ChatGPT

AI Agent System for Automated MCP Tool Generation and Integration

Introduction and Objectives

Building an AI-driven agent system for automated **Model Context Protocol (MCP)** tool generation requires a robust design that can analyze application specifications, create functional tools, test them, and deploy to a Next.js-based platform. MCP is an emerging open protocol for extending AI models with tools and knowledge, allowing external servers to provide new data sources and capabilities to AI agents [1]. In our context, *MCP tools* are essentially functions exposed by a Python MCP server that perform actions like calling APIs, retrieving data, or processing inputs [2]. These tools are discoverable by AI clients (e.g. an LLM-based agent can list available tool names, descriptions, and parameters) and can be invoked to perform tasks [3].

The goal is to design a **multi-agent system** that takes in application documentation (e.g. API reference, resource specs) *without direct code access*, and automatically produces **production-ready MCP tools** that integrate with the platform. The system must:

- **Analyze Application Specs:** Parse API docs, resources, or other documentation to understand what functionalities (endpoints, operations) are available and needed (without reading the app's source code).
- **Determine Required Tools:** Decide which MCP tools should be created to expose these capabilities to the AI (e.g. one tool per relevant API endpoint or use-case).
- **Generate Tool Implementations:** Write syntactically correct Python code for each tool, conforming to MCP server standards (using decorators, correct function signatures, return formats, etc.).
- **Automated Testing & Validation:** Run the new tools (calling the actual APIs if possible) to verify they work as intended and match the documentation's contract. This includes checking output formats, error handling, and compliance with specs.
- **Human-in-the-Loop Checkpoints:** Include points where a human developer or reviewer can inspect and approve (or adjust) the generated tools, especially before deployment or any high-stakes action.
- **Deploy & Integrate Tools:** Once approved, incorporate the new tools into the MCP server and the Next.js platform so that the AI agent (and the platform's users) can utilize them in production.

The output of this research and design will focus on system architecture, process flow, and integration strategy rather than code. We will draw on current best practices in multi-agent orchestration, automated code generation and testing, and MCP tool development. Authoritative sources — including official MCP documentation, research on multi-agent code generation, and real-world case studies — inform this design.

System Architecture Overview

At a high level, the system will use a **modular multi-agent architecture** to break down the complex task into specialized roles. This is analogous to a microservices approach: instead of one monolithic AI trying to do everything (which can lead to errors and "jack of all trades, master of none" issues [4]), we delegate sub-tasks to expert agents. Each agent operates semi-independently on its portion of the workflow, passing results to the next agent in a pipeline. This specialization and pipelining improve reliability and maintainability [5].

The core agents and their roles are:

- **Analyzer Agent:** Reads and interprets application documentation and specifications. It identifies what MCP tools are needed (e.g. which API endpoints or resources should be wrapped as tools) and extracts the details required for implementation (endpoint URLs, parameters, expected responses, authentication requirements, etc.). It outputs a structured plan or list of tool definitions.
- **Generator Agent:** Takes the analyzer's plan and writes the actual Python code for each tool. It ensures each function follows the MCP server's format (using the `@mcp.tool()` decorator, proper async signatures, type hints, and docstrings for discovery [3]). This agent will leverage an LLM skilled in code generation to produce correct, well-documented code for the tools.
- **Tester Agent:** Automatically tests the generated tools. This involves invoking each tool (which in turn calls the external API or performs the action) with sample inputs to verify that it executes without errors and the outputs align with documentation or expected results. The tester agent may design basic test cases (including edge cases) and check response validity (e.g. status codes, JSON schema, etc.).
- **Validator Agent:** Performs additional validation on the tools and their outputs. This includes static analysis (ensuring code syntax is correct and style guidelines met), compliance checks against the API specification (do the tool's inputs/outputs match the spec?), and evaluating whether all required tools were properly generated. The validator collates results from testing and its own checks to determine if the tools meet the acceptance criteria or if refinements are needed.

These agents will communicate and work in sequence as part of an **end-to-end workflow pipeline** (with potential iterations). A central Orchestrator coordinates the process, triggering each agent in turn and aggregating their outputs. The architecture emphasizes persistent context sharing – agents don't work in isolation but rather build on each other's results using a shared state or memory. For example, the Analyzer's output (list of tool requirements) becomes input context for the Generator; the Generator's code output is passed to the Tester; test results feed into the Validator and possibly back to the Generator for fixes. By maintaining a common context, the system ensures continuity and avoids losing information between steps [6].

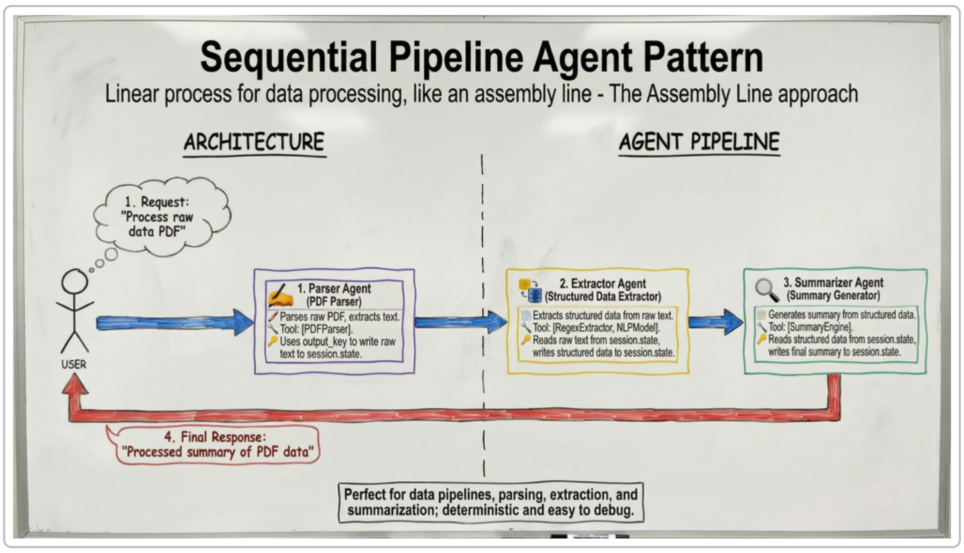

Sequential Pipeline Agent Pattern
Linear process for data processing, like an assembly line - The Assembly Line approach
ARCHITECTURE
AGENT PIPELINE
1. Request: "Process raw data PDF"
USER
1. Parser Agent (PDF Parser)
Parses raw PDF, extracts text.
Tool: [PDFParser].
Uses output_key to write raw text to session.state.
2. Extractor Agent (Structured Data Extractor)
Extracts structured data from raw text.
Tool: [RegexExtractor, NLPModel].
Reads raw text from session.state, writes structured data to session.state.
3. Summarizer Agent (Summary Generator)
Generates summary from structured data.
Tool: [SummaryEngine].
Reads structured data from session.state, writes final summary to session.state.
4. Final Response: "Processed summary of PDF data"
Perfect for data pipelines, parsing, extraction, and summarization; deterministic and easy to debug.

Figure: Example of a sequential multi-agent pipeline (assembly line pattern). Each agent performs a specialized step and passes its output via shared state to the next agent [7]. In our design, a similar sequence (Analyze → Generate → Test → Validate) ensures a deterministic and debuggable flow.

Inter-agent communication can be handled through structured messages or a shared memory store. For instance, the system might use an internal data structure or file (like a "planning document") that all agents read and write to, as was done in one successful multi-agent setup where agents communicated through a shared markdown plan [8] [9]. In our case, the Analyzer could write out a "Tool Specifications" list that the Generator reads from, and the Tester could append results to a "Testing Report" that the Validator and human reviewers examine. The Google ADK design patterns similarly encourage using a shared `session.state` object where each agent's output is stored under a key for the next agent to pick up [6]. This approach provides traceability and persistent context across the pipeline.

Agent Roles and Responsibilities

Analyzer Agent (Application Understanding)

Role: The Analyzer is essentially the system architect or planner. Its job is to understand the application's capabilities from provided specifications or documentation and determine what MCP tools should be created. This agent parses inputs such as OpenAPI/Swagger files, REST API docs, SDK references, or other technical docs. It extracts key information about each API endpoint or resource: the endpoint URLs, HTTP methods, query/path parameters, request/response schemas, authentication requirements, rate limits, etc. It then decides which endpoints are relevant to expose as tools and formulates a definition for each tool.

Techniques: The Analyzer may use a combination of deterministic parsing and LLM-driven summarization. If a formal spec like an OpenAPI JSON/YAML is provided, the agent can parse it programmatically to get a list of endpoints and schemas (this is reliable and avoids hallucination). For unstructured documentation, an LLM could be prompted to identify all the API operations and their details – for example, by splitting the docs into sections per endpoint and extracting fields (name, purpose, inputs, outputs). This could be done by a sub-step where the Analyzer first acts as a *Parser* (converting docs into text) and an *Extractor* (structuring the data), akin to ADK's example of parsing a

PDF then extracting structured data [7]. The output from the Analyzer is a structured list of tool specs, where each spec might include:

- Tool name (e.g. `getUserDetails` corresponding to a "GET /users/{id}" endpoint).
- Description (a human-readable summary from the docs, which will go into the tool's docstring for discoverability).
- Input parameters (names and types, e.g. `user_id: str` or complex objects if needed).
- Output expectation (e.g. returns JSON data as a Python dict or perhaps a formatted string; the agent notes if the API returns JSON, we might return it as structured text).
- Any special handling (e.g. this endpoint requires an API key or a prior auth token, or has pagination parameters, etc.).
- Test example (if the documentation provides an example request/response, the Analyzer can capture that as a basis for testing later).

The Analyzer prioritizes which tools to generate if not all are needed. For instance, if an API has 100 endpoints but the platform only needs a subset (perhaps based on user scenarios), the agent ranks importance or filters by keywords. In general, however, it's safer to generate wrappers for all documented endpoints or at least all "public" operations, since the cost is mainly in development time (which is automated here) and not runtime cost. The output is essentially a *blueprint* for the Generator.

Example: Suppose the docs describe a **Weather API** with endpoints for getting current weather, forecast, and alerts. The Analyzer might produce:

- Tool: `get_current_weather(city)` – calls GET `/weather/current?city={city}`.
- Tool: `get_forecast(city, days)` – calls GET `/weather/forecast?city={city}&days={n}`.
- Tool: `get_alerts(area)` – calls GET `/alerts?area={code}`.

Each with a description and noted parameters. It would also note if an API key is needed in headers, etc., so that the Generator knows to include that. This breakdown is presented to the next stage, and optionally to a human for review if desired (a checkpoint could be placed here to confirm the plan before coding).

Generator Agent (Tool Code Generation)

Role: The Generator is responsible for transforming the plan from the Analyzer into actual code – specifically, Python functions that implement each tool's logic. It plays the role of the software developer or "builder". Each tool will be implemented as an asynchronous Python function decorated with `@mcp.tool()` (from the MCP server framework) to register it. The generator must ensure the function signature (name and parameters) matches what was planned, and that the code correctly calls the relevant API or performs the described action.

MCP Tool Format: MCP tools have specific structural requirements. Using the FastMCP framework (a Pythonic way to build MCP servers), defining a tool is as simple as decorating a function with `@mcp.tool()` and writing an async function body [10]. The function's name and docstring become the tool's name and description that clients see, and its parameters (with type hints) define what inputs the tool accepts [3]. The Generator should follow these conventions strictly so that the resulting tools are self-documenting and discoverable by clients. For instance:

```
@mcp.tool()
async def get_forecast(city: str, days: int=3) -> str:
    """Get the weather forecast for a city for a number of days."""
    # ... call external API and return result ...
```

This would create a tool “get_forecast” that clients can list and see it requires a `city` and `days` parameter, with the provided description.

Code Synthesis Process: Under the hood, the Generator agent likely uses a powerful code-generating LLM (such as GPT-4 or similar) with a prompt that provides the tool spec and possibly example code patterns. It can be prompted in a chain-of-thought style to first outline the solution (e.g., figure out how to form the HTTP request for the API, what error handling to do) and then produce the code. We want the code to be *functional and syntactically correct* on the first attempt as much as possible. The agent will incorporate details from documentation such as endpoint URLs, required headers or auth tokens, expected JSON fields in the response, etc.

To ensure correctness, the Generator might employ a few strategies:

- **Few-Shot Examples:** Provide the LLM with one or two examples of simple MCP tool functions (perhaps from known templates or previous successful generations) so it understands the format and style expected. E.g., an example of a simple `@mcp.tool` function that calls a REST endpoint and returns formatted text could be given in the prompt.
- **Validation of Syntax:** After code generation, the system can do a quick syntax check (for example, attempt to parse the Python code or run a linting tool). If syntax errors are detected, the Generator agent (or a subordinate process) can either automatically correct them or prompt the LLM to fix issues. This is an automated guardrail to catch obvious mistakes before testing.
- **Conformance to MCP Standards:** The Generator ensures return values and content follow MCP conventions. According to OpenAI’s guide for MCP servers, tool results should be returned as a **content array** of one or more content items, each with a type (e.g. "text") and payload [11] [12]. In many cases, returning a plain string or Python dict might be automatically wrapped by the MCP framework, but the safest approach is to follow the expected output schema. For example, if an API returns JSON, the tool might return it as a JSON string within a content item. These rules can be encoded in the prompts or enforced by post-processing. (For initial implementation, returning simple text or Python objects is acceptable, as FastMCP will package it appropriately, but being aware of content types will be important for advanced features like multimodal responses [13] [14].)

Helper Functions & Reusability: The Generator may also create helper utilities if needed. For example, if multiple tools require authentication or hitting the same base URL, it could generate a small helper (within the same module) for sending requests (similar to how the weather example had a `make_nws_request` function to handle HTTPX calls [15]). However, caution is needed: since we lack the full application context, the agent shouldn’t create overly complex shared logic unless confident. An MVP approach is to let each tool be self-contained (possibly with duplicate code for simplicity) and refine later to factor out common code if necessary.

Output: The result from the Generator agent is the code for the new tools, likely as one or more Python files or strings. This code can either be directly written into the MCP server project or held in memory for testing first. At this stage, we have candidate implementations but not yet proven to work — that’s where testing comes in.

Tester Agent (Automated Tool Testing)

Role: The Tester agent acts as a QA engineer, automatically exercising the generated tools to validate their behavior. The key challenge is **testing without direct access to the application's internals**. We treat each tool as a black-box function that should adhere to the API's contract as per documentation. The Tester will call each tool function (likely by spinning up the MCP server in a test mode or by invoking the function directly if possible) with test inputs and then observe the outputs or any errors.

Testing Workflow: For each tool, the Tester will generate one or more test cases. These may include:

- **Basic Positive Case:** A typical valid input scenario (using example values from documentation if available). For example, if the tool is `get_user_details(user_id)`, the tester might call it with a sample user_id (perhaps from docs or a dummy ID) and expect a result. If the API is live and accessible, the agent will actually perform the HTTP call through the tool and get a real response. The tester then checks if the response is well-formed (e.g., contains expected fields or messages).
- **Edge or Error Case:** If applicable, test how the tool handles invalid or boundary inputs. For instance, pass an invalid ID or a missing required parameter (if the tool function signature allows optional parameters) to see if it returns a graceful error message or raises an exception. The docs might state how errors are returned (e.g. a 404 with a JSON error object). The tester can simulate that and verify the tool's behavior (e.g., does it catch a 404 and return "User not found" message as perhaps coded, or does it propagate an exception?). This helps ensure robustness.
- **Schema Validation:** If an OpenAPI spec is available, the tester can validate the structure of the response against the schema. There are tools like Dredd that take an OpenAPI spec and a running API, send requests and check that responses match the spec (status codes, JSON fields, types, etc.) [16] [17]. We can leverage a similar concept: after calling the tool, we compare the result to the expected response schema in the documentation. For example, if the spec says the response JSON should have field "price" as a number, and our tool returns `"price": "19.99"` as a string, that's a discrepancy. The Tester would flag such mismatches. This is essentially **contract testing**, treating the docs as the contract. It ensures our tool implementation didn't mis-handle or transform data incorrectly.

Execution Environment: To execute tests, we need the MCP server running with the new tools. One approach is to use an in-memory test harness. The FastMCP library supports connecting a client directly to a FastMCP instance in-memory for testing, avoiding network calls [18] [19]. We can instantiate the MCP server (with our generated tools loaded) in a test mode and then call each tool via the MCP client API. This simulates how an AI agent would call it. Alternatively, since the tools are just Python functions, we could call them directly in code for simplicity (passing a dummy `Context` if required by signature, or adjusting if needed). The in-memory approach is cleaner and closer to real usage: for example, using `client.call_tool("tool_name", {params})` and capturing the result [20].

Output & Feedback: The Tester agent will produce a report of test outcomes. For each tool, it notes whether tests passed or failed. A "pass" might mean the tool ran without exceptions and returned data that appears correct (e.g., an HTTP 200 and a JSON with expected keys). A "failure" could be: runtime error (exception thrown), incorrect output format (did not conform to spec or expectations), or logically incorrect data (perhaps the tool parsed something wrongly). The Tester should log as much detail as possible for failures – e.g., stack traces or error messages, differences between expected vs actual output, etc. This detailed feedback is crucial for the next step, as it will guide the Generator (or human) in fixing issues.

Importantly, the Tester agent's routine should also watch out for external factors: - If the external API has **rate limits** or usage quotas, tests should be rate-aware. The agent can include delays between calls or use a sandbox environment if available. If hitting a rate limit (HTTP 429), it should note that and possibly retry after a wait, or at least inform that too many rapid calls might not be possible (which is an edge case to handle in production usage too).
- If the API requires **authentication**, the tester needs valid credentials or tokens. In a secure design, we wouldn't hard-code keys in generated code; instead, the tools might expect an environment variable or use a configured API key. For testing, the agent framework should inject any necessary keys. If no credentials are provided, the tester might skip tests that require auth or use mock responses if possible. This is something to plan in deployment: perhaps allow a way to supply a test API key to the system for the target application.
- For **non-deterministic data** (e.g., an API returns changing data like stock prices), writing a strict assertion is tricky. The tester may then only verify structural aspects (correct fields and types) rather than exact values.

Validator Agent (Verification and Refinement)

Role: The Validator agent serves as an extra layer of assurance, combining the outputs of testing with its own analysis to decide if the generated tools are acceptable. It acts as a code reviewer and quality control. The Validator performs both **static validation** (code review, spec conformance checks) and **dynamic validation** (reviewing test results provided by the Tester). It determines if all conditions are met for deployment or if further iteration is needed.

Static Code Review: The Validator will inspect the generated code without running it. Potential checks include:

- **Correct Syntax and Style:** Ensure no syntax errors (which should already be caught earlier) and that the code is idiomatic. Possibly run a linter or formatter check. While style issues don't impede function, maintaining a standard (pep8, naming conventions consistent with existing codebase, etc.) is good for long-term maintainability.
- **MCP Compliance:** Verify that each function is properly decorated and has docstrings. Ensure the tool naming is clear and descriptive. Confirm that the return types are handled as per MCP expectations (e.g., functions returning complex objects should be returning them in a way that the MCP framework will encapsulate in content items). The OpenAI connector guide explicitly states that tool definitions must conform to the expected shape for them to work in ChatGPT or similar contexts [21] . The Validator ensures no tool deviates from those standards (for example, a tool must return something serializable; returning a database cursor object, hypothetically, would be invalid).
- **Completeness:** Check that all major capabilities from the spec have corresponding tools. If the Analyzer missed something critical (for instance, an important endpoint), the Validator could flag that as a gap. This could be done by cross-referencing the documentation: e.g., if an OpenAPI spec has 10 GET operations and only 8 tools were generated, the Validator asks why (it could be intentional filtering, or an oversight).
- **Security & Error Handling:** Ensure that no sensitive info is hard-coded (like API keys should not be directly embedded; better to pull from environment variables or context). Check that the tools handle errors gracefully (the code should catch HTTP errors and return a user-friendly message rather than raw tracebacks). Also verify that any potentially dangerous operations are sandboxed or safe (for example, if a tool writes to a file, that might be outside scope – likely our tools mainly call external APIs though).
- **Documentation and Clarity:** The Validator can ensure each tool's docstring is populated (possibly using the info from the spec) and inputs are explained. This is important because when

a client (like an AI agent or developer) lists available tools, they will see the description. A clear description helps correct usage. The Analyzer/Generator should have done this, but the Validator double-checks (e.g., docstring not empty or too terse).

Review of Test Results: The Validator agent examines the output from the Tester. For any failed tests, it categorizes the issues and decides if they can be fixed automatically or need human attention. Example issues and actions:

- If a tool returned an unexpected schema (say missing a field), the Validator might deduce the fix (perhaps the code didn't extract that field from the API response). It can then pass a targeted feedback to the Generator agent: e.g. *"Tool X: response is missing field 'Y'. Modify the code to include that from the API response JSON."*
- If there was a runtime exception (like a KeyError due to an unexpected response structure), the Validator notes it and could attempt to trace it to a cause (maybe the API returned an error message the code didn't handle).
- If everything passed, the Validator will still do a final check if the outputs align with the spec qualitatively. It might, for instance, take the actual output from a test call and compare it with an example in the documentation if available, ensuring they align (or at least that the output isn't nonsensical).

Iterative Feedback Loop: Crucially, if any issues are found, the Validator triggers a feedback loop. This means the system does not immediately fail or ask for human help for every issue – instead, it tries *AI self-correction*. The Validator can formulate a set of instructions or error descriptions and feed it back to the Generator agent for another iteration of code refinement. This approach has precedent in research: multi-agent code generation systems like AgentCoder show that having a test executor agent feed errors back to the code generator (programmer agent) for refinement dramatically improves correctness [22]. In our design, the Tester/Validator combination serves a similar purpose: they provide *ground truth feedback* so the Generator (programmer) can fix its mistakes. By cycling through generation → test → validation multiple times if needed, the system converges on a working solution. This is akin to how a human developer might write code, run tests, and debug in a loop.

To keep the loop efficient, we might limit the number of iterations (for example, allow up to 2 or 3 refinement cycles automatically). If after that the tools still aren't passing tests, it's a signal that either the documentation is insufficient or the task is too complex, so human intervention is warranted at that point.

When the Validator is finally satisfied (all tests pass and code quality checks out), it will mark the toolset as *ready for human review*. This triggers the human-in-the-loop phase.

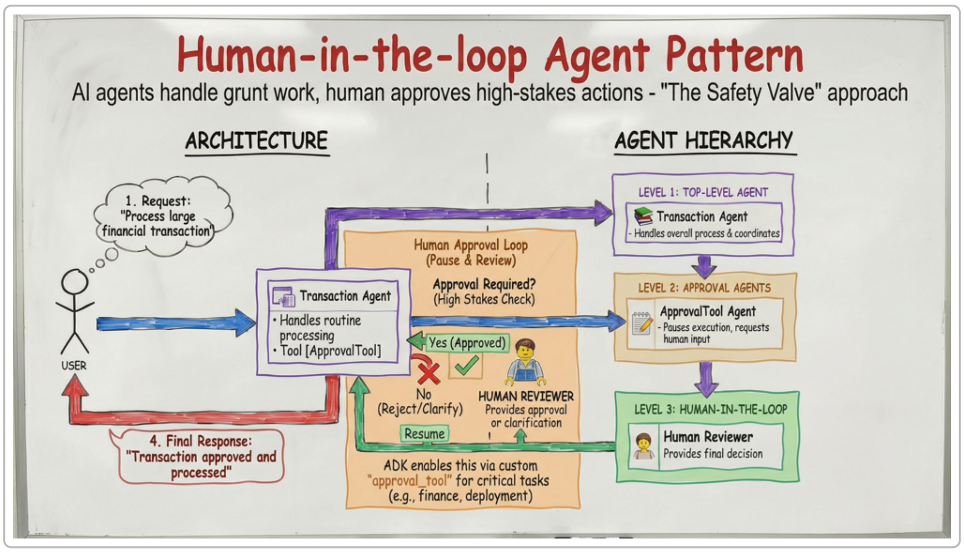
Human-in-the-Loop Integration

Even with advanced automation, we incorporate **human oversight** as a safety net for critical decisions and deployments. Human review is essential because AI-generated code, especially one interfacing with external systems, can have subtle issues or unintended consequences that an automated test might not catch (for instance, a misunderstanding of a business logic or an edge case not documented). In our pipeline, the human-in-the-loop acts as a final gatekeeper before tools go live, aligning with the "human approval for high-stakes actions" pattern [23]. Deploying code to production or giving an AI new capabilities to act can be considered high-stakes, so a human should approve it just as one would approve a code change via code review in a conventional development cycle.

Checkpoints for Human Review: There are two potential checkpoints for human intervention:

1. **After Analysis (Optional):** Once the Analyzer agent produces the list of proposed tools and their specifications, a human reviewer (perhaps a lead engineer or the API owner) could review this plan. This early check ensures the system is about to build the right things. For example, the human might spot that an endpoint is internal or deprecated and should not be exposed, or notice that the analyzer missed a crucial parameter. If any corrections are needed, the human can adjust the specs or provide clarifications. The plan can then be fed into generation. This checkpoint is optional and could be bypassed for speed if confidence in the analyzer is high, but it's useful for complex or critical apps.

2. **Pre-Deployment (Mandatory):** After the Validator agent signals that tools are successfully generated and tested, a human should review the final artifacts. At this stage, the human can inspect the generated code and documentation for sanity. They might run a quick manual test themselves or read through the code to ensure no obviously harmful or inefficient code is present. Because the system has already validated functionality, the human can focus on qualitative aspects: is the code clear? Are there any potential side effects? Does it align with the expected use of the API? This is the final QA. The human can either approve deployment or send it back for another iteration (with comments). In practice, we can implement this as an "approval required" step: the system might pause and present the results via the Next.js frontend (e.g., show a diff of changes or a report) and wait for a button click or confirmation from the reviewer [24]. Only on approval does it proceed to integrate the new tools.



Human-in-the-loop Agent Pattern
AI agents handle grunt work, human approves high-stakes actions - "The Safety Valve" approach
ARCHITECTURE
AGENT HIERARCHY
1. Request: "Process large financial transaction"
USER
Transaction Agent
• Handles routine processing
• Tool [ApprovalTool]
Human Approval Loop (Pause & Review)
Approval Required? (High Stakes Check)
Yes (Approved)
No (Reject/Clarify)
HUMAN REVIEWER Provides approval or clarification
Resume
ADK enables this via custom "approval_tool" for critical tasks (e.g., finance, deployment)
4. Final Response: "Transaction approved and processed"
LEVEL 1: TOP-LEVEL AGENT
Transaction Agent
- Handles overall process & coordinates
LEVEL 2: APPROVAL AGENTS
ApprovalTool Agent
- Pauses execution, requests human input
LEVEL 3: HUMAN-IN-THE-LOOP
Human Reviewer
Provides final decision

Figure: Human-in-the-loop pattern, where agents handle groundwork and a human reviewer must approve high-stakes actions before proceeding [23]. In our case, the "Transaction Agent" is analogous to the Generator/Validator producing new code, and the "ApprovalTool Agent" represents the process pausing for human confirmation.

In terms of implementation, one way to realize the pause is via a custom "approval tool" or a gating function. For example, the Validator agent (or orchestrator) could call a special MCP tool or trigger that essentially creates a review task for the human – similar to ADK's approach of an agent calling an `ApprovalTool` that waits for human input [24]. The Next.js front-end could show the list of tools and their statuses, and the human can click "Approve" or "Reject with feedback". If rejected, the feedback goes back to the relevant agent or simply ends the automation for manual handling.

End-to-End Workflow Pipeline

Bringing all the components together, we can outline the full lifecycle from input to deployment. Below is the sequential workflow, with iterative loops and checkpoints integrated:

1. **Specification Input:** A user or developer provides the application spec/documentation to the system. This could be uploading an OpenAPI file, pasting documentation text, or pointing the system to a URL. The Next.js front-end might offer an interface for this. The input is stored and possibly pre-processed (e.g., if it's a PDF or HTML, convert to text).

2. **Analysis Phase (Analyzer Agent):** The orchestrator invokes the Analyzer agent with the documentation input. The Analyzer parses and extracts the required information, then produces a structured **Tool Specification List**. For each proposed tool, the spec includes name, description, parameters (with types), and the intended API call or action. The entire analysis output is saved to shared context (e.g., a JSON structure or a markdown table for human readability).

3. *Checkpoint:* Optionally, present the list of tools to a human for review/confirmation. If the human modifies or comments on the plan, update the spec list accordingly before proceeding.

4. **Generation Phase (Generator Agent):** The Generator agent iterates over the Tool spec list and generates code for each tool. It creates a Python module (or multiple modules if needed) containing all the new tool functions, possibly including any helper functions and necessary imports. The code is kept in memory or a staging area (not yet merged into the main server codebase). After generation, a quick syntax validation is done (ensuring the code can run).

5. **Initial Validation:** The system can attempt to *load* the generated code into a test MCP server instance to ensure it registers the tools properly. Using something like `FastMCP.from_openapi()` isn't needed here since we already generated code, but we could use FastMCP's in-memory client to list the tools and verify they show up with correct names and descriptions. For example, call `client.list_tools()` and ensure all expected tool names are present, which confirms the `@mcp.tool` decorators worked and the server started without runtime errors. This step catches errors like missing imports or decorator misuse early.

6. **Testing Phase (Tester Agent):** The Tester agent runs through the list of tools, executing tests as described earlier. It might do this sequentially, resetting any state between calls (since each tool call might be independent). The tester logs results for each tool: e.g.

7. `get_forecast`: Passed basic test (returned 200 and data structure matches spec), Passed edge test (invalid city returned graceful error).

8. `get_alerts`: Failed (threw exception on valid input, perhaps due to JSON key error). And so on. These results are stored in a **Test Report**.

9. **Validation & Feedback Phase (Validator Agent):** The Validator agent reads the Test Report and also reviews the code/spec. It compiles a list of any issues:

10. For each failing tool, it analyzes why. Suppose `get_alerts` failed because the external API's response JSON had a key `alerts` but our code expected `features` (documentation mismatch). The Validator notes the discrepancy.

11. It checks code quality: maybe it finds that `get_current_weather` tool is missing a docstring or the code didn't handle an empty response case that the docs mention. These are added to issues.
12. It verifies all intended tools were generated.
13. If issues are found, the Validator forms a feedback message for the Generator agent. This could be done tool-by-tool: e.g. *"Revise get_alerts: use key 'alerts' instead of 'features' as per API response. Also add null-check if no alerts."*
14. The orchestrator then initiates a refinement iteration: it feeds this feedback into the Generator agent (potentially with the original prompt context) to regenerate or patch the code for the affected tools. This is effectively a new round of generation, focused on fixing rather than creating from scratch.
15. The updated code replaces the old code for those tools. Then the cycle goes back to Testing phase for those tools (regression testing the others too if needed).

This loop continues until the Validator finds no more issues or until a preset iteration limit is reached. Each cycle should converge closer to correct tools. Research supports that such an iterative approach can significantly boost accuracy – for instance, AgentCoder achieved **96%+ pass rates** on coding tasks by iterating code generation with test feedback, far outperforming single-pass approaches [25]. We anticipate similar benefits in our domain of API tool generation, as the iterative loop will catch and correct many mistakes.

1. **Final Review (Human Approval):** Once the Validator agent is satisfied (all tests pass, and code quality is good), the system pauses for human approval. The Next.js interface will show the summary of the new tools: their names, descriptions, code, and test results (e.g., all green checkmarks). The human reviewer can browse the code (read-only or maybe with an option to tweak minor things if needed) and the logs of test runs. If everything looks good, the human authorizes deployment. If not, the human can either request another iteration with specific feedback (which the Generator agent can attempt to implement) or choose to manually edit the code.

This step ensures accountability and safety. As noted in Google's multi-agent patterns, having a human in the driver's seat for irreversible actions (like deploying code) is a wise safeguard [23]. Our system effectively enforces that – nothing goes live until a human clicks "Approve". This is akin to a code review + deployment approval in one.

1. **Deployment Phase:** Upon approval, the system integrates the new tools into the production environment. This could be automated by merging the code into the MCP server codebase and redeploying the MCP service. For example, if the MCP server is containerized, the system might trigger a rebuild/deploy of the container with the updated code. If using a dynamic server approach, it might hot-load the new tool module into a running server (though typically, restarting the MCP server is straightforward and acceptable).

The Next.js platform will then have access to these tools. Depending on the platform design, integration might involve: - Updating a configuration so the front-end knows these new capabilities exist (e.g., if the UI needs to present new options to users or if certain workflows now become possible via the AI agent). - Ensuring the AI agent (if it's part of the platform's back-end) is aware of the tools. If the AI agent uses the MCP protocol correctly, it can dynamically discover the new tools at runtime by calling `list_tools` on the MCP server [26]. So the main integration is simply deploying the MCP server with the new endpoints; the agent will then "see" them and can start using them in response to user queries. For instance, if a user asks for weather info, the AI can now invoke the `get_forecast` tool that was just created. - We should also log this deployment event and potentially monitor initial usage to catch any runtime issues not found in testing.

1. **Post-Deployment Monitoring (Ongoing):** Though outside the immediate scope of generation, it's worth noting that once deployed, the system should monitor the performance of these tools. If exceptions occur in production (maybe a corner case was missed), it can feed that back as a new iteration (perhaps in a future update cycle). Logging and analytics around tool calls could be part of the platform to continuously improve reliability.

In summary, the pipeline resembles an **assembly line** with quality control loops. It begins with an idea (the documentation) and ends with a tangible product (working MCP tools integrated into the system). The use of multiple specialized agents makes the process structured and transparent, as opposed to a single giant black-box prompt. This modular design also means we can improve or replace individual components (say, use a better parsing algorithm, or a stronger code model) without overhauling the entire system.

Key Design Considerations and Techniques

Multi-Agent Collaboration and Persistence

The choice of a multi-agent system is motivated by the need for specialization and reliability. Studies have shown that dividing tasks among specialized agents can yield better outcomes than a single agent doing all. For example, MetaGPT and ChatDev introduced multiple agents for software development, each role mimicking a team member (architect, developer, tester, etc.), improving the structure of the process [27]. However, those systems sometimes went overboard with too many agents (5–7 agents), incurring heavy communication overhead [28]. Our design keeps the team lean (four core agents), focusing on the critical skills needed. This mirrors the approach of **AgentCoder**, a recent code generation framework that uses only three agents (programmer, test designer, test executor) and found that this minimal collaboration actually *improved* performance while drastically reducing token usage compared to frameworks with more agents [29] [30]. In our case, the Analyzer and Generator are analogous to AgentCoder's "programmer" (one focusing on planning, one on coding), and the Tester and Validator together play the role of test generation/execution and feedback. By keeping agents modular but not too many, we strike a balance between specialization and efficiency [31].

Inter-agent communication is implemented through the orchestrator and shared context. Rather than free-form chat between agents (which can get chaotic), the orchestrator enforces a sequence and provides each agent with the relevant context slice. This is effectively a **sequential pipeline pattern**, which is deterministic and easier to debug [32]. The state (memory) persists across the pipeline via structured data (like the spec list and test reports). Agents may also log their intermediate reasoning or decisions into the context for transparency. For example, the Analyzer could record not just *what* tools it chose, but *why* (e.g., "Skipping endpoint /admin/* as it's not user-facing"). These notes could help the Generator or aid a human reviewer. The persistent state acts as the single source of truth that grows as we move along the pipeline.

One interesting aspect is *tool use by agents*. We are building MCP tools, but the agents themselves could use tools to help them do their job. For instance, the Analyzer agent might utilize an **OpenAPI parsing tool** if one exists, or an OCR tool if the documentation is an image. The Tester might use a **JSON schema validator** tool to compare outputs to spec. If we integrate such utilities, we essentially have an agent using one tool to help create another tool – a meta aspect of the design. This is supported by frameworks like LangChain or ADK, where an agent can be equipped with tools for its internal process (e.g., a Regex extraction tool for parsing) [7]. Our architecture could allow the inclusion of such helper tools to the agents, though initial focus will be on the main logic.

Documentation Parsing and Requirement Extraction

Accurate parsing of the application documentation is foundational. We consider different documentation formats: formal (like OpenAPI, Postman collections) vs. informal (like developer guide text, wiki pages).

- **Formal Specs (OpenAPI/Swagger):** These are machine-readable and can be parsed into data structures easily. If the user provides an OpenAPI spec, our job is greatly simplified. We can directly extract endpoints, operations, parameters and schemas. In fact, the existence of FastMCP's `FastMCP.from_openapi()` method shows that automating MCP tool generation from OpenAPI is feasible and has been implemented to some degree [33]. That function likely generates a skeleton MCP server that forwards calls to the actual API according to the spec. We can leverage such functionality either by using it as a baseline (and then possibly asking the Generator agent to refine or document the generated tools), or by using the spec data to guide the LLM generation. Using the spec data as direct input to the prompt can help ensure the LLM doesn't hallucinate endpoints or parameter names. For example, we could serialize part of the spec (like a particular path's info) into a concise format and prompt the LLM: *"Write an MCP tool for the following endpoint: [spec excerpt]. It should take these parameters and return the result as text."*. The LLM then essentially does a codegen similar to Swagger Codegen but in natural language style.

- **Unstructured Docs:** When only human-readable docs are available, we rely on the Analyzer agent's NLP capabilities. It might need to identify phrases like "**GET /users/{id}:** returns the user details…" and capture that. A possible approach is to use prompt templates that extract info. For example: *"List all API endpoints described in the text and for each give: HTTP method, endpoint URL, purpose, parameters, response format."*. The agent might output a list which we can then refine into the tool spec format. There could be challenges if the docs are lengthy or split across pages – chunking and summarizing might be needed. A vector database could be used: embed the documentation text, and for each potential tool or query, retrieve relevant sections for the agent to read. This reduces context size for the LLM. We should also look for any provided examples in the docs, since those are gold for testing later (the tester can use example inputs or verify the example output matches what our tool returns).

- **Prioritization:** Not all documented features may be relevant. The Analyzer could apply simple rules, like skip internal or admin endpoints, or label certain tools as "nice-to-have" vs "required" based on context (for instance, if documentation sections are labeled beta or deprecated). The prioritization could also be user-driven: maybe the user indicates which features they care about. For an MVP, we might attempt to generate everything and let the human trim the list at the review stage if needed. Future enhancements could include a heuristic or even an interactive Q&A: the Analyzer might ask the user "The API has 50 endpoints; do you want tools for all, or only certain categories?" to narrow scope.

- **Authentication and Setup Info:** Many apps have an auth process (API keys, OAuth). Documentation might include this in an introduction. The Analyzer should detect if an API key or token is required for calls. If so, it can prescribe how the Generator should incorporate it (e.g., include a header `Authorization: Bearer {API_TOKEN}` for each request). Possibly, the Analyzer might suggest creating a dedicated `authenticate()` tool or at least mention that each tool function should pull a token from environment or a config. It could even generate a resource in MCP for the API key (e.g., a resource that stores the key securely). This requires careful design to not expose secrets. Likely, we would opt to have the API key be read from an environment variable or config file that the generated code accesses (this way, the actual key is never hardcoded or handled by the LLM). We should mark this clearly so the human can later plug in the real key on deployment.

- **Rate Limits and Usage Guidelines:** If docs mention “limit X requests per minute”, the Analyzer could surface that information too. It might not directly affect code (beyond maybe adding a note or using a rate-limiter library), but it’s useful context for testing and for the user to know. The system can incorporate a simple delay between tool calls in test if such limits exist to avoid throttling.

In short, the Analyzer agent must transform documentation (whatever form it’s in) into a clear set of requirements for the toolset. It is the stage most prone to information loss or misinterpretation, so investing in robust parsing here is important. We can gradually enhance this with more sophisticated NLP or even by fine-tuning an LLM on API docs if needed. The success of the later stages hinges on the accuracy of this step’s output.

MCP Tool Generation Rules and Standards

When generating MCP tools, consistency and correctness in format are crucial so that the resulting tools are immediately usable. The design therefore establishes some *rules and conventions* the Generator (and Validator) should enforce:

- **Function Signatures:** Use descriptive yet concise names (likely based on the API endpoint or action, e.g., `createOrder`, `listUsers`). Parameter names should match those in the API (if an API uses `user_id` in query, the tool param should probably be `user_id` as well, to avoid confusion). Include type hints for all parameters and the return type. Return type might often be `str` or `Any` (if returning raw JSON). We might choose to return JSON responses as Python dicts (type hint `-> dict`) which FastMCP will turn into a structured content automatically. However, returning as `str` after formatting nicely can be more user-friendly if the data is mostly for reading. This is a design decision possibly influenced by how the platform uses the results (for an agent’s consumption vs. direct display). For now, we can aim to return plain text or dicts.
- **Docstrings:** Every tool function should have a docstring that briefly explains what it does, and lists the Args. This can be directly taken or paraphrased from the API documentation. For example:

```python
async def get_user_details(user_id: str) -> dict:
    """Retrieve details for a given user ID.
    Args:
        user_id: The unique identifier of the user.
    Returns:
        A dictionary of user details, including name, email, and other
profile info.
    """
    ...
```

This ensures when a client calls `list_tools`, they see a helpful description [3]. The Generator should be careful to not just copy large swathes of text (maybe a sentence or two is enough) and to sanitize any references that don't make sense out of context.

- **Error Handling:** Wrap external calls in try/except blocks. If an HTTP call fails (network error or non-2xx status), have the tool return a graceful message or an empty result with a message. For example,

```
try:
    resp = await client.get(url)
    resp.raise_for_status()
except Exception as e:
    return f"Error fetching data: {e}"
```

This way the tool doesn’t crash the entire agent; it returns a message the AI can handle. We should consider how we represent errors: possibly as a simple text string (like “Unable to fetch data, got 404.”). More advanced approach: use MCP’s ability to return structured content with an error flag, but that might complicate things. Simplicity is fine initially. The main idea is no uncaught exceptions.

- **Content Formatting:** If the API returns JSON, decide whether to return it as JSON string, Python dict, or formatted text. The OpenAI docs for connectors show that for certain standardized tools like `search`, they expect a JSON string inside a `text` content item [34]. In our general case, we have flexibility. If the primary consumer is an AI model, sometimes giving it JSON is useful so it can parse easily, other times a readable text is better if it’s just going to be fed back in conversation. Perhaps a good compromise: return a Python dict (which FastMCP will likely encapsulate as structured content) so that if an agent using LangChain or similar gets it, it can parse, but also convert that to a string for the model. Given our focus is design, we can state that as a decision point. For implementation, starting with returning Python native structures or strings that closely reflect the API response is sufficient. The Validator can ensure this matches what the spec says (e.g., “returns a list of products” should indeed return a Python list or JSON representing that list, not something unrelated).

- **Reusability and Modularization:** If multiple endpoints require similar setup (like all calls need an auth header or hitting the same base URL), the generator might create a global constant for the base URL and use it in all tools (as seen in the weather example with `NWS_API_BASE` [35]). It might also generate a single function to get an auth token if needed and call it in each tool. However, since our input is no code (just docs), we likely don’t have any ready authentication function unless we generate it. We can incorporate a simple pattern if, say, an API key is required:

```
API_KEY = os.getenv("API_KEY")  # assume key is set in env
headers = {"Authorization": f"Bearer {API_KEY}"}
```

The generator should include `import os` if it does that. We should define whether to place common code in a separate section (like outside of tool functions but in the same file) or just within each tool. For MVP, duplicating a bit of code in each tool might be acceptable (less elegant, but straightforward). For example, each function might independently define `base_url` and do the request. This avoids issues of one function relying on another being defined before it, etc. In future, an optimization could be to refactor common code after initial generation – a step that could even be another agent's role (like a Refactor Agent). But that might be over-engineering at the start.

- **Logging and Debug Info:** During testing or even production use, having logs is helpful. FastMCP suggests not printing to stdout (for STDIO servers) as it can corrupt the protocol [36]. Instead, you'd use `ctx.info()` or logging to stderr. For simplicity, we might not add logging in generated code at first, or if we do, ensure it doesn't break anything (maybe only in a debug mode). This is a minor detail but worth the Validator checking (no stray print statements, etc., as they warned [37]).

By codifying these rules, we ensure the Generator agent's output meets a **"definition of done"** for each tool. The Validator agent will specifically look for adherence to these rules and flag any violation (which then might be auto-correctable by another gen pass or by human in review).

We also align these rules with official standards. For example, using type hints and docstrings is not just for show – FastMCP uses them to auto-generate tool definitions behind the scenes [38], which likely populates things like the tool schema that clients see. Adhering to that makes our tools properly integrated. Also, if in the future we want to utilize something like **LangChain's MCP integration**, having well-defined tools will allow those frameworks to load them easily as `langchain.tools` [39].

Automated Testing and Validation Workflows

Testing without source code access emphasizes **black-box testing** and **contract testing**. Our design essentially treats the external API as the ground truth and the documentation as the expected contract. The Tester and Validator perform a role similar to what QA teams do when validating an API implementation against its spec. In fact, as mentioned, tools like Dredd or Postman's test suites do exactly this by reading the OpenAPI spec and calling the API to ensure it complies [16]. Our system mirrors this but in a more dynamic, AI-driven fashion for each generated tool.

One workflow detail: if the external API is accessible (e.g. a public API or a test instance provided), testing is straightforward by actual calls. If the API is private or not reachable in the test environment, we may need to simulate responses. In some cases, the application spec might come with example responses. We could use those to simulate the API behavior (essentially mock the API). However, using real calls is preferable for realism. The infrastructure should allow the testing agent's environment (which likely runs on a server or cloud function) to make HTTP requests to the target endpoints. We must handle secrets carefully if needed (for instance, the orchestrator can retrieve an API key from a secure store and provide it to the environment where tests run).

Failure handling: When a test fails, the Tester logs it and the Validator decides next steps. If a failure is minor (like a slight format issue), we attempt auto-correction. If a failure is major (say the tool completely mis-called the API, hitting a 404 because of wrong URL), we definitely loop back and fix the URL in code via another generation iteration. The iterative approach is a key strength — it adds resilience to the generation process. The system is not expected to get everything perfect in one shot, but through tests and fixes it will *approach* perfection. This is how we mitigate the unpredictability of LLM output.

The loop continues until tests pass. We should define a cutoff in case of endless loops (if the spec is impossible or LLM stuck). After, say, 3 attempts, if a particular tool still fails, the system should flag it to the human with an explanation (e.g., "Tool X still failing after 3 attempts: API returned unexpected data we can't handle"). The human might then step in to debug or decide to skip that tool.

OpenAPI Spec Validation: If an OpenAPI spec was given, beyond just running the tool, we could incorporate a direct spec validation. For example, use a JSON Schema validator (the OpenAPI schemas can be converted to JSON Schema) to check the actual JSON output. If our tool returns a Python dict `result`, we can validate `result` against the schema for that endpoint's response. There are libraries in Python (like `jsonschema`) to do this. The Validator agent can perform this check as part of its routine. If the output doesn't match (extra fields, wrong types), it can flag those details. This adds a formal contract test on top of the functional test.

Parallel vs Sequential Testing: For initial implementation, sequentially testing tools is fine. But if there are many tools, this could take time (especially if some calls are slow or if we include delay for rate limits). The architecture could allow the Tester agent to spawn multiple threads or asynchronous tasks to test different tools in parallel to speed up the process. However, concurrency must be controlled if using the same API key to avoid hitting rate limits too quickly. Possibly, test low-risk read endpoints in parallel, but throttle those hitting heavy endpoints or making changes. Given an MVP likely won't have an enormous number of tools, sequential is acceptable and simpler.

Handling Edge Cases and Limitations

Despite careful design, there are inherent challenges and limitations:

- **Incomplete or Ambiguous Documentation:** If the provided documentation is lacking details (e.g., it doesn't clearly state the format of responses, or some parameters are not documented), the Analyzer (or later, the Generator) might have to make educated guesses. This is a classic garbage-in, garbage-out risk. The system should flag ambiguities for human input. For example, if the doc mentions an endpoint but not what it returns, the Analyzer could include a note like "(response not documented, assumed to be JSON)" and the Validator might recommend a manual check. Human review is especially critical in such cases to avoid the agent making a wrong assumption that slips through. In worst case, the tool might function but not exactly meet user needs because of a misunderstanding.
- **Dynamic or Stateful Operations:** Some APIs require multiple steps (e.g., obtaining a token from a login endpoint, then using it, or making a POST then a GET to check status). Our pipeline currently generates mostly stateless tools (each corresponds to one call). If an operation is stateful (like "start a job" and "check job status"), the Analyzer might produce two tools for them. But coordinating them is beyond just tool generation – it might require an agent at runtime to use both in sequence. That's fine (the AI agent can do that). But we should ensure the tools provide enough info to chain them (e.g., the "start_job" tool should return a job ID which the "check_status" tool can take). If such patterns exist, the Analyzer should catch them and ensure outputs/inputs align (maybe by reading in documentation that likely mentions it). This is complex but important for completeness.
- **Rate Limiting and Quotas:** As noted, the test phase must be careful. Also, in production, if the AI agent uses these tools heavily, we could hit limits. The system design could incorporate a global rate limiter or use the API's recommended wait/retry patterns. Perhaps generate code that checks for 429 status and in response returns a message like "Rate limit hit, please wait", which the AI could interpret. These details might be considered advanced; initially, we might assume moderate usage. But it's something to plan for if this system is to be robust in real-world usage.
- **Error Propagation to AI:** When an MCP tool returns an error message string, how does the AI handle it? Likely it will just see it as the result of its action and may relay it to the user or decide an alternative. This is fine. But we should ensure errors are phrased in a helpful way. Perhaps include in the docstring or metadata what errors might occur. (e.g., a user might ask the AI: "Get details of user 123." If the tool returns "User not found.", the AI should convey that to the user.

That's okay. If it returned an obscure technical error, the user might be confused. So we prefer friendly error messages in tool outputs.)

- **Security:** The system must avoid exposing sensitive data. Since we work only from docs, we likely won't have secrets except the ones needed to call the API (keys). Those should never be printed or returned via a tool. The generated code should not log the key or include it in responses. The Validator can scan for accidental inclusion of key in strings, etc. Another security aspect is injection – if the AI is creating code, we must ensure it doesn't inadvertently create a vulnerability, like constructing an OS command. However, given our context is calling known APIs with fixed endpoints, this is less of a concern than if the AI was allowed to run arbitrary system commands. We limit the scope by design to just HTTP requests and data formatting.

- **AI Limitations and Hallucination:** The LLM might hallucinate nonexistent endpoints or parameters if the docs are unclear. This is where combining deterministic parsing and LLM insight helps. And the testing phase will catch if it called a wrong URL (likely a 404). At least with a real API, hallucinations get exposed by failure to fetch. That triggers a correction loop. There is a risk though: what if the LLM hallucinates a minor detail that still passes tests? (Maybe it calls a slightly wrong endpoint that by coincidence returns something.) This is unlikely if we have the official spec to compare. If we rely purely on textual docs, slight misinterpretations could slip through if the tests aren't comprehensive. Human review should catch weird cases (like "this tool is returning data but from the wrong endpoint" – a human who knows the domain might see that). Ultimately, having humans set the specification and then verifying final results mitigates hallucination risk.

- **Feasibility and Expected Accuracy:** Given state-of-the-art LLMs and multi-agent methods, we expect the system to successfully generate the majority of tools correctly after a couple of iterations. AgentCoder's results (over 90% success in one shot for code problems with GPT-4 [25]) are encouraging – albeit coding puzzles differ from API integration. The advantage in our scenario is that we have a fixed reference (the API itself) to test against, which guides corrections. We anticipate that straightforward CRUD style endpoints will be handled well. More complex logic (if any required) like "if this parameter is X, then do Y" as per docs will rely on the LLM catching that nuance. It's possible some fine-tuning or adding those details into prompts explicitly will help. Over time, as the system generates more tools, we could accumulate a library of patterns (like how to handle pagination, or how to structure search query results, etc.) which can be reused to further improve initial generations.

In summary, while limitations exist, the design proactively addresses many of them via the combination of automation and human oversight. The iterative loop and approval step act as fail-safes for scenarios the AI can't fully resolve on its own.

Implementation Roadmap

Implementing this system will be an iterative project in itself. We outline a roadmap with phases, acknowledging risks and recommending tools and infrastructure choices along the way.

Phase 1: MVP (Minimum Viable Prototype)

Scope: Focus on the core end-to-end functionality with a simple scenario. For the MVP, we might assume a well-documented public API (e.g., a weather API or GitHub API) and generate a couple of tools for it. We will implement the pipeline in a straightforward sequential manner, possibly even with some steps initially done in a semi-automated way (to validate the concept).

- **Analyzer (MVP):** Implement basic parsing for OpenAPI specs first (since that's low-hanging fruit). If an OpenAPI is provided, skip LLM and directly produce tool specs from it. For textual docs, use a limited prompt to extract a list of endpoints (perhaps require docs to follow a pattern or headings for now). Make this agent a simple Python function or script that yields a structured spec (can be hardcoded for the test API to start, to simulate it).
- **Generator (MVP):** Use a reliable LLM (GPT-4 via OpenAI API, for example) to generate code from a given spec. Start with prompting it for one tool at a time. Manually inspect or unit test the code in development to ensure the prompt is tuned correctly. The generation could be done within a Python environment (e.g., using the OpenAI SDK) orchestrated by our script. We'll likely need to provide the model with some context about MCP format. Perhaps include a short system message like "You are a code generator for an MCP server. Follow examples strictly." Then feed examples and the task.
- **Testing (MVP):** Rather than building a full agent, in MVP we can directly call the function and check outputs. For instance, after generation, manually call the tool function (or via an MCP client if possible) in Python to see if it returns plausible data. If actual API calls can be made (with a known test key), do it. If not, stub the HTTP call (maybe monkeypatch `httpx.get` to return a sample response for testing). Keep it simple just to validate the loop concept.
- **Validator (MVP):** In initial prototype, the "validator" can be just a series of checks in code (no AI). For example, after running tests, our script checks if exceptions were thrown. It can also verify presence of `@mcp.tool` decorators by scanning the code text or the list_tools output. Basically, implement a rudimentary validator that returns true/false for each tool passing. If something fails, we can print a message or even attempt one regeneration with a slightly modified prompt. This could even be done manually at first (developer notices an error and tweaks the prompt or code).
- **Human Review (MVP):** In the first prototype, the development team itself plays the "human" by examining the final output. Since it's all happening in a controlled environment, we won't implement the UI for approval just yet, but we'll note at what points manual inspection happened.

Goal of MVP: Demonstrate that given an API spec, the system can produce a working MCP server code and successfully call the real API via the generated tool. For example, show that by providing a weather API spec, the tool `get_forecast(city)` is generated and when called, returns actual forecast data as per the API. This will validate the concept and flush out obvious issues in the pipeline.

The mantra for MVP is **"Start simple"** – as the Google ADK team suggests, we won't try to build the full nested loop with all bells and whistles at once [40]. A basic sequential chain that we can test end-to-end is priority. Complexity (like parallelism, advanced parsing, etc.) comes later.

Phase 2: Expand Capabilities and Robustness

Once the basic pipeline works for a simple case, we expand:

- **Robust Analyzer:** Incorporate the LLM-based analysis for arbitrary docs. We might integrate a small language model prompt that can handle longer documentation by chunking. Also start handling authentication info and more complex parameter types. Possibly integrate an existing tool like APIExtractor (if any) or at least structure our own.
- **Enhanced Generator:** Build prompt templates for consistency. Perhaps introduce a "Generator Agent" class that can be fed different tasks (initial generation vs fixing a specific issue). We might

start using few-shot learning: include 1-2 examples of tools in the prompt. We will also hook in a static analyzer (like running `pyflakes` or similar) to catch errors just after generation automatically. Additionally, consider memory: if generating multiple tools, ensure the context doesn't overflow. We might generate tools one by one to keep prompts small, or generate all in one go (risky for context length). Generating individually might be better, though one tool may need knowledge of others if shared data (rare).

- **Automated Tester Agent:** Develop a testing module that can run within our system. Possibly integrate something like **Dredd** or write custom calls. We may allow concurrency if needed. Also implement JSON schema validation using the spec. At this stage, if the API is open, we use the real endpoints. We handle simple secrets via environment variables for tests.
- **Validator Agent with LLM:** We can consider using an LLM to assist in validation for nuanced issues. For example, we could prompt an LLM: "Here is the tool code and here is the spec, do you see any inconsistencies or possible improvements?" This could catch things like "the tool returns too much data, maybe filter it" or "doc says parameter should be int but code treats as str." However, this might be overkill. Many validations can be coded (like schema checks, etc.). Perhaps use LLM for style suggestions or ensuring descriptions are user-friendly. This is optional and to be explored.

- **User Interface & Approval Workflow:** Integrate with Next.js front-end properly. Develop pages/components for uploading specs, viewing generated plan, and reviewing final tools. Use Next.js API routes or a backend endpoint to trigger the Python orchestration. For the human approval, implement a simple UI where each tool is listed with a status (maybe green/red from tests) and a code snippet. The human can click "Approve" or "Request Changes". If changes, perhaps allow them to edit description or note something which could then be fed back to an agent or require a manual fix. The "Approve" button triggers the deployment sequence.

- **Deployment Integration:** Scripting the deployment of the MCP server with new tools. If the platform is cloud-based, this might mean pushing to a repository or calling a deploy hook. Alternatively, if the MCP server is part of the Next.js app (less likely, since Next.js is Node – more likely MCP server runs separately), ensure the front-end knows how to contact it. We might deploy the MCP server to a serverless function or container and provide its URL to Next.js. This requires some DevOps planning. Using **FastMCP Cloud** (if the team opts, since FastMCP offers a cloud deployment option [41]) could be a quick way to host the server with minimal fuss. But a self-hosted approach (embedding in our infrastructure) gives more control.

- **Logging and Monitoring:** Add logging of agent decisions for later analysis. For instance, log what corrections were made in each iteration; log how often human intervention was needed. This will be useful to measure improvement over time.

Risk Factors in Implementation:
- *LLM Dependency:* The quality of output heavily depends on the LLM's abilities. GPT-4 is strong but costly; perhaps we can use it for generation and a cheaper model for parsing if needed. We should also prepare for cases where the LLM fails or the API is not reachable. Always have timeouts and error catching around LLM calls and external calls to keep the pipeline from stalling indefinitely. - *Cost and Performance:* Each run might entail multiple LLM calls (for multiple tools and for iterations) plus external API calls. For large APIs, this can add up in time and cost. We should optimize by reusing context (maybe the LLM can generate multiple tools in one broader prompt if that's more efficient context-wise). Also, parallelizing some steps can reduce wall-clock time. Caching could be employed: e.g., if documentation doesn't change, we could cache the analyzed spec or even the generated code for endpoints so we don't redo it next time (unless asked to). But caching AI outputs must be done carefully (maybe better to re-run generation for fresh output if context changes). - *Tool Reliability:* If the external

API has unstable sandbox (some test environments are flaky), our tests might fail due to external issues, not our code. We should differentiate that. Perhaps incorporate retries or identify when a failure is due to network vs code. - *User Experience:* The flow should be intuitive for the user. They provide docs, and after some processing time (which could be several minutes for big APIs), they get something to review. We might need to show a progress indicator or logs in real-time to keep them engaged (like “Generating tool X… Testing tool X…”). Next.js can use server-sent events or WebSockets to stream status updates from the backend orchestrator.

Phase 3: Advanced Features and Case Studies

After successfully deploying the system for initial uses, we can consider advanced improvements:

- **Parallel or Distributed Agent Execution:** Possibly use multiple LLM instances (or different LLMs) for different agents in parallel. For example, the Analyzer and Generator could potentially work in tandem on different parts of the spec (if documentation is modular, one could analyze one section while another generates a previously analyzed section). However, synchronization and ordering are important because generation depends on analysis output. But perhaps splitting by endpoint groups could be done (with one agent handling one set of endpoints and another a different set). This would shorten processing time for very large specs.
- **Integration of Learning:** Over time, the system could learn from its mistakes. We might build a knowledge base of common fixes. If we see often the LLM forgets to handle pagination links, we can preemptively include code for that if documentation contains certain keywords. Or maintain a library of templates for certain standard APIs (like known patterns for CRUD). Essentially, the more we use the system, the better it can get if we allow it to accumulate patterns (with developer oversight).
- **Multi-Modal Inputs:** Some app specs might include UML diagrams or other images. Extending the Analyzer to use image-to-text (OCR) or interpret diagrams (maybe not soon, but conceptually possible) could be interesting.
- **Scribe Agent (Documentation):** In the Reddit example, they had a Scribe agent for documentation [42]. We could similarly introduce an agent to produce documentation for the generated tools or integration guides for the platform team. For instance, after deployment, an agent could generate a markdown file summarizing what tools were added, how to use them, example calls, etc., which is great for internal knowledge. This would help the team or even end-users understand the new capabilities. While not in initial scope, it’s a nice add-on when time permits.

- **Continuous Update Mode:** If the target application’s API changes (version updates, new endpoints), the system could be rerun on the new documentation to update the MCP tools. We could even automate detection of changes if docs are accessible via URL (poll occasionally). Then generate diff and perhaps auto-update tools. This could make the integration always up-to-date with minimal manual effort – a powerful proposition for maintaining connectors.

- **Tool Recommendations:** The question hints we might provide tool recommendations. Possibly, if multiple integration options exist, the Analyzer might suggest one approach vs another (like using a certain endpoint vs another to achieve a goal). However, likely “tool recommendations” was referring to recommending which technical tools (frameworks, libraries) to use in building this system. For that:

- We recommend **FastMCP** as the MCP server library (since it’s purpose-built and has convenient features for testing and deployment [18] [41]).

- For orchestrating multi-agent, since our backend is Python, we could use frameworks like **LangChain** which supports agents and even has some MCP client adapters [43]. However, LangChain might be more useful if we want the agent to *use* tools; here we are *creating* tools. We might instead use a simpler orchestration, possibly just writing our own sequence with OpenAI API calls for each agent role. The Google ADK is interesting but it's TypeScript and targeted at their LLM (Gemini). We can take pattern inspiration from it but implement in Python.
- Using **AsyncIO** in Python will help manage asynchronous calls (especially to test tools concurrently or to call LLM and wait for responses).
- For parsing specs: the **swagger-parser** (for OpenAPI) or just PyYAML if we parse ourselves. For textual data, possibly using **spacy** or regex for some parts, but LLM might handle it end-to-end.
- Testing: use **httpx** or **requests** for HTTP calls within tools. Possibly use **pytest** to structure tests if we want to formalize it, but that might be overkill; our Tester agent can just call functions and assert conditions directly.

Infrastructure Planning: - The system will likely consist of a **Python service** (could be a FastAPI app or a set of cloud functions) that executes the agent pipeline. This service interacts with the OpenAI API (or other LLMs) and potentially with the target application's API for testing. - The **Next.js front-end** will need to communicate with this Python service. We can set it up via API endpoints (Next.js API routes could proxy to the Python service) or via direct client calls if the Python service provides an HTTP API. Since Next.js runs on Node, it might not directly run the Python code; it's cleaner to separate. - Security: ensure the Python backend is secured (only accessible to the authorized front-end or internal network) because it might have access to sensitive keys and can execute code. - **Data storage:** Keep records of generated tools and versions, perhaps in a database. So we have a history (useful for rollback if a new generation fails, we can revert to a previous stable tool version). - **CI/CD for the tools:** If treating the MCP server code as a codebase, we might integrate with Git. For example, upon final approval, commit the new tool code to a repository, run automated tests (could be the same tests we did, but nice to double-check in CI), and deploy. This way standard software engineering practices complement our AI generation pipeline. This is essentially bringing the AI's output into the regular development workflow at the final stage, which is good for maintainability.

Risks and Mitigations

- **Incorrect Code Deployment:** If a bug slips through and gets deployed, it could affect users or data. Mitigation: thorough testing and human review as described. Additionally, feature flagging could be used (deploy the new tools but not expose them to all users until validated in prod).
- **Over-reliance on AI:** The system should not operate in a fully automatic mode for mission-critical systems without oversight. That is why the human-in-loop is mandatory at deployment in our design. Over time, as confidence builds, the human might just rubber-stamp approvals, but the option to intervene is always there.
- **Model and Integration Drift:** As APIs evolve or if the documentation is wrong (yes, that happens), the AI might implement something that doesn't actually work. Having a tight test harness with the real API catches most of this. But if documentation and actual behavior diverge subtly (like a field named differently), the test will reveal it and our loop adjusts the code to actual behavior – effectively, our system can act as a documentation correctness verifier too. This is a positive side-effect; we might sometimes end up updating our internal spec based on reality.
- **Scalability:** If multiple users want to use this system simultaneously for different apps, how well does it scale? LLM calls are the heavy part, which scale linearly with number of tasks (but can be parallelized if calls don't interfere). We should ensure the backend can queue or run multiple pipelines in parallel if needed, possibly with task scheduling. This might not be needed initially, but if this becomes a service offered in a platform, multi-tenancy must be considered.

- **Cost management:** Using GPT-4 extensively can be expensive. In a production scenario, we might explore using GPT-3.5 for some steps (e.g., initial parsing might be fine with a cheaper model, and use GPT-4 for code gen and critical parts). Also, limit tokens by providing only necessary context. We can also consider open-source models if they become good enough for code (there are ones like Code Llama, etc., but as of 2025 GPT-4 is likely superior in quality). Perhaps fine-tune a smaller model on our domain (API integration tasks) to reduce cost in the long run. But that's an advanced consideration.

Tool Recommendations for Development

To implement this system, the following tools and libraries are recommended:

- **FastMCP** (Python MCP server framework) – for easily defining and running MCP tools and server [44]. We'll use this to host our generated tools. Its utilities for testing (in-memory client) and deployment (FastMCP Cloud or self-host HTTP) are very handy [18] [41].
- **OpenAI GPT-4 API** – for the LLM powering the agents. Possibly supplemented by Anthropics' Claude (especially since a Claude-based example was given for multi-agent roles [8]). Claude is known to handle longer context well, which could help with large documentation. Either or both can be tried.
- **LangChain or Custom Orchestration** – We might not need the full LangChain overhead, but it does have an **MCP client adapter** which could let us easily connect an agent to the MCP server to test calling the tools [43]. However, for the creation pipeline, probably a custom script is fine. The Google ADK, while instructive, is in TS, so we won't directly use it, but we learned pattern ideas from it (sequential pipeline, human approval etc.). We can implement similar patterns in Python manually.
- **Testing Tools:** `requests` / `httpx` for making HTTP calls within tools. For schema validation, `jsonschema` library. Possibly `pytest` if we formalize tests, but an internal harness might suffice.
- **Version Control:** Git for managing the MCP server code. Possibly automatically commit changes via a script. This provides traceability of what the AI changed, which is useful for audit.
- **Next.js Components:** Leverage Next.js APIs for UI. We might use Next's serverless functions to interface with the Python backend. Alternatively, run the Python backend as a separate service accessible via REST API to the Next.js front-end. The choice depends on deployment constraints (Next.js could be hosted on Vercel and Python on another cloud, communicating securely).

Finally, we plan incremental integration testing of the whole pipeline on a non-production environment. For instance, spin up the system to generate tools for a sample API and have a staging MCP server where we can play with the tools via a chat interface (like hooking it to ChatGPT via developer mode to simulate usage). This will show end-to-end that a user query can invoke one of these newly minted tools correctly.

Conclusion

This design outlines a comprehensive approach to automate the creation of MCP tools and integrate them into a Next.js platform, using a combination of AI agents and traditional software testing practices. By dividing the work among specialized agents – Analyzer, Generator, Tester, Validator – and introducing iterative feedback loops, the system can achieve a high level of accuracy and reliability in generated code [22] [29]. The incorporation of human-in-the-loop checkpoints ensures that critical decisions and final deployments are supervised for safety [23].

We emphasized a design-centric perspective: focusing on the flow of information (from documentation to deployed tool), the interactions between components, and the protocols (MCP, JSON-RPC, etc.) that glue it together. This approach is resilient to the uncertainties of AI output because it doesn't rely on a single-shot solution; instead, it's an evolving pipeline with multiple validation gates.

Upon implementation, this system will empower rapid development of integrations: given an API spec, within a short time the platform will have new MCP tools ready to use. This can dramatically reduce the manual effort typically required to write and test integration code, while still maintaining quality through automated testing and human approval. It represents a significant step towards **AI-assisted software development pipelines**, where repetitive coding tasks are handled by AI agents under the guidance of established software engineering principles and human oversight.

Sources:

- OpenAI, *"Building MCP servers for ChatGPT and API integrations"* – Explanation of Model Context Protocol usage and tool format [1] [12].
- CodeSignal Learn, *"Exploring MCP Primitives: Tools, Resources, and Prompts"* – Descriptions of MCP tools and how they're defined/discovered [2] [3].
- Saboo, S. (2025), Google Developers Blog, *"Developer's guide to multi-agent patterns in ADK"* – Rationale for multi-agent architecture and design patterns (Sequential Pipeline, Human-in-the-loop) [4] [23].
- Huang et al. (2023), *"AgentCoder: Multi-Agent Code Generation with Effective Testing and Self-optimisation"* – Demonstrates multi-agent code generation with iterative test feedback, using three specialized agents for efficiency [45] [29].
- Reddit r/ClaudeAI, *"How I Built a Multi-Agent Orchestration System with Claude"* – Example of 4-agent collaboration (Architect, Builder, Validator, Scribe) and shared communication approach [8] [46].
- R3d_cr0wn (2023), DEV Community, *"Automated Contract Testing with OpenAPI and Dredd"* – Highlights using OpenAPI spec as a contract for testing APIs with tools like Dredd [16] [17].
- J. Lowin, *FastMCP GitHub Repository* – FastMCP features for building/testing MCP servers and note on generating servers from OpenAPI specs [18] [33].

---

[1] [11] [12] [21] [34] Building MCP servers for ChatGPT and API integrations
https://platform.openai.com/docs/mcp

[2] [3] [10] [26] Exploring MCP Primitives: Tools, Resources, and Prompts | CodeSignal Learn
https://codesignal.com/learn/courses/developing-and-integrating-a-mcp-server-in-python/lessons/exploring-and-exposing-mcp-server-capabilities-tools-resources-and-prompts

[4] [5] [6] [7] [23] [24] [32] [40] Developer's guide to multi-agent patterns in ADK - Google Developers Blog
https://developers.googleblog.com/developers-guide-to-multi-agent-patterns-in-adk/

[8] [9] [42] [46] How I Built a Multi-Agent Orchestration System with Claude Code Complete Guide (from a nontechnical person don't mind me) : r/ClaudeAI
https://www.reddit.com/r/ClaudeAI/comments/1l11fo2/how_i_built_a_multiagent_orchestration_system/

[13] [14] [39] [43] Model Context Protocol (MCP) - Docs by LangChain
https://docs.langchain.com/oss/python/langchain/mcp

[15] [35] [36] [37] [38] [44] Build an MCP server - Model Context Protocol
https://modelcontextprotocol.io/docs/develop/build-server

[16] [17] Enforcing API Correctness: Automated Contract Testing with OpenAPI and Dredd - DEV Community
https://dev.to/r3d_cr0wn/enforcing-api-correctness-automated-contract-testing-with-openapi-and-dredd-2212

[18] [19] [20] [33] [41] GitHub - jlowin/fastmcp: The fast, Pythonic way to build MCP servers and clients
https://github.com/jlowin/fastmcp

[22] [25] [27] [28] [29] [30] [31] [45] AgentCoder: Multi-Agent Code Generation with Effective Testing and Self-optimisation
https://arxiv.org/html/2312.13010v3