perplexity

Developing a Database Management Agent with Human-in-the-Loop Communication in Python

Creating a database management agent that handles read-only database queries and supports communication with another agent under a human-in-the-loop setup involves combining database querying, agent communication protocols, and user feedback handling. Below is a structured outline of how to design and implement such a system.

1. Core Requirements

- **Read-Only Database Operations**:
  - Listing database users
  - Listing tables
  - Showing user privilege information
  - Displaying active connections
- **Python-Based Implementation**
- **Bidirectional Communication**: The agent can interact both with a peer agent and a human for oversight and approval.
- **Human-in-the-Loop**: The human can approve, reject, or modify the actions or results of the agent.

2. System Architecture Overview

Component	Description
Query Agent	Connects to the database and executes read-only queries
Communication Layer	Handles messaging between agents and the human operator
Human-in-the-Loop	Interface for human review and intervention
Agent Peer	Another agent with which information and requests are exchanged

3. Implementation Steps

3.1 Read-Only Database Agent

- Use SQLAlchemy or native connectors (like psycopg2 for PostgreSQL) in read-only mode.
- Example functionalities:
  - **List Tables**: `SELECT table_name FROM information_schema.tables WHERE table_schema='public';`

- **List Users**: DB-specific user listing queries (e.g., PostgreSQL: `\du`)
- **User Privileges**: Query appropriate privilege tables/views
- **Active Connections**:
  - PostgreSQL: `SELECT * FROM pg_stat_activity;`
  - MySQL: `SHOW PROCESSLIST;`

3.2 Communication Layer

- Use HTTP endpoints (Flask or FastAPI), WebSocket, or message queues (e.g., RabbitMQ, Redis Pub/Sub) for agent-to-agent and agent-human communication.
- Design message formats (JSON recommended) indicating query, result, and action requests.

3.3 Human-in-the-Loop Integration

- Provide a simple web UI (Flask, Streamlit, FastAPI with Jinja2 templates) to display results and prompt human responses (Approve/Reject/Modify).
- All actions flow through the human-review layer before being sent back to the requesting agent or executed.

3.4 Bidirectional Agent Communication

- Define protocols for sending/receiving queries and results (REST endpoints, WebSocket channels, or even asynchronous Python with asyncio).
- Ensure appropriate logging and traceability for all communication and actions.

4. Example: High-Level Code Snippet

```
# Example: Agent skeleton using FastAPI for communication and psycopg2 for PostgreSQL
from fastapi import FastAPI, Request
import psycopg2
import json

app = FastAPI()

# Database connection, ensure user has read-only privileges
conn = psycopg2.connect("dbname=yourdb user=readonly password=*** host=localhost")

def list_tables():
    with conn.cursor() as cur:
        cur.execute("SELECT table_name FROM information_schema.tables WHERE table_schema=
        return cur.fetchall()

@app.post("/query")
async def handle_query(request: Request):
    data = await request.json()
    query_type = data.get("type")
    if query_type == "list_tables":
```

```
        tables = list_tables()
        # Insert human-in-the-loop logic here for approval
        return {"tables": tables}
```

5. Security & Best Practices

- Use **parameterized queries** to prevent SQL injection.
- Ensure the agent's database user has strictly limited (read-only) permissions.
- Implement **authentication and authorization** on API endpoints to prevent unauthorized access.
- Log all queries and actions for auditability.

6. Tools & Libraries

- **Database Access**: psycopg2, PyMySQL, SQLAlchemy
- **Web Framework**: FastAPI, Flask, or Django
- **UI**: Streamlit, Flask templates, or React (for a more dynamic UI)
- **Agent Communication**: WebSockets, HTTP (requests, aiohttp), or message queues
- **Testing**: pytest, unittest

7. Example Workflow

1. **User (Agent or Human) sends a query request**.
2. **DB Agent executes the query in read-only mode**.
3. **Result is presented to the human user for review/approval**.
4. **Upon approval, result and any actions are sent to the requesting party**.
5. **System logs all actions for traceability**.

8. Customization

- Extend with natural language query understanding using NLP (spaCy, transformers).
- Integrate with chatbots for conversational querying.
- Add multi-database support with abstraction layers.

This approach builds a robust foundation for a secure, auditable, and flexible database management agent with human oversight, well-suited for sensitive or compliance-driven environments.

Leveraging Generative AI for Parameter Extraction and Validation

To efficiently process user requests (such as database queries) and ensure all necessary parameters are provided, you can employ generative AI (GenAI) models to both extract structured parameters and validate input completeness. Here's how you can structure such a system:

1. Parameter Extraction with Generative AI

- **Prompt Engineering:** Feed user inputs (in natural language) to a large language model (LLM), using well-crafted prompts that guide the model to extract specific parameters. For example, the model is prompted to extract `table_name`, `user_name`, or `privileges` from phrases like "List all tables for user 'admin'."
- **Few-shot Examples:** Enhance accuracy by providing several input-output sample pairs within the prompt to teach the model the expected parameter schema and format[1].
- **Structured Output:** Ask the LLM to return parameters as a JSON object or dictionary, e.g.:

```
{
  "action": "list_tables",
  "user": "admin"
}
```

- **Handling Ambiguity:** The model can flag ambiguous or missing parameters in its output, asking for user clarification if needed[2] [3].

2. Input Validation and Feedback

- **Required Fields Check:** After extracting the parameters, the model (or your code) compares against a predefined schema of required fields for each type of action/query.
- **GenAI for Intelligent Validation:** You can prompt the LLM to detect missing, malformed, or ambiguous values, and to generate a clear, context-aware message to the user, e.g., "Please specify which database to search."
- **Dynamic Dialogue:** If parameters are missing or unclear, generate follow-up questions directly via the LLM, supporting a conversational "human-in-the-loop" flow that collects all required data before proceeding[4] [3].

3. Implementation Workflow

Step	Description
User Input	User enters a request in natural language
GenAI Parameter Extraction	LLM parses and emits a structured parameter set
Validation Layer	Check for missing/invalid fields using application logic or GenAI
Respond for Corrections	If issues exist, LLM crafts clarifying questions
Final Confirmation	Once all parameters are present and valid, continue the workflow

4. Practical Example (Python-based)

```
import openai

def extract_and_validate(user_input, required_fields):
    # Step 1: Extract parameters with GenAI
    prompt = f"Extract parameters from: '{user_input}'. Return as a JSON. If any required
    response = openai.ChatCompletion.create(
        model="gpt-4-turbo",
        messages=[{"role": "user", "content": prompt}]
    )
    params = response['choices'][^2_0]['message']['content']
    # Step 2: Validate
    missing = [f for f in required_fields if f not in params or not params[f]]
    if missing:
        return f"Missing parameters: {', '.join(missing)}. Please provide them."
    return params
```

- Replace `required_fields` with your action-specific parameter schema.
- Expand this pattern to handle follow-up questions and iterative refinement, simulating a natural conversation to resolve missing data[1] [3].

5. Best Practices

- Use strong prompt design and few-shot learning for better extraction accuracy[1].
- Supplement GenAI validation with traditional, code-based checks for critical flows[3].
- Protect against prompt injection or malicious input by sanitizing all user content and restricting what is passed to the model[4].
- Consider privacy and compliance if sending user data to external AI APIs.

6. Additional Resources

- Platforms like Google Vertex AI, Amazon Bedrock, and OpenAI's GPT API provide tools for prompt tuning, parameter extraction, and conversational validation flows[1] [5].
- Guardrail solutions and LLM-as-a-judge techniques can help with safe, transparent input and output validation in conversational agents[4] [6].

By combining generative AI models with robust validation logic, you can automate the extraction of required parameters from user messages, intelligently identify and request missing information, and maintain a seamless, human-friendly workflow for database management or similar applications.

⁂

Enabling Agent-to-Agent Invocation in Python

To allow your database management agent to be called from another agent, you need to implement a communication interface between agents. Modern multi-agent architectures commonly use standard protocols, messaging layers, or SDKs to facilitate such interactions.

Key Approaches

1. HTTP/API Endpoint

Expose your agent as a RESTful or WebSocket server (using frameworks like FastAPI or Flask). Other agents can then send requests (e.g., POST with query parameters) and receive responses. This approach is simple and framework-agnostic.

Example:

- Agent A exposes `/query` endpoint.
- Agent B sends requests (with parameters) and receives structured responses.

2. Agent Protocols (A2A, ACP)

Use a dedicated agent communication protocol, such as Google's Agent-to-Agent (A2A) Protocol or the Agent Communication Protocol (ACP). These protocols standardize message passing, event triggers, and enable richer orchestration features like context-sharing and event-driven workflows[7] [8] [9] [10].

Benefits:

- Built-in support for bidirectional, multi-agent orchestration.
- Shared session context and memory.
- Extensible with custom tools and event types.

Python Example using FastA2A (Pydantic AI):

```
from pydantic_ai import Agent

database_agent = Agent("your_underlying_logic")
app = database_agent.to_a2a()  # Exposes agent as an A2A server

# Another agent can now send requests to the database agent using the A2A protocol.
```

3. Library-Based Orchestration

Platforms like OpenAI Agents SDK, AutoGen, and LangChain allow families of agents to be chained, nested, or invoked as "tools" or external services[11] [12] [13].

- Agents can be added to a tool list or called programmatically as part of task pipelines.

- Some libraries allow direct reference to other agent objects, or invoke them through middleware abstractions for more powerful orchestrations[14].

4. Shared Context and Thread Management

If maintaining conversational context is necessary (for example, a multi-step user request or human-in-the-loop review), agents should pass a shared conversation/thread ID and/or shared state object with each call. This ensures that Agent B has the necessary previous context when called by Agent A[15].

Best Practices

- **Expose a clear callable interface** (REST endpoint, protocol handler, or Python class).
- **Document expected input/output** schemas for interoperability.
- **Manage session or context**, especially in chat-based, stateful, or multi-turn scenarios.
- **Handle authentication and permissions** as you would for an external service, especially for sensitive database operations.
- **Utilize standard protocols/APIs** for future-proofing and compatibility with orchestration frameworks.

Resources and Further Reading

- [Multi-Agent Communication with the A2A Python SDK][7]
- [Google Agent-to-Agent SDK Guide][8]
- [Pydantic FastA2A for Python Agents][10]
- [LangChain and OpenAI Agents SDK for chaining agents][12] [13]

By following these patterns, you can make your database management agent interoperable and callable from other agents, leveraging both simple API designs or advanced agent protocols for complex multi-agent ecosystems.

⁂

How the A2A Protocol Enables Seamless Agent-to-Agent Communication

The Agent2Agent (A2A) protocol is a modern, open standard designed specifically to allow AI agents—built on different frameworks, running in diverse environments—to communicate, collaborate, and coordinate in a secure and structured fashion. Here's how A2A achieves seamless agent-to-agent communication:

Key Principles and Architecture

1. Universal Interoperability

- **Open Standard:** A2A is built on widely adopted technologies such as HTTP(S), JSON-RPC 2.0, and Server-Sent Events (SSE), making it easy to integrate agents regardless of programming language or platform[16] [17] [18].
- **Framework-Agnostic:** Agents do not need to know each other's internal logic or implementation—interactions happen through common and standardized protocols[16] [18] [19].

2. Dynamic Discovery and Capability Sharing

- **Agent Card:** Every agent exposes a machine-readable "Agent Card" (typically a JSON file at a well-known endpoint, e.g., `/.well-known/agent.json`). This advertises the agent's identity, capabilities, supported formats, and authentication details, enabling other agents to discover and understand what tasks can be delegated[16] [18] [20].
- **Dynamic Task Orchestration:** Any agent can discover specialized agents for specific tasks and communicate directly, enabling efficient division of labor in complex multi-agent workflows[16] [17] [20].

3. Structured and Secure Message Exchange

- **Standard Message Format:** Communication involves structured messages containing headers (metadata), payloads (core content), authentication info, and routing instructions, allowing clarity and consistency[21] [19].
- **Authentication & Security:** Built-in enterprise-grade authentication (supporting API keys, JWT, OIDC, and OpenAPI schemes) ensures messages are exchanged only between authorized agents, safeguarding sensitive information[18] [22] [20].

4. Task-Centric Communication Model

- **Task Lifecycle:** Communication centers around clearly defined "Tasks," each with a lifecycle (e.g., submitted, working, completed, failed), tracked via unique IDs. Agents manage tasks asynchronously, supporting both short and long-running processes—crucial for real-world, multi-step flows and human-in-the-loop scenarios[16] [18] [19].
- **Event-Driven & Real-Time Updates:** Supports synchronous calls, streaming (using SSE for real-time updates), and asynchronous notifications (such as task completion callbacks or error alerts), allowing flexible coordination that matches business or user requirements[18] [21] [19].

5. Modality Agnosticism

- **Multi-Modal Collaboration:** Beyond simple text, A2A supports communication involving structured data (JSON), files, audio, video, and even negotiates user interface details and content types, supporting richer collaboration between agents[18] [19].

6. Privacy and Opacity

- **Opaque Execution:** Agents only expose their capabilities, accepting tasks or requests, but do not reveal proprietary internal state, code, memory, or tool details. This preserves privacy and intellectual property while enabling powerful multi-agent cooperation[23] [19] [20].

How a Typical A2A Interaction Works

Step	Action
Discovery	Agent A fetches Agent B's Agent Card to identify its capabilities and connection details
Initiation	Agent A sends a structured task/message to Agent B, using a unique task ID
Processing	Agent B processes the task, optionally sending real-time updates
Completion	Agent B signals task completion (success/failure) with final data or results

Benefits for Developers and Enterprises

- **Plug-and-Play Integration:** Standardized interfaces mean instant compatibility between agents from different providers or built for different domains[17] [20].
- **Scalability:** Teams can add, remove, or upgrade agents independently without rewriting integration code[20] [24].
- **Security and Compliance:** Fine-grained, enterprise authentication and data privacy aligned with modern security standards[22] [19].
- **Adaptability:** Suitable for instant responses or long-duration collaborative research, across cloud and on-premise setups[18] [19].

Example Use Cases

- **Automated Data Workflows:** A data extraction agent collaborates with an analysis agent, and then a reporting agent, all through A2A.
- **Cross-Domain Coordination:** Customer support agents, logistics agents, and billing agents from different vendors coordinate in real time to resolve support tickets.

A2A's standardized, secure, and discoverable approach is rapidly becoming foundational for orchestrating next-generation, interoperable AI systems where agents from any ecosystem can seamlessly work together[16] [17] [18] [20].

⁂

Security Features of the A2A Protocol for Safe Agent Interactions

The Agent2Agent (A2A) protocol is engineered with robust security features to ensure that agents can interact safely, protect sensitive data, and maintain trust between collaborating systems. Key security mechanisms incorporated into the A2A protocol include:

1. Enterprise-Grade Authentication

- **API Keys:** Agents can require API key verification for each request, ensuring only trusted parties gain access.
- **JWT (JSON Web Token):** Supports token-based authentication. JWTs are widely used for securely conveying user identity and claims between agents.
- **OIDC (OpenID Connect) and OpenAPI Schemes:** Integration with advanced authentication standards fits enterprise needs, including single sign-on and federated identity management.

2. Secure Communication Channels

- **HTTPS by Default:** All agent-to-agent communication is conducted over encrypted HTTPS connections, ensuring data confidentiality and preventing eavesdropping or man-in-the-middle attacks.
- **Server-Sent Events (SSE):** Real-time streaming and notifications use secure, authenticated connections.

3. Structured Message Validation

- **Strict Message Schemas:** All messages conform to standardized, machine-readable schemas (typically JSON or JSON-RPC 2.0), minimizing risks from malformed or injected content.
- **Payload Integrity Checks:** Incoming data is validated for correctness and completeness before processing.

4. Fine-Grained Authorization

- **Capability Scoping:** Each agent advertises only its permitted actions via a public Agent Card. This limits accidental or unauthorized invocation of sensitive tasks.
- **Role-Based Access Controls:** Agents can restrict operations based on client identity, role, or other policy-driven factors.

5. Task Isolation and Opaque Execution

- **Task Encapsulation:** Messages and results are scoped to individual, uniquely identified tasks, preventing cross-task data leakage.
- **Opaque Execution:** Agents never expose internal state or history, safeguarding intellectual property and preventing information disclosure.

6. Privacy and Compliance

- **Minimal Data Exposure:** Agents share only what is absolutely required for each interaction.
- **Audit and Logging:** Secure audit trails are supported, making it possible to track and review all cross-agent exchanges for compliance and incident response.

7. Protection Against Common Attacks

- **Rate Limiting and Throttling:** Mitigation against denial-of-service, brute-force, and resource exhaustion attacks.
- **Replay Protection:** Requests and responses may include nonces, timestamps, or unique task IDs to detect and block replay attempts.
- **Input Sanitization:** All incoming content is checked for injection and protocol abuse.

Summary Table

Security Feature	Description
Authentication	API Key, JWT, OIDC, OpenAPI standards
Secure Channels	Encrypted HTTPS, authenticated SSE
Message Validation	Schema checks, payload integrity
Fine-Grained Authorization	Capability scoping, roles, policy restrictions
Task Isolation/Opacity	Per-task encapsulation, no internal state exposure
Privacy & Compliance	Minimal required data shared, audit/support logging
Abuse/Attack Protection	Rate limiting, replay protection, input sanitization

These integrated security practices allow A2A-enabled agents to interact confidently across networks, organizations, and cloud environments, supporting trustworthy and compliant AI-driven workflows.

1. https://www.googlecloudcommunity.com/gc/Community-Blogs/Unlocking-the-Power-of-Gen-AI-A-Guide-to-Automating-Information/ba-p/673390
2. https://docs.uipath.com/document-understanding/automation-cloud/latest/user-guide/generative-features
3. https://www.blog.qualitypointtech.com/2025/03/leveraging-generative-ai-for-data.html?m=1
4. https://docs.aws.amazon.com/wellarchitected/latest/generative-ai-lens/gensec04-bp02.html
5. https://docs.automationanywhere.com/bundle/enterprise-v2019/page/data-extraction-with-gen-ai.html
6. https://news.mit.edu/2024/making-it-easier-verify-ai-models-responses-1021
7. https://towardsdatascience.com/multi-agent-communication-with-the-a2a-python-sdk/
8. https://dev.to/vivekyadav200988/google-agent-to-agent-a2a-communication-sdk-in-depth-guide-with-python-example-41gp
9. https://github.com/i-am-bee/acp
10. https://ai.pydantic.dev/a2a/

11. https://microsoft.github.io/autogen/0.2/docs/Use-Cases/agent_chat/
12. https://openai.github.io/openai-agents-python/running_agents/
13. https://python.langchain.com/docs/tutorials/agents/
14. https://github.com/strands-agents/sdk-python/issues/84
15. https://www.reddit.com/r/ChatGPTCoding/comments/1epzvxl/best_way_to_connect_different_ai_agents_together/
16. https://fractal.ai/blog/orchestrating-heterogeneous-and-distributed-multi-agent-systems-using-agent-to-agent-a2a-protocol
17. https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability/
18. https://a2aprotocol.ai
19. https://google.github.io/A2A/specification/
20. https://dev.to/_37bbf0c253c0b3edec531e/what-is-googles-a2a-g8m
21. https://dev.to/aniruddhaadak/understanding-the-a2a-protocol-a-beginners-guide-to-agent-to-agent-communication-4a91
22. https://www.solo.io/topics/ai-infrastructure/what-is-a2a
23. https://github.com/a2aproject/A2A
24. https://dev.to/sienna/googles-new-move-how-does-the-a2a-protocol-enable-ai-agents-to-add-friends-3iff