This is an excellent multi-agent collaboration demo with a strong foundation, especially with the Voice Net Protocol and the integration of MCP servers. Based on the provided files, here are some enhancements to improve its capabilities and robustness.

Top Priority: Implement Full Autonomous Tool Use

What: Fully implement the agent-side logic for autonomous MCP tool use.

Why: Several files (e.g., verify_agent_tool_integration.py and demo_agent_airport_search.py) indicate that while the MCP connection works, the BaseAgent class does not yet handle the full tool-use loop (i.e., autonomously deciding to use a tool, sending the request to Claude, and processing the tool_use response). The demo_aerospace.py and demo_multi_aviation.py demos show the *potential* but the agents aren't actually calling the tools themselves based on the conversation.

How:

1. **Modify BaseAgent.generate_response:**
   - Pass the list of available MCP tools (retrieved from MCPManager) to the client.messages.create call using the tools parameter.
   - After receiving a response, check if it contains tool_use blocks.
2. **Implement the Tool-Call Loop:**
   - If tool_use blocks are present, iterate through them.
   - Use the ToolExecutor or MCPManager to call the requested tool with the specified arguments.
   - Format the tool results into a tool_result message.
   - Send *another* request to client.messages.create, appending the original assistant response (with the tool_use blocks) and the new user message (with the tool_result blocks).
   - The final response from this second call will be the agent's natural language answer based on the tool's output.
3. **Pass MCPManager to Agents:** The Orchestrator or the CLI commands should fetch the MCPManager instance and pass it to the constructor of BaseAgent (or specialized agents like AerospaceAgent) so they have access to the call_tool method.

Enhancement 2: Introduce State Persistence and Agent Memory

What: Add a mechanism to save and restore the collaboration state, and give agents a simple

"scratchpad" or memory.

Why: The QUICKSTART.md file mentions "State persistence" as a "Phase 2" goal. Currently, the SharedChannel history is in-memory and lost on restart. Furthermore, agents are stateless; they only know what's in the current context window, making it impossible to perform long-running tasks or remember instructions.

How:

1. **Session Persistence:**
   - Create a simple StateManager class that can serialize the SharedChannel.messages list to a JSON or database file.
   - Add "save" and "load" commands to the CLI (src/cli/main.py) that call the StateManager.
2. **Agent Memory:**
   - Add a self.memory (e.g., a dictionary or a string) to the BaseAgent class.
   - Modify BaseAgent._build_system_prompt to include the contents of this memory (e.g., "CURRENT MEMORY/SCRATCHPAD: ...").
   - Give agents a special internal tool or keyword (e.g., "MEMORIZE: ...") that allows them to update their own self.memory.

Enhancement 3: Improve Orchestration for Directed Communication

What: Make the Orchestrator "smarter" by allowing it to handle directed agent-to-agent requests instead of just polling all agents.

Why: Currently, when one agent speaks (e.g., "Alpha One, this is Alpha Lead, requesting status..."), the Orchestrator simply runs another turn and polls *all* agents to see if their SpeakingCriteria are met. This doesn't guarantee "Alpha One" will speak next.

How:

1. **Enhance VoiceNetProtocol.parse:** Ensure the recipient callsign is reliably parsed.
2. **Modify Orchestrator.process_responses:**
   - Check the most recent message in the channel.
   - If it has a specific recipient_callsign (and it's not a broadcast), find the agent with that callsign.
   - Instead of polling all agents, *force* only the addressed agent to respond (i.e., call agent.respond() directly on that agent).
   - This makes the SquadLeaderAgent's role of "assigning tasks" and requesting status far more effective.

Enhancement 4: Dynamic Tool Discovery and Prompt Injection

What: Automatically provide the list of available MCP tools to the agent's system prompt instead of hardcoding them.

Why: The configs/aerospace.yaml file hardcodes a list of available tools into the system_prompt. This is brittle; if the aerospace-mcp server adds a new tool, the prompt becomes outdated.

How:

1. When creating an agent in interactive_command or demo_aerospace.py, first get the MCPManager.
2. Call manager.get_available_tools() to get the list of tool definitions.
3. Format this list into a readable string (e.g., "You have access to the following tools: ...").
4. Dynamically inject this string into the agent's system_prompt (from the config file or the _default_aerospace_prompt) before instantiating the AerospaceAgent.

Enhancement 5: Support for Multiple Channels and Private Messages

What: Evolve the architecture from a single SharedChannel to support multiple channels and private agent-to-agent messaging.

Why: The current system is a single "party line." A more complex simulation (like a flight crew) might require separate channels (e.g., "Command" channel, "Engineering" channel) or the ability for the Captain to send a private message to the First Officer.

How:

1. **Multi-Channel:**
   - Modify the Orchestrator to manage a dictionary of SharedChannel objects instead of just one.
   - Update agent configurations (configs/default.yaml, configs/aerospace.yaml) to include a list of channels they are "listening" to.
   - Update Orchestrator.run_turn to process messages for each channel.
2. **Private Messages:**
   - This is a larger change, but could be built on the multi-channel concept (a private message is just a temporary channel between two agents).
   - Alternatively, modify the Message class to include a visibility field (public or

private_to=[agent_id]) and have the SharedChannel respect this when providing context windows to agents.