AgentRunner Class Methods - Comprehensive Notes

AgentRunner and Runner Class Relationship

The Runner class serves as a **static wrapper** around the AgentRunner class:

- Runner provides **class methods** that delegate to a global `DEFAULT_AGENT_RUNNER` instance
- AgentRunner contains the actual implementation
- This pattern allows users to call `Runner.run()` without creating an AgentRunner instance
- Global functions `set_default_agent_runner()` and `get_default_agent_runner()` manage the default instance

Usage Pattern:

```
# Using Runner (recommended)

result = await Runner.run(agent, input)

# Using AgentRunner directly

runner = AgentRunner()

result = await runner.run(agent, input)
```

Public Interface Methods

1. run() - Primary Async Execution Method

Purpose: Main method for executing agent workflows asynchronously

Key Features:

- Runs agents in a continuous loop until final output is generated
- Handles tool calls, agent handoffs, and guardrail validation
- Manages turn limits and error handling
- Returns a complete RunResult object

Process Flow:

1. Validates input and runs input guardrails
2. Executes agent turns in a loop
3. Processes tool calls and handoffs
4. Runs output guardrails on final result

5. Returns structured result with complete execution data

2. run_sync() - Synchronous Wrapper

Purpose: Provides synchronous interface to the async run() method

Implementation:

- Uses asyncio.get_event_loop().run_until_complete()
- Cannot be used in existing async contexts (Jupyter, FastAPI, etc.)
- Identical functionality to run() but blocks until completion

3. run_streamed() - Streaming Execution

Purpose: Returns a streaming result object for real-time event monitoring

Key Features:

- Starts agent execution in background task
- Returns RunResultStreaming object immediately
- Allows streaming of semantic events as they occur
- Non-blocking - execution continues asynchronously

Core Implementation Methods

4. _start_streaming() - Streaming Core Logic

Purpose: Core implementation for streaming agent execution

Responsibilities:

- Manages the main agent loop for streaming mode
- Handles trace management and span creation
- Coordinates input/output guardrail execution
- Manages event queue for streaming updates
- Handles agent handoffs and completion states

5. _run_single_turn() - Single Turn Execution

Purpose: Executes one complete turn of the agent loop

Process:

1. Runs agent start hooks if needed
2. Gets system prompt and configuration
3. Prepares input with conversation history
4. Calls model for response
5. Processes response and executes tools
6. Returns structured single step result

6. _run_single_turn_streamed() - Streaming Single Turn

Purpose: Streaming version of single turn execution

Key Differences:

- Streams model response events in real-time
- Updates streaming result object during execution
- Handles event queue management
- Processes final response after streaming completes

7. _get_single_step_result_from_response() - Response Processing

Purpose: Converts raw model response into structured execution result

Functions:

- Processes model output (text, tool calls, handoffs)
- Updates tool usage tracking
- Executes tools and handles side effects
- Creates SingleStepResult with next step information

Guardrail Management Methods

8. _run_input_guardrails() - Input Validation

Purpose: Executes and validates input guardrails before agent processing

Process:

- Runs all input guardrails concurrently
- Monitors for tripwire triggers
- Cancels remaining guardrails if tripwire triggered
- Raises InputGuardrailTripwireTriggered exception on violations
- Returns list of guardrail results

9. _run_output_guardrails() - Output Validation

Purpose: Validates final agent output against configured guardrails

Similar to input guardrails but:

- Runs after agent produces final output
- Uses agent output and context for validation
- Raises OutputGuardrailTripwireTriggered on violations

10. _run_input_guardrails_with_queue() - Streaming Input Validation

Purpose: Streaming version of input guardrail execution

Features:

- Pushes guardrail results to event queue as they complete
- Handles concurrent execution with result streaming
- Updates streamed result object with guardrail outcomes

Model Interaction Methods

11. _get_new_response() - Model Response Generation

Purpose: Handles actual model API calls with proper configuration

Responsibilities:

- Resolves model from agent/config settings
- Applies model settings and tool choice logic
- Makes API call with tracing support
- Updates usage tracking in context
- Returns structured ModelResponse

12. _get_output_schema() - Schema Resolution

Purpose: Extracts and validates agent output schema configuration

Logic:

- Returns None for string output types
- Returns existing AgentOutputSchemaBase instances
- Wraps other types in AgentOutputSchema

Agent Configuration Methods

13. _get_handoffs() - Handoff Resolution

Purpose: Resolves available handoffs for current agent

Process:

1. Collects handoff definitions from agent
2. Converts Agent objects to Handoff objects
3. Evaluates is_enabled conditions asynchronously
4. Returns only enabled handoffs

14. _get_all_tools() - Tool Collection

Purpose: Gathers all available tools for agent execution

Implementation:

- Delegates to agent.get_all_tools(context_wrapper)
- Returns complete list of tools available to agent

15. _get_model() - Model Resolution

Purpose: Resolves the model to use based on configuration hierarchy

---

Priority Order:

1. run_config.model (if Model instance)
2. run_config.model (if string, resolved via provider)
3. agent.model (if Model instance)
4. agent.model (resolved via provider)

Class Architecture Notes

Design Patterns:

- **Facade Pattern:** Runner class provides simplified interface
- **Template Method:** Core execution flow with customizable steps
- **Strategy Pattern:** Configurable models, tools, and guardrails

Error Handling:

- Comprehensive exception handling with structured error data
- Graceful degradation with proper cleanup
- Detailed tracing for debugging

Concurrency:

- Heavy use of asyncio for concurrent operations
- Proper task cancellation and cleanup
- Event-driven streaming architecture