Hybrid FastMCP and Lastmile Agent Integration for ALAN IDE

**Integrating FastMCP with Lastmile’s MCP-Agent for ALAN IDE Orchestration**

*High-level MCP server architecture illustrating* ***Resources****,* ***Tools****, and* ***Prompts*** *accessible to an LLM*[*learnbybuilding.ai*](https://learnbybuilding.ai/post/creating-a-mcp-server-to-run-a-crm#:~:text=This%20architectural%20diagram%20provides%20an,of%20this%20conversational%20AI%20system)*. In our design, the FastMCP server provides this tool layer, while the Lastmile agent framework adds an orchestration layer on top.*

**Architecture Overview**

**ALAN IDE’s AI orchestration** is built as a two-layer system: a **tool layer** (FastMCP server) and an **agent layer** (Lastmile’s MCP-Agent framework). This design follows the Model Context Protocol (MCP) philosophy of separating tool integrations from “agent brain” logic[fleak.ai](https://fleak.ai/blog/mcp-intelligence#:~:text=scalable). The FastMCP server exposes core IDE capabilities (file system, code analysis, execute commands, etc.) as standardized MCP tools. On top of this, the Lastmile **mcp-agent** framework manages high-level agent behaviors – coordinating when and why tools are invoked to achieve complex tasks (multi-step workflows, AI-assisted refactoring, CI pipelines, etc.).

**Key Components in the Architecture:**

* **FastMCP Tool Server (Backend)** – A Python MCP server providing IDE functions as **tools** (e.g. file I/O, semantic queries, run commands) and data **resources** (e.g. file contents, code graphs). It uses the FastMCP SDK to define tools with Python functions and serves them via the MCP protocol[modelcontextprotocol.io](https://modelcontextprotocol.io/quickstart/server#:~:text=The%20FastMCP%20class%20uses%20Python,create%20and%20maintain%20MCP%20tools).
* **Agent Orchestration Layer** – The **Lastmile mcp-agent** framework running within the backend, which implements the “agent brain.” It can run **Agents** (logical AI assistants) that use the MCP tools to perform tasks. This layer implements patterns from Anthropic’s *Building Effective Agents* (e.g. multi-step reasoning, tool-using ReAct loops, evaluators)[github.com](https://github.com/lastmile-ai/mcp-agent#:~:text=Inspiration%3A%20Anthropic%20announced%202%20foundational,updates%20for%20AI%20application%20developers)[fleak.ai](https://fleak.ai/blog/mcp-intelligence#:~:text=scalable).
* **Large Language Model (LLM)** – An AI model (Claude, GPT-4, etc.) that powers the agent’s reasoning and language generation. The LLM is invoked via an *Augmented LLM interface* provided by mcp-agent, which allows the LLM to call MCP tools through the agent[github.com](https://github.com/lastmile-ai/mcp-agent#:~:text=match%20at%20L580%20AugmentedLLM%20is,servers%20and%20functions%20via%20Agents). The LLM might run remotely (via API) but communicates with the tool layer through the MCP protocol.
* **IDE Frontend (ALAN UI)** – A Monaco/CodeMirror-based client application (running in browser or electron) that developers interact with. It connects to the backend’s MCP server to send agent requests (e.g. “Refactor this code”) and receives real-time streaming feedback (via Server-Sent Events or WebSockets). The frontend uses standard MCP message formats to remain compatible with any MCP client or model integration.

**Integration Summary:** The FastMCP server acts as the unified “toolbox” for the AI agent, while the Lastmile framework is the “conductor” deciding which tools to use and in what sequence[fleak.ai](https://fleak.ai/blog/mcp-intelligence#:~:text=scalable). This separation of concerns makes the system modular – the tool layer can be extended independently, and the agent layer can implement complex logic (or even multiple cooperating agents) without altering how low-level tools work. The result is a flexible AI-driven IDE: the agent layer orchestrates multi-step coding tasks by calling FastMCP-provided tools, and the IDE UI stays updated with the agent’s progress in real time.

**FastMCP Server – Exposing IDE Tools via MCP**

Setting up the FastMCP server is the first step. FastMCP is a Python SDK that greatly simplifies building MCP servers[modelcontextprotocol.io](https://modelcontextprotocol.io/quickstart/server#:~:text=The%20FastMCP%20class%20uses%20Python,create%20and%20maintain%20MCP%20tools). We will use it to expose core IDE functions as MCP **tools** and relevant data as **resources**.

**1. Installation and Setup:** Install the required packages in a Python 3.11+ environment (we use the uv tool for convenience as recommended, or pip directly):

bash

CopyEdit

pip install fastmcp mcp-agent # install FastMCP and Lastmile’s agent framework

# (Alternatively, use `uv`: uv add "mcp[cli]" "mcp-agent")

In your server script (e.g. alan\_ide\_server.py), initialize the FastMCP server:

python

CopyEdit

from mcp.server.fastmcp import FastMCP

mcp = FastMCP("alan\_ide") # Give the MCP server a name (used for identification)

This creates an MCP server instance named “alan\_ide.” FastMCP will automatically generate protocol-compliant tool definitions for any functions we register, using Python type hints and docstrings as metadata[modelcontextprotocol.io](https://modelcontextprotocol.io/quickstart/server#:~:text=The%20FastMCP%20class%20uses%20Python,create%20and%20maintain%20MCP%20tools).

**2. Defining Tools:** We expose IDE operations by decorating Python functions with @mcp.tool(). Each such function becomes an MCP-accessible tool. For example, we can define a couple of fundamental IDE tools: reading a file, writing a file, and executing a shell command:

python

CopyEdit

@mcp.tool()

def read\_file(path: str) -> str:

"""Read the contents of a file given its path."""

with open(path, 'r') as f:

return f.read()

@mcp.tool()

def write\_file(path: str, content: str) -> str:

"""Overwrite a file with the given content and return a success message."""

with open(path, 'w') as f:

f.write(content)

return f"File {path} updated."

@mcp.tool()

def run\_command(command: str) -> str:

"""Execute a shell command and return its output (or error)."""

import subprocess, shlex

try:

result = subprocess.check\_output(shlex.split(command), stderr=subprocess.STDOUT, timeout=10)

return result.decode('utf-8')

except subprocess.CalledProcessError as e:

return e.output.decode('utf-8')

Each tool’s docstring and signature will be exposed through MCP, so the agent (or an LLM client) knows what the tool does and how to call it. For instance, the read\_file tool accepts a file path and returns the file’s text; its description helps the AI understand its purpose. FastMCP’s introspection uses these to advertise the tool to the LLM[modelcontextprotocol.io](https://modelcontextprotocol.io/quickstart/server#:~:text=The%20FastMCP%20class%20uses%20Python,create%20and%20maintain%20MCP%20tools).

We can similarly add tools for querying a semantic code graph (if the IDE has one), running tests, retrieving documentation, etc. **Resources** (static data) can be added if needed – for example, a resource might be the entire workspace’s file list or an index of symbols. In FastMCP, resources are typically provided by adding them to the server instance (not shown here, but conceptually similar). **Prompts** (pre-defined prompt templates) can also be registered if we want the LLM to have canned instructions for common tasks[learnbybuilding.ai](https://learnbybuilding.ai/post/creating-a-mcp-server-to-run-a-crm#:~:text=MCP%20servers%20can%20offer%20three,primary%20types%20of%20capabilities), though we focus on tools for now.

**3. Running the MCP Server:** Finally, start the server and choose a transport that supports our use case. For IDE integration and web clients, an HTTP-based transport is appropriate (so the frontend and external LLMs can connect). We’ll use **Server-Sent Events (SSE)** for streaming responses. FastMCP supports SSE out of the box:

python

CopyEdit

if \_\_name\_\_ == "\_\_main\_\_":

mcp.run(transport="sse", port=8080)

This launches the MCP server on localhost:8080, exposing an SSE endpoint (commonly at /sse) for clients to connect and a corresponding endpoint for sending requests. The SSE transport ensures that results from tools can be streamed back to the client incrementally[blog.cloudflare.com](https://blog.cloudflare.com/ar-ar/streamable-http-mcp-servers-python/#:~:text=Initially%2C%20remote%20MCP%20communication%20between,client%20sends%20requests%20for%20tool) – crucial for real-time feedback.

**At this stage,** we have a running FastMCP server with IDE tools. It’s a self-contained “unified toolbox” accessible via MCP[fleak.ai](https://fleak.ai/blog/mcp-intelligence#:~:text=scalable). An LLM or agent connecting to this server can discover the tools and invoke them securely. However, on its own, this tool layer doesn’t decide *when* to call tools or *how* to break down complex tasks – that’s the role of our agent layer.

**Agent Layer – Lastmile’s mcp-agent Integration**

The **mcp-agent** framework by Lastmile provides the orchestration logic on top of MCP. It manages the **Agent** – an entity that can plan and execute multi-step tool usage with the help of an LLM. We will integrate mcp-agent into our FastMCP server process so that an agent can be invoked as part of handling certain MCP requests or user commands.

**1. Initializing the Agent App:** We create an MCPApp which represents the agent application and manages connections to MCP servers:

python

CopyEdit

from mcp\_agent.app import MCPApp

app = MCPApp(name="alan\_agents")

The MCPApp handles configuration (like API keys, logging, etc.) and the lifecycle of MCP server connections[trendshift.io](https://trendshift.io/admin/repository/ask-ai/13216#:~:text=agents.%20%60mcp,contributions%5D%28%2FCONTRIBUTING.md)[github.com](https://github.com/lastmile-ai/mcp-agent#:~:text=match%20at%20L559%20,a%20set%20of%20MCP%20servers). We need to tell it how to connect to our FastMCP server. Since our tool server is running locally (and even within the same process in this setup), we can register it with the agent app. The mcp-agent library provides a **connection manager** and a gen\_client utility to connect to MCP servers[github.com](https://github.com/lastmile-ai/mcp-agent#:~:text=,a%20set%20of%20MCP%20servers).

For example, if the FastMCP server is accessible at localhost:8080 with SSE, we might do:

python

CopyEdit

from mcp\_agent.mcp.gen\_client import MCPClientParams

# Register the local FastMCP server (via SSE URL or local connection)

app.mcp\_conn\_manager.add\_server(

name="ide\_tools",

client\_params=MCPClientParams(

transport="sse", host="localhost", port=8080

)

)

Here we add a server connection named "ide\_tools" pointing to our FastMCP service. Now the agent framework knows how to reach the IDE tools server. (If the agent runs in the same process, an alternative is using transport="stdio" for an in-process connection[blog.cloudflare.com](https://blog.cloudflare.com/ar-ar/streamable-http-mcp-servers-python/#:~:text=Initially%2C%20remote%20MCP%20communication%20between,client%20sends%20requests%20for%20tool), but SSE over localhost works as well.)

**2. Defining an Agent:** An **Agent** in mcp-agent represents a distinct autonomous assistant with access to certain tools. We can define agents for different roles or workflows. For example, a simple “finder” agent might just retrieve information, whereas a “refactor” agent will modify code. We create an agent by specifying its name, an optional role/instruction prompt, and which MCP servers (tool collections) it can use:

python

CopyEdit

from mcp\_agent.agents.agent import Agent

refactor\_agent = Agent(

name="refactor",

instruction="You are a code refactoring agent. Improve code as requested using the IDE tools.",

server\_names=["ide\_tools"] # this agent can use tools from the "ide\_tools" server we registered

)

This sets up an agent with a certain persona (“code refactoring agent”) and grants it access to all tools on our ide\_tools server (which includes read\_file, write\_file, etc.). We could create other agents similarly (e.g., a ci\_agent with instructions for running tests and analyzing failures, or an explainer agent that only reads code and explains it). Each agent is isolated in terms of the tools it can see, which is good for modular design.

**3. Attaching an LLM to the Agent:** Lastmile’s framework uses an **AugmentedLLM** abstraction to interface with actual language models[github.com](https://github.com/lastmile-ai/mcp-agent#:~:text=match%20at%20L580%20AugmentedLLM%20is,servers%20and%20functions%20via%20Agents). We choose an LLM provider and attach it to the agent. For instance, to use OpenAI’s GPT-4 via their API:

python

CopyEdit

from mcp\_agent.workflows.llm.augmented\_llm\_openai import OpenAIAugmentedLLM

async with refactor\_agent: # initialize connections

llm = await refactor\_agent.attach\_llm(OpenAIAugmentedLLM)

# Now we can use llm to generate responses with tool use

The attach\_llm(OpenAIAugmentedLLM) call wraps the GPT-4 model with an “augmented” interface that knows how to utilize MCP tools. Under the hood, this likely provides the model a description of available tools and intercepts the model’s outputs to execute tool calls when needed[github.com](https://github.com/lastmile-ai/mcp-agent#:~:text=match%20at%20L580%20AugmentedLLM%20is,servers%20and%20functions%20via%20Agents). (For Anthropic’s Claude, one would use AnthropicAugmentedLLM similarly[github.com](https://github.com/lastmile-ai/mcp-agent#:~:text=AnthropicAugmentedLLM). The choice is abstracted by the AugmentedLLM class – it’s **model-agnostic**, as the framework emphasizes[github.com](https://github.com/lastmile-ai/mcp-agent#:~:text=Each%20pattern%20is%20model,making%20everything%20very%20composable).)

**4. Agent Logic Patterns:** By default, an agent with an AugmentedLLM will follow a ReAct-style loop: the LLM can output an action (tool invocation) which the framework executes, then the LLM sees the result, and so on, until a final answer is produced[trendshift.io](https://trendshift.io/admin/repository/ask-ai/13216#:~:text=filesystem%20or%20fetch%20URLs%20finder_agent,initializes%20the%20MCP%20servers%20and)[trendshift.io](https://trendshift.io/admin/repository/ask-ai/13216#:~:text=AnthropicAugmentedLLM%20finder_agent%20%3D%20Agent%28%20name%3D,attach_llm%28AnthropicAugmentedLLM%29%20result%20%3D%20await). Lastmile’s mcp-agent also implements more advanced **workflow patterns** (from Anthropic’s *Building Effective Agents*) like parallel reasoning, decision routers, self-evaluators, etc.[trendshift.io](https://trendshift.io/admin/repository/ask-ai/13216#:~:text=Agent%28name%3D,Student%20short%20story%20submission%3A)[trendshift.io](https://trendshift.io/admin/repository/ask-ai/13216#:~:text=Agent%28name%3D,Load%20short_story.md). For example, one can create a **ParallelLLM** agent that runs multiple sub-agents (proofreader, fact\_checker, etc.) concurrently and then combines results[trendshift.io](https://trendshift.io/admin/repository/ask-ai/13216#:~:text=%3Csummary%3EExample%3C%2Fsummary%3E%20%60%60%60python%20proofreader%20%3D%20Agent%28name%3D,grader). These patterns can be composed, but for our IDE use cases, a straightforward sequential approach (possibly with loops for retries) is often sufficient.

**Embedding Agent Orchestration as a Tool:** We have two ways to invoke these agents in the system:

* *Direct invocation via code (e.g. when the user clicks “Run Refactor Agent” in the UI, we call the agent’s logic).*
* *Invocation via the LLM itself as an MCP tool.* For instance, we could register an MCP **tool** on FastMCP called run\_refactor that, when called, internally triggers the refactor\_agent to run its process. This second approach makes the agent accessible to other agents or the LLM in a conversation (the LLM could decide to delegate a subtask to a higher-level agent via this tool).

To illustrate, we embed the refactoring agent logic inside a FastMCP tool handler:

python

CopyEdit

@mcp.tool(name="agent\_refactor", description="Refactor code in a file based on given instructions.")

async def agent\_refactor(file\_path: str, goal: str) -> str:

"""High-level agent tool: refactor the specified file to achieve the given goal."""

async with refactor\_agent: # ensure agent’s connections (to IDE tools, LLM) are ready

llm = await refactor\_agent.attach\_llm(OpenAIAugmentedLLM)

# Step 1: Agent reads the file content using a tool

file\_content = await llm.call\_tool("read\_file", file\_path) # instruct agent/LLM to use read\_file

# Step 2: Ask LLM to refactor the content

user\_prompt = f"Refactor the following code to {goal}:\n```{file\_content}```"

refactored\_code = await llm.generate\_str(user\_prompt)

# Step 3: Write the refactored code back to file

await llm.call\_tool("write\_file", file\_path, refactored\_code)

return f"Refactoring complete for {file\_path}"

In this pseudo-code, llm.call\_tool() explicitly invokes a tool via the agent’s connection (this is one way to do it – alternatively, we could rely on the LLM to decide to call read\_file itself from the prompt; we show an explicit call for clarity). The agent reads the file, the LLM generates a refactored version, and then the agent writes it back. The whole operation is packaged as a single MCP tool call agent\_refactor from the perspective of an outside client. This demonstrates **how we embed mcp-agent logic inside a FastMCP tool handler** – the handler coordinates multiple internal tool calls and LLM interactions to fulfill a high-level request.

**Agent Lifecycle:** The agent’s lifecycle is managed via the context manager (async with refactor\_agent), which on enter will automatically initialize connections to the ide\_tools server (via our MCPConnectionManager)[trendshift.io](https://trendshift.io/admin/repository/ask-ai/13216#:~:text=filesystem%20or%20fetch%20URLs%20finder_agent,initializes%20the%20MCP%20servers%20and) and tear them down on exit. The Lastmile framework abstracts away the low-level connection handling, so we can focus on the agent’s logic[trendshift.io](https://trendshift.io/admin/repository/ask-ai/13216#:~:text=agents.%20%60mcp,contributions%5D%28%2FCONTRIBUTING.md).

**Workflow: Lifecycle of an Agent-Driven Request**

Let’s walkthrough what happens when an agent-powered action is triggered in ALAN IDE. This could be initiated by a user (clicking a button or issuing a command in the UI) or by an LLM during a chat. We’ll consider a user-initiated example for clarity:

1. **User Triggers an Agent Task:** Suppose the developer chooses “AI Refactor” on a file. The IDE frontend will send a request to the backend to start the refactoring agent. This might be done by calling the agent\_refactor MCP tool via an HTTP request, or by a custom API endpoint that invokes the agent code directly. In our unified design, the simplest way is to call the MCP tool:
   * The frontend could issue a POST to the MCP server’s /sse/messages endpoint with a message instructing agent\_refactor to run, or use a lightweight MCP client library in JS to do the same. For example, sending a JSON like {"action": "call", "tool": "agent\_refactor", "params": {"file\_path": "src/foo.py", "goal": "improve performance"}}.
2. **MCP Server Dispatches to Tool Handler:** The FastMCP server receives the request and recognizes it as a call to the agent\_refactor tool (registered earlier). It invokes our handler agent\_refactor(...) in the Python backend. This handler now executes the agent orchestration logic using mcp-agent as described:
   * It initializes the refactor\_agent (connecting to the tools if not already connected).
   * It attaches the LLM and sends the prompt (which includes reading the file content as needed).
   * **Streaming Feedback:** As the agent works, partial results can be sent back. For instance, after the file is read, the agent might stream a message like “Read 200 lines from foo.py…”. When the LLM is generating the refactored code, we could stream tokens of that code as they come (if using a streaming LLM API). Each of these updates can be sent over the SSE connection to the frontend. The FastMCP framework allows the tool handler to emit intermediate results to the SSE response stream – e.g., by yielding results or using the stream=True flag in the protocol[blog.cloudflare.com](https://blog.cloudflare.com/ar-ar/streamable-http-mcp-servers-python/#:~:text=support%20both%20the%20existing%20Server,new%20Streamable%20HTTP%20transport%20concurrently)[blog.cloudflare.com](https://blog.cloudflare.com/ar-ar/streamable-http-mcp-servers-python/#:~:text=,been%20kept%20as%20an%20alias). The Cloudflare MCP spec update introduced “streamable HTTP” which unifies request/response streaming, but SSE is sufficient for our needs[blog.cloudflare.com](https://blog.cloudflare.com/ar-ar/streamable-http-mcp-servers-python/#:~:text=The%20MCP%20spec%20was%20updated,connection%20and%20for%20sending%20messages)[blog.cloudflare.com](https://blog.cloudflare.com/ar-ar/streamable-http-mcp-servers-python/#:~:text=While%20most%20MCP%20clients%20haven%E2%80%99t,or%20the%20new%20transport%20method).
   * The agent might call multiple tools or LLM prompts in a loop until the goal is achieved. Each tool invocation is executed via the FastMCP server (since the agent is connected to it). For example, read\_file returns the data (the agent gets it via the MCP client), then later write\_file is called. These internal calls do not need separate user approval because they are initiated by the orchestrator agent, which we trust for this workflow.
3. **Agent Completes and Returns:** Once the refactoring is done, the agent\_refactor tool handler returns a final message (e.g. “Refactoring complete”). The FastMCP server sends this as the final SSE event or response to the frontend. At this point, the frontend might refresh the file content (which has been modified) in the editor. The SSE connection for this request can be closed.
4. **UI Updates:** The IDE frontend, having listened to the SSE stream, displays the agent’s feedback in real-time. For example, it might show a live log:
   * “Agent is reading *src/foo.py*…”
   * “Agent: I propose the following changes… [diff or code snippet]”
   * “Agent is writing changes to *src/foo.py*…”
   * “Refactoring complete.”

Because the protocol is standardized, any MCP-compliant client could handle these messages. In our case, using SSE means the browser receives events which JavaScript can append to an output panel. (If using WebSockets, the mechanism is similar but bidirectional; SSE is one-way from server to client, which is acceptable here since the initial trigger came from the user via HTTP.)

1. **Agent Lifecycle Cleanup:** The agent context manager ensures all tool connections are closed after use, freeing resources. The FastMCP server remains running, ready for the next request – which could be another agent invocation or a direct tool call.

**Note on LLM-driven workflow:** If the user were chatting with an AI in the IDE (say “Find and fix TODOs in my code”), the LLM (Claude, etc.) itself could decide to call tools. In that scenario, the LLM (via its integration) connects to the FastMCP server and calls tools like read\_file directly[trendshift.io](https://trendshift.io/admin/repository/ask-ai/13216#:~:text=world%21,route%28). Our architecture still supports that – the FastMCP server is multi-client and can serve both our internal orchestrator and external LLM calls concurrently. The mcp-agent layer isn’t necessarily involved in a simple LLM direct tool use (the LLM itself acts as an agent). But for complex, multi-step tasks that exceed a single LLM prompt’s scope, our agent layer can be invoked. In fact, the LLM could even call the agent\_refactor tool, effectively outsourcing a subtask to the orchestrator agent. This demonstrates a powerful capability: **agents calling other agents.** The design allows recursive tool use and even multi-agent collaboration, though such scenarios should be managed to avoid confusion.

**Extensibility via a Plugin System**

To **enable modular expansion**, we design the system so new tools and agents can be added easily – ideally without modifying the core server code. This can be achieved with a plugin architecture:

* **Tool Plugins:** We can create a directory (e.g. plugins/tools/) where each Python module defines additional @mcp.tool functions. On startup, the server scans this directory and imports each module. The act of importing registers the tool via the decorator on the global mcp instance. For example, a plugin graph\_tools.py might add a query\_graph() tool for semantic code queries. As long as it runs @mcp.tool() on a function, that tool becomes available. FastMCP will automatically include it in the MCP schema sent to the LLM client (tools are discoverable at runtime).
* **Agent Plugins:** Similarly, we can allow new agent workflows to be added. An agent plugin might define a new Agent with specific instructions and possibly register an MCP tool handler that invokes that agent (similar to our agent\_refactor). For instance, a ci\_agent\_plugin.py could define a ci\_agent = Agent(name="ci", instruction="Continuously integrate and fix tests", server\_names=["ide\_tools"]) and a tool run\_ci\_pipeline() that uses this agent. By placing this in the plugins and loading it, the system gains a new high-level capability.
* **Registration Mechanism:** We can implement a simple plugin loader in the main server script:

python

CopyEdit

import importlib, pathlib

plugin\_dir = pathlib.Path(\_\_file\_\_).parent / "plugins"

for plugin\_path in plugin\_dir.glob("\*.py"):

importlib.import\_module(f"plugins.{plugin\_path.stem}")

Each plugin module might contain both FastMCP tool definitions and use the app (MCPApp) to define new agents or connect to external MCP servers if needed. As long as they follow the conventions, they will be integrated. We should document a clear interface for plugin authors (e.g. they can access the global mcp and app objects to register tools and agents).

* **Dynamic vs Static:** Truly dynamic (hot-plugging at runtime) is complex, so our approach loads plugins on startup. This is typically sufficient – adding a plugin then restarting the server is a one-command process. Because the system is local-first and simple to run, this is acceptable.

Using this plugin system, third-party or user-contributed extensions can provide new **tools** (maybe connecting to external APIs, or advanced code analysis functions) or new **agent behaviors** (for example, a “Documentation Generator” agent that writes docstrings for all functions by reading the code). The decoupling ensures that adding a new tool automatically makes it available to all agents (if their instructions/server access allow), and adding a new agent doesn’t interfere with existing ones.

**Example Workflows Enabled by the System**

Let’s explore a few high-level AI workflows in ALAN IDE that this architecture supports, and how they function under the hood:

**AI-Led Code Refactoring Example**

**Goal:** Use AI to refactor a piece of code for improvement (performance, readability, etc.).

* **Trigger:** Developer selects a function or file and requests “AI Refactor”. They may optionally describe the refactoring goal (e.g. “optimize this for speed”).
* **Agent Execution:** The refactor\_agent is invoked (via the agent\_refactor tool as described earlier).
  1. The agent reads the target code (read\_file tool).
  2. LLM analyzes the code and proposes changes. This might be done in one shot or iterative: e.g., the LLM could identify sections to change, possibly ask for clarification or perform small edits one by one. For simplicity, assume it produces a complete refactored version.
  3. The agent writes the new code back (write\_file tool). It might also use a diff tool (if available) to generate a diff for the user to review.
* **Streaming Feedback:** The IDE UI shows each step as it happens (reading file, refactoring in progress, writing file). When done, it could show a summary of changes or the diff.
* **Result:** The source file is updated in the editor. The developer can review the changes (since they are applied in the workspace, possibly highlighted). Because this happened via the local tool layer, it respects file locks, version control, etc., as if the developer edited it – we could even integrate this with an undo or require confirmation before saving, depending on preferences.

This workflow demonstrates how the agent effectively acted as an autonomous pair-programmer carrying out a multi-step refactoring: reading code, reasoning about improvements, and making edits. All these steps were achieved by combining basic tools in sequence, orchestrated by the agent’s logic.

**CI Toolchain Orchestration Example**

**Goal:** Automate running tests and fixing any failures (Continuous Integration loop).

* **Trigger:** Developer triggers a “Run CI Agent” for the project (perhaps on a commit or manually).
* **Agent Execution:** A ci\_agent (with access to tools like run\_command, read\_file, write\_file) starts a loop:
  1. It calls run\_command("pytest") (for example) to run the test suite. The output (including any failures) is captured.
  2. The agent parses the test output. If all tests passed, it streams a success message and finishes. If there are failures, it proceeds to fix them:
     + It may call a read\_file on the files where failures occurred or on the test file to understand the context.
     + It then asks the LLM something like: “Given this failing test output, suggest a code change to fix the issue.” The LLM might produce a patch or an explanation + code fix.
     + The agent applies the fix by calling write\_file with the changes (or possibly a specialized patch\_file tool if available to apply diffs).
  3. The agent re-runs run\_command("pytest") to see if the failure is resolved. This loop can continue for a few iterations if multiple issues are present.
  4. If after fixes all tests pass, the agent may even call a git\_commit tool (if one is exposed and user allowed) to commit the changes, or simply report success.
* **Feedback:** Throughout this, the IDE UI stream updates: test output is streamed (perhaps truncated or summarized by the agent), the agent might explain what it’s doing (“Test X failed, applying fix to module Y”), and it reports final status (“All tests passed. 2 issues fixed and changes saved.”).
* **Result:** Quick automation of the tedious fix/test cycle. Importantly, each code modification is done via the standard file write tool – so it can be tracked by source control and verified by the developer. The agent essentially serves as an AI-powered CI worker integrated into the IDE.

This showcases a multi-step **workflow agent** that involves decision-making and looping: the CI agent had to decide whether to continue or stop based on test results. Lastmile’s patterns like an Evaluator-Optimizer loop could be used here (the test results act as evaluation, the LLM acts as optimizer fixing code)[trendshift.io](https://trendshift.io/admin/repository/ask-ai/13216#:~:text=,test_input), or it can be scripted with plain logic. The mcp-agent framework allows either approach: we could implement the loop in Python around LLM calls, or craft an agent prompt that instructs the LLM to do the loop itself. In practice, a combination (letting Python handle external commands and iteration, and LLM handle code generation) works well.

**“Explain-then-Patch” Chain Example**

**Goal:** Provide an explanation for a piece of code or an error, then suggest a fix. This is useful for educational purposes or code review scenarios.

* **Trigger:** Developer highlights some code and asks “Why is this not working? Fix it.”
* **Agent Execution:** This could be a single agent or two agents working in sequence (one explains, another fixes). Using our system, we can implement it in one agent by prompt engineering:
  1. The agent (with read access) reads the relevant code (read\_file or a specific snippet provided).
  2. The LLM is prompted first to **explain**: “Explain what the following code does and why the bug X is happening.” The explanation is streamed to the user (this satisfies the “explain” part). We ensure to stream this before moving to patching. This can be done by yielding the explanation through the SSE channel.
  3. Then the agent (or the same prompt, continued) asks the LLM for a **patch**: “Now suggest a code change to fix the bug.” The LLM generates a fix (could be a diff or full code).
  4. The agent outputs the fix – perhaps both by displaying a diff to the user and by applying it via write\_file. Alternatively, the agent might wait for user approval after explanation before applying the patch (this could be an interactive pattern where the user’s go-ahead is needed – something feasible if using a WebSocket for two-way communication).
* **Feedback:** The explanation comes as a nicely formatted text, then the proposed patch as code. If the patch is applied, the IDE shows the code changes. If user approval was needed, the agent would pause (this requires an interactive mechanism – which could be done by designing the agent to return after explanation, and only run patch if user re-invokes it or confirms).
* **Result:** The developer learns *why* the issue happened and gets it fixed with minimal effort. This chain improves trust in the AI by showing rationale before performing an edit.

This example can be implemented by either a single composite agent prompt or by orchestrating two agent calls (one to an “explainer agent” and one to a “fixer agent”). Our architecture supports both: we could have an explain\_agent that returns an explanation, then invoke a patch\_agent, or simply have one agent with a multi-part instruction. Since the mcp-agent framework allows chaining of patterns, one could even conceive a meta-agent that first uses an *Augmented LLM* (for explanation) then a *Swarm or Router* to do the fixing[trendshift.io](https://trendshift.io/admin/repository/ask-ai/13216#:~:text=,test_input)[trendshift.io](https://trendshift.io/admin/repository/ask-ai/13216#:~:text=Agent%28name%3D,Load%20short_story.md) – but this might be over-engineering for this use case. A straightforward approach works well.

Each of these workflows is enabled by the same underlying pieces (tools + agent orchestration), just configured differently. The **modularity** of the system shines here: we can mix and match tools and agents to create new capabilities. For instance, if tomorrow we add a new tool for querying an online knowledge base, we can upgrade our explain agent to use it for more context, without changing the rest of the system.

**Real-Time Streaming and Frontend Integration**

Real-time feedback is crucial for a good developer experience. We have chosen **Server-Sent Events (SSE)** as the streaming transport between the backend and the IDE frontend for its simplicity and compatibility. Here’s how the integration works on the client side and some best practices:

* **Establishing SSE Connection:** When the IDE needs to invoke an agent or long-running tool, it opens an EventSource to the server’s SSE endpoint (e.g. http://localhost:8080/sse). This keeps a channel open for the server to push messages. If the protocol requires an initial request message, the frontend can send it via a POST request (for example, to /sse/messages as per MCP spec[blog.cloudflare.com](https://blog.cloudflare.com/ar-ar/streamable-http-mcp-servers-python/#:~:text=Initially%2C%20remote%20MCP%20communication%20between,client%20sends%20requests%20for%20tool)). Some implementations combine this (e.g. by sending a first event). The exact details can be managed by an MCP client library if available for JS, but implementing a minimal version is straightforward: send a fetch to trigger the action, then listen on EventSource for results.
* **Receiving Events:** The backend sends events as text chunks, which the EventSource API delivers to our JS callback. We might use event types like "message" or custom event names to distinguish stages. For example, the server can send event: status with data: "Running tests..." and later event: result with data: "Tests passed." The frontend can route these to appropriate UI components (like a log panel vs. a final result display).
* **WebSocket Proxy (Optional):** If bi-directional communication or more flexibility is needed (for instance, to allow the agent to prompt the user mid-way and wait for response), a WebSocket could be used. FastMCP doesn’t natively speak WebSocket, but we can easily stand up an ASGI app (via Uvicorn/Starlette) alongside the FastMCP server to proxy messages. One approach is to have the WebSocket route on the server side, and internally forward messages to the MCP tool handlers or agents, and vice versa. However, unless interactive back-and-forth is required, SSE suffices for one-way streaming of results. SSE has the benefit of simplicity (auto-reconnect, events are just text) and fits well with the stateless request/response nature of MCP.
* **Ensuring UI Responsiveness:** Because the agent tasks can be lengthy (several seconds or more), streaming keeps the user informed. Each tool invocation result or each chunk of LLM output should be sent as soon as available. For example, when using OpenAI’s streaming API, we can forward those tokens immediately to SSE. When running a shell command, we might stream its output live (by reading subprocess stdout line by line) rather than waiting for completion. This way, the frontend can display output progressively (similar to how a terminal shows logs in realtime).
* **Client-Side MCP Compatibility:** By adhering to the “standard MCP protocol,” we also make it possible for other clients or editors to plug in. For instance, an extension in VS Code or a different IDE that understands MCP could connect to our FastMCP server to use the same toolset. MCP is meant to be like a “universal USB for AI tools”[fleak.ai](https://fleak.ai/blog/mcp-intelligence#:~:text=MCP%20complements%20agent%20orchestration%20tools,are%20called%20and%20information%20exchanged), so our server could even be used by external AI agents (with proper auth). In the context of ALAN IDE, the Monaco/CodeMirror frontend essentially acts as the initiator of tasks and a renderer of results, leaving the heavy logic to the backend.

**Deployment and Dev Setup Instructions**

**Development Server Setup:** To run this integrated system locally:

1. **Ensure dependencies**: Python 3.11+, install fastmcp and mcp-agent (and any specific LLM SDK or API keys required – e.g., set OPENAI\_API\_KEY or ANTHROPIC\_API\_KEY in environment or in mcp\_agent.secrets.yaml as needed by Lastmile’s framework). The Lastmile agent will load API keys from environment or a config file for the LLM provider (the quickstart mentions a mcp\_agent.secrets.yaml to store keys, which you should prepare before running agents)[trendshift.io](https://trendshift.io/admin/repository/ask-ai/13216#:~:text=agent%20%60%60%60%20,agent%20that%20uses%20the%20fetch).
2. **Run the server**: Launch the combined script. If using uv:

bash

CopyEdit

uvicorn alan\_ide\_server:app # if we integrate with ASGI app, or

uv run alan\_ide\_server.py # if using uv (astral) to run the script

However, since we used mcp.run(...), that call will start the event loop and serve indefinitely. Simply running python alan\_ide\_server.py should start the FastMCP SSE server on port 8080 (or your chosen port).

1. **Open the IDE frontend**: If the frontend is a web application, open it in the browser. It should connect to http://localhost:8080 for MCP. In a development setup, you might serve the frontend from the same origin (to avoid CORS issues) or configure appropriate CORS on the MCP server. If the frontend is static, you can use a simple file server or integrate it into the Python backend (e.g., serve a directory with Starlette). The key is that the frontend knows how to reach the backend.
2. **Invoke a test workflow**: Try a simple operation to verify the chain. For example, in a console, you could simulate an MCP client call:
   * Use the mcp CLI (if installed via mcp[cli]) to list tools: mcp ls --host localhost --port 8080 should show read\_file, write\_file, agent\_refactor, etc.
   * Try calling a basic tool: mcp call read\_file '{"path": "README.md"}' to see it working.
   * Then simulate an agent call: mcp call agent\_refactor '{"file\_path": "example.py", "goal": "make it more Pythonic"}'. Because this triggers a multi-step process, you should see streamed responses. The CLI might display intermediate messages and the final result. This is essentially what the IDE UI would be doing behind the scenes when you trigger an agent.
3. **Logging and Debugging**: The mcp-agent framework provides a logger (accessible via mcp\_agent\_app.logger) that you can use to print out agent decisions, tool calls, etc., for debugging[trendshift.io](https://trendshift.io/admin/repository/ask-ai/13216#:~:text=mcp_agent,This%20agent%20can%20read%20the). During development, it’s useful to see the sequence of steps the agent is taking (e.g., which tool it decided to call, what the LLM’s thought process is). You might enable verbose logging for the agent when running the dev server.

**Production Considerations:** Since this is local-first and developer-focused, “deployment” is typically just running it locally. But if one wanted to expose this setup for remote collaboration or a centralized server, you’d want to:

* Secure the MCP endpoints (with authentication, since tools like file write are powerful).
* Potentially run the FastMCP server behind an HTTPS reverse proxy (for browser to connect securely).
* Scale the agent layer if multiple users connect (the framework can handle multiple sessions, but heavy parallel use might require more resources or separate processes per user).

For a single developer using ALAN IDE locally, performance should be fine – FastMCP is lightweight, and tool calls are just local function calls. The heavy lifting is in the LLM API calls, which are network-bound by the AI service used (OpenAI/Anthropic). Those can be optimized by batching requests if needed or using a local model for faster iteration if available.

**Conclusion**

We have designed a comprehensive system where the **FastMCP tool server** and **Lastmile’s mcp-agent framework** work in concert to provide an AI-augmented development environment. The FastMCP layer turns the IDE’s capabilities into a standardized API that any AI model can interface with (the “plumbing” of tool use)[fleak.ai](https://fleak.ai/blog/mcp-intelligence#:~:text=scalable). The mcp-agent layer adds the intelligent orchestration – implementing proven agent patterns (reactive tools usage, workflows, even multi-agent coordination) on top of those tools[github.com](https://github.com/lastmile-ai/mcp-agent#:~:text=1.%20Model%20Context%20Protocol%20,ready%20AI%20agents)[github.com](https://github.com/lastmile-ai/mcp-agent#:~:text=2,agnostic%20way). By keeping these layers modular, ALAN IDE’s agent system is highly extensible and maintainable:

* **Modularity:** New tools (IDE features) can be added without touching agent logic. New agents (AI behaviors) can be introduced without altering the core server.
* **Real-Time Interaction:** SSE streaming ensures the developer is kept in the loop, seeing what the AI is doing step-by-step, which builds trust and allows intervention if needed.
* **Standards-Compliant:** Using the MCP protocol means our solution aligns with an emerging standard for AI tool use, making it future-proof and interoperable. *MCP is not an orchestration engine by itself, but an integration layer*[*fleak.ai*](https://fleak.ai/blog/mcp-intelligence#:~:text=scalable) *– by pairing it with mcp-agent (the orchestration), we get the best of both.*
* **Local-First:** The entire setup runs with a single command on a developer’s machine (just pip install and run), with minimal configuration. This caters to privacy and speed, as code never leaves the local environment except when hitting the LLM API.

In essence, we built an **AI co-developer inside the IDE**. Developers can edit, run, and debug code as usual, and call upon AI agents for complex tasks – whether it’s refactoring legacy code, running tests and applying fixes, or explaining code and issues. The agents leverage the same tools a human would (file reads, writes, command execution), but at machine speed and with the insight of powerful LLMs. By following this guide, you can implement the architecture step-by-step, adjust it to your specific needs (different tools or agent behaviors), and have a robust system powering ALAN IDE’s intelligent features. Enjoy your new AI-augmented workflow!