript-sdk#:~:text=%2F%2F%20Add%20an%20addition%20tool,)) ([GitHub - JoeBuildsStuff/mcp-server-client](https://github.com/joeBlockchain/mcp-server-client#:~:text=MCP%20Server)) ([GitHub - JoeBuildsStuff/mcp-server-client](https://github.com/joeBlockchain/mcp-server-client#:~:text=npm%20run%20start%3Aserver)) ([Example Servers - Model Context Protocol](https://modelcontextprotocol.io/examples#:~:text=Development%20tools)) ([Example Servers - Model Context Protocol](https://modelcontextprotocol.io/examples#:~:text=%2A%20Google%20Maps%20,based%20persistent%20memory%20system)) ([Example Servers - Model Context Protocol](https://modelcontextprotocol.io/examples#:~:text=%2A%20Brave%20Search%20,automation%20and%20web%20scraping%20capabilities)) ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=LangGraph%3A%20LangGraph%20integrates%20seamlessly%20with,the%20functionality%20of%20their%20agents)) ([Mastering Agents: LangGraph Vs Autogen Vs Crew AI - Galileo AI](https://www.galileo.ai/blog/mastering-agents-langgraph-vs-autogen-vs-crew#:~:text=Crew%20AI%3A%20Crew%20AI%20offers,recall%20important%20information%20across%20multiple))\*\*\*