The OOCA Ticket Triage Agent is implemented using a layered architecture that separates data models, orchestration logic, tools, and prompts to ensure maintainability and safe execution. All inputs and outputs are defined using Pydantic schemas, enforcing strict runtime validation so that every triage result conforms to a known structure. The central orchestrator, TriageAgent, coordinates ticket processing by selecting between an AI-driven triage path and a deterministic rule-based fallback implemented in triage_logic.py, ensuring tickets are always processed even when external AI services are unavailable. The system follows a Model Context Protocol (MCP) style design in which all relevant context, ticket metadata, tool definitions, tool inputs and outputs, and system instructions are explicitly structured and passed to the model. This removes hidden state and constrains the LLM to reason only over auditable, schema-validated context. Tools are exposed through a shared BaseTool interface, allowing the agent to invoke knowledge-base search, customer history lookup, and region status checks without embedding tool-specific logic. New tools can be added by implementing the same interface without modifying the core agent. In addition to the runnable implementation, the repository includes skeleton code prepared for a future real MCP connection, defining interfaces and boundaries for standardised context exchange while deferring full transport and runtime integration.

For demonstration, the repository provides a fully runnable mock setup in which tools operate locally. This allows end-to-end triage flows, including tool invocation and structured output generation, to be exercised without live external systems. Configuration flags control tool enablement, making it easy to switch between mocked components and future real integrations. When an OpenAI API key is configured, the agent operates in AI mode using a structured system prompt that defines classification objectives, tool usage rules, and a required JSON output aligned with the TriageOutput schema. The model may request tool calls, which are executed by the agent and injected back into the model's context in an observe–act–respond loop consistent with MCP principles. All final outputs are validated before being returned. If the

OpenAI API is unavailable, the agent automatically falls back to rule-based triage, providing deterministic keyword-based classification while preserving availability.
Operational risks are mitigated through architectural constraints rather than implicit trust in the model. Misclassification risk is reduced through explicit prompt instructions, low-temperature configuration, and strict schema validation. Hallucinated information is prevented because the model cannot fabricate tool results; all external data must originate from explicit tool executions. Tool failures are handled defensively so triage can proceed with partial context while logging tool call traces. Escalation quality is managed through periodic human review and prompt or rule refinement. For production evaluation, the system should be assessed against labelled tickets to measure urgency accuracy, issue-type correctness, routing precision, sentiment detection, and knowledge-base relevance. Operational monitoring should track latency, tool failures, and invalid outputs. A human-in-the-loop process with regular sampling and mandatory review of critical or negative-sentiment tickets is recommended. Deployment should follow a phased rollout, progressing from shadow mode to assisted operation, then to selective automation, and ultimately to full automation with continuous auditing. Existing logging of tool calls and structured outputs provides a foundation that can be expanded as the system evolves toward fully integrated MCP-backed systems.