You’re thinking in exactly the right direction: treat every “agent” (or capability boundary) as its own MCP Server process, each with its own Kernel (and therefore its own model binding). The orchestrator then becomes a pure coordinator that never holds a model—it just routes intent and composes results. Below is a concrete, stepwise architectural path from your current in‑process Kernel usage to a fully decoupled, multi‑process (eventually multi‑container) agent mesh.

**1. Core Principle: Model Binding Lives with the Agent Process**

Because the model (Azure OpenAI deployment) is registered inside a [Kernel](vscode-file://vscode-app/c:/Users/cesardl/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html), and a [ChatCompletionAgent](vscode-file://vscode-app/c:/Users/cesardl/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html" \o ") depends on a [Kernel](vscode-file://vscode-app/c:/Users/cesardl/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html), the clean separation is:

* Chat Agent MCP Server = owns Kernel + model registration + answer synthesis logic.
* Orchestrator MCP Server = owns no model; just calls tools on other MCP servers (KB + Chat).
* KB MCP Server = already separate (provides domain content / search).

Think of each MCP Server as a “capability microservice” for AI reasoning or data access.

**2. Target Interaction Flow (After Decoupling)**

1. Client (your IDE / higher-level orchestrator) calls Orchestrator MCP tool: ask\_domain\_question.
2. Orchestrator calls KB MCP tool (search\_knowledge or get\_kb\_content) to gather snippets (if heuristics allow).
3. Orchestrator then calls Chat Agent MCP tool, e.g. generate\_answer, passing:
   * User question
   * KB snippets (array or structured objects)
   * Optional context flags (requestedMaxKb, heuristicSkipKb, provenance hints)
4. Chat Agent:
   * Builds a prompt (or multi‑message chat sequence) using its local Kernel + model.
   * Returns JSON: { answer, provenance (serviceId, deployment, temperature, mode), usedKbFlag, tokenEstimates }.
5. Orchestrator merges disclaimers, returns final aggregated payload to the original caller.

Result: Orchestrator stays logic-only; Chat Agent becomes the model execution boundary.

**3. Phase-by-Phase Migration Plan**

**Phase A (Current → Transitional)**

* Extract the prompt assembly + agent invocation section from [AskDomainQuestionAsync](vscode-file://vscode-app/c:/Users/cesardl/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html" \o ").
* Wrap it in an internal service class (still in-process) for clean contract definition.
* Define a data contract DTO for: AnswerRequest { Question, Snippets[], Preferences } and AnswerResponse { Answer, Provenance, Disclaimers }.

**Phase B (Standalone Chat MCP Server – STDIO)**

* New project: src/chat-agent/
  + Implements MCP tool: generate\_answer
  + Inside tool: reconstruct current Kernel + ChatCompletionAgent logic.
  + Accept parameters: [question](vscode-file://vscode-app/c:/Users/cesardl/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html), kbSnippets (list), maxWords?, [temperature?](vscode-file://vscode-app/c:/Users/cesardl/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html).
  + Returns structured payload with provenance (serviceId, deployment, temperature, mode, kbGrounded).
* Orchestrator replaces internal method call with MCP client call (like it already does for KB).
* Remove direct Azure OpenAI references from orchestrator project (except maybe for fallback simulation / fake mode).

**Phase C (Add HTTP Transport Option)**

* Wrap chat-agent with an HTTP+SSE transport (when MCP SDK supports or via gateway).
* Orchestrator configuration chooses transport: STDIO (local dev) vs HTTP (remote container).
* Abstract connection creation with a factory: IChatAgentClient behind which you can swap transports.

**Phase D (Containerization)**

* Container 1: orchestrator
* Container 2: kb-mcp-server
* Container 3: chat-agent
* Shared configuration via environment variables (AzureOpenAI\_\_\* only in chat-agent container).
* Optional: network overlay / service discovery (docker compose, k8s service names).

**Phase E (Scaling & Specialization)**

* Multiple chat-agent variants:
  + chat-agent-summary (low temperature)
  + chat-agent-brainstorm (higher temperature / different deployment)
  + Orchestrator chooses which to invoke based on question type or a routing rule.
* Introduce a registry: orchestrator queries a list of available agent endpoints (static config or discovery service).

**Phase F (Session & Memory)**

* If you want multi-turn or memory:
  + Persist conversation state in the chat-agent (Redis / in-memory with session ID).
  + Or orchestrator passes conversation history each time (stateless chat-agent).

**4. Contracts & Payload Design**

Define stable, versioned tool input/output schemas:

Input (to Chat Agent):

Output:

Versioning: add a top-level schemaVersion when you evolve the contract.

**5. Security & Secrets**

* Keep AzureOpenAI\_\_ApiKey only in the chat-agent environment (principle of least privilege).
* Orchestrator & KB servers never see the key.
* For later cloud deployments: store secrets in Key Vault / managed identity; chat-agent fetches them at startup.

**6. Performance / Latency Considerations**

| **Aspect** | **In‑Process** | **Cross‑Process (STDIO)** | **HTTP (local)** | **HTTP (remote/container)** |
| --- | --- | --- | --- | --- |
| Latency Overhead | Minimal | ~2–5 ms | ~5–10 ms | 10–50+ ms (network) |
| Isolation | Low | Medium | High | Highest |
| Fault Containment | Weak | Better | Strong | Strong |
| Horizontal Scaling | Hard | Possible (multi procs) | Good (ports) | Good (k8s/compose) |

Mitigation strategies once remote:

* Batch snippet transmission (compress if large).
* Keep chat-agent warm (no cold start).
* Optionally stream partial answer tokens over SSE.

**7. Routing & Model Flexibility**

To support multiple models:

* Either run multiple chat-agent containers each with its own deployment.
* Or one chat-agent MCP server with a tool parameter modelAlias mapped to different service registrations (multiple [AddAzureOpenAIChatCompletion](vscode-file://vscode-app/c:/Users/cesardl/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html) with distinct [serviceId](vscode-file://vscode-app/c:/Users/cesardl/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html" \o ") values).
* Orchestrator decides alias based on question classification (lightweight rule or future classifier).

1. Observability & Provenance Add these fields gradually:

latencyMs (agent side) modelInputTokenEstimate / modelOutputTokenEstimate orchestratorTraceId (pass through from orchestrator to correlate logs across processes) kbSnippetCount & kbSnippetChars Instrument:

Structured logs (JSON to stderr per MCP rule). Optional OpenTelemetry trace spans (span: orchestrator.ask, child spans: kb.lookup, chat.generate).

**9. Error Handling Strategy**

Common failure modes & orchestrator reactions:

| **Failure** | **Orchestrator Action** |
| --- | --- |
| KB server unreachable | Skip KB, set disclaimer, still call chat-agent |
| Chat-agent unreachable | Return partial answer (maybe last cached KB snippet) or error payload |
| Chat-agent returns empty | Add disclaimer “Empty model response” and fallback small template |
| High latency (>timeout) | Cancel token; return fallback indicating timeout |

Use cancellation tokens in STDIO/HTTP clients for hard timeouts (avoid orchestrator thread pile-ups).

**10. Suggested Minimal First Refactor (No Code Yet)**

1. Define DTO contracts (internal for now) for Chat Request/Response.
2. Move current prompt assembly + agent invocation into a private helper service inside orchestrator (mirrors future remote call).
3. Add a feature flag UseExternalChatAgent=false.
4. When flag = true: call future MCP client (placeholder stub now).
5. Later: implement real chat-agent MCP server following the internal contract.

This de-risks the transition: you can switch sources with configuration toggles.

**11. Future HTTP Gateway (Optional)**

If MCP native HTTP support lags, wrap the chat-agent with:

* Minimal ASP.NET Core API: POST /generate
* Inside controller: reuse same Kernel logic
* Add SSE endpoint for streaming tokens (if needed later)
* Provide adapter layer that implements IChatAgentClient so orchestrator doesn’t care whether underlying transport is STDIO MCP or HTTP.

**12. Summary**

To decouple: Give each agent its own process + Kernel; the orchestrator becomes a pure coordinator invoking MCP tools exposed by those agents. You accomplish model isolation, clear provenance, scaling flexibility, secret containment, and evolutionary path to multi-container and HTTP.

**13. Quick Glossary (Reiterating Provenance Fields)**

* serviceId: Your internal identifier tying an invocation to a specific registered model/service inside a Kernel (important when multiple models coexist).
* deployment: Azure OpenAI “deployment name” (selects the underlying model + capacity settings in your Azure resource).
* temperature: Sampling randomness—lower (≈0–0.3) = deterministic and grounded answers; higher = more creative variance.

If you’d like, I can draft the DTO schemas + a tiny “external client shim” (no runtime change yet) to prepare for Phase B—just say the word and I’ll produce them.