**[Suna.so](http://suna.so/) Java Agent Technical Documentation**

This documentation provides in-depth technical details about the [Suna.so](http://suna.so/) Java Agent implementation, focusing on architecture, request flows, and key components.

**Table of Contents**

1. [Architecture Overview](https://file+.vscode-resource.vscode-cdn.net/Users/ayousri/projects/javanativestuff/2025-may/suna-ai-java/docs/architecture-overview.md)
2. [Request Flow End-to-End](https://file+.vscode-resource.vscode-cdn.net/Users/ayousri/projects/javanativestuff/2025-may/suna-ai-java/docs/request-flow.md)
3. [Agent Execution Loop](https://file+.vscode-resource.vscode-cdn.net/Users/ayousri/projects/javanativestuff/2025-may/suna-ai-java/docs/agent-execution-loop.md)
4. [Sandbox and Tool Architecture](https://file+.vscode-resource.vscode-cdn.net/Users/ayousri/projects/javanativestuff/2025-may/suna-ai-java/docs/sandbox-tool-architecture.md)
5. [Billing, Accounts, Projects, and Subscriptions](https://file+.vscode-resource.vscode-cdn.net/Users/ayousri/projects/javanativestuff/2025-may/suna-ai-java/docs/billing-accounts-projects.md)
6. [Context Management and Token Handling](https://file+.vscode-resource.vscode-cdn.net/Users/ayousri/projects/javanativestuff/2025-may/suna-ai-java/docs/context-management.md)
7. [LLM Integration](https://file+.vscode-resource.vscode-cdn.net/Users/ayousri/projects/javanativestuff/2025-may/suna-ai-java/docs/llm-integration.md)
8. [Streaming and Real-time Updates](https://file+.vscode-resource.vscode-cdn.net/Users/ayousri/projects/javanativestuff/2025-may/suna-ai-java/docs/streaming.md)
9. [Authentication and Authorization](https://file+.vscode-resource.vscode-cdn.net/Users/ayousri/projects/javanativestuff/2025-may/suna-ai-java/docs/auth.md)
10. [Database Schema and Access Patterns](https://file+.vscode-resource.vscode-cdn.net/Users/ayousri/projects/javanativestuff/2025-may/suna-ai-java/docs/database.md)
11. [Redis Usage for Pub/Sub and Caching](https://file+.vscode-resource.vscode-cdn.net/Users/ayousri/projects/javanativestuff/2025-may/suna-ai-java/docs/redis-usage.md)

**System Glossary**

* **Agent**: An autonomous AI entity capable of executing tasks using tools and LLM capabilities
* **Thread**: A sequence of messages, tool calls, and results that represent a conversation
* **Project**: A container for related threads and resources
* **Sandbox**: An isolated execution environment where agent code and tools run securely
* **Tool**: A function with well-defined inputs and outputs that agents can use to interact with external systems
* **LLM**: Large Language Model, the foundation of agent intelligence
* **Context Window**: The amount of text an LLM can consider at once
* **Context Management**: Techniques to summarize and maintain relevant information within the context window
* **Tool Registry**: A service that manages available tools and their schemas

**Request Flow End-to-End**

This document provides a detailed technical walkthrough of how a request flows through the [Suna.so](http://suna.so/) Java Agent system, from initial client request to final response. This covers the entire lifecycle including authentication, execution, streaming, and all the underlying technical implementations.

**1. Client Request Initiation**

**1.1 HTTP Request Entry Point**

The client initiates a request to start an agent run by sending a POST request to the /api/agent/runs endpoint handled by AgentController.startAgent(). The request includes:

* **Authorization Header**: JWT token for authentication
* **Request Body**: AgentRunRequest JSON payload with agent configuration
* **Optional File Uploads**: Files to be made available in the agent's sandbox

**Example Request:**

* POST to /api/agent/runs
* Headers include Authorization (JWT token) and Content-Type
* Request body contains model specifications, initial prompt, and configuration
* May include file uploads to be made available in the agent's sandbox

**2. Authentication & Authorization Flow**

**2.1 JWT Token Validation**

JwtUtils.getUserIdFromToken() parses and validates the JWT token:

1. Extracts the token from the Authorization header
2. Validates signature, expiry, and issuer claims
3. Returns the userId from the JWT payload
4. Throws ResponseStatusException with appropriate status codes for failures

**2.2 Authorization Checks**

After validating the token, AuthUtils verifies permissions:

1. AuthUtils.verifyProjectAccess() checks if the user has access to the project
2. AuthUtils.isAccountAdmin() verifies admin-level operations
3. Access checks query the database with RLS policies through DBConnection

**3. Orchestration & Resource Initialization**

**3.1 Account & Project Resolution**

AgentController.startAgent() orchestrates the initial setup:

1. accountService.getOrCreateAccountForUser(userId) retrieves or creates the user's account asynchronously
2. projectService.getOrCreateProjectForAccount(accountId, userId) retrieves or creates a project for this account asynchronously
3. Both operations involve database transactions via DBConnection and return CompletableFuture<> results that must be joined

**3.2 Sandbox Provisioning**

SandboxService.createSandbox() provisions an isolated environment:

1. Calls the Daytona API via HttpClientService to create a workspace
2. Returns DaytonaWorkspace with sandboxId and other metadata
3. Updates the project record with the sandbox details via projectService.updateProjectSandboxId()

The controller then calls the SandboxService to create a sandbox, retrieves the sandbox ID, and updates the project record with this information.

**3.3 Thread Creation**

ThreadManager.createThread() initializes a new conversation thread:

1. Generates a UUID for the thread
2. Inserts thread metadata in the database with project and account associations
3. Returns a Thread object with the new ID

**3.4 File Upload Handling**

For multipart requests containing files:

1. Files are extracted from the MultipartFile[] array
2. Each file is uploaded to the sandbox via sandboxFileService.uploadFile(sandboxId, path, bytes).join()
3. File paths in the sandbox are tracked and appended to the initial prompt

**4. Agent Run Submission**

**4.1 Initial Message Creation**

ThreadManager.addMessage() adds the user's initial prompt to the thread:

1. Creates a new message with type "user" in the database
2. The message includes the initial prompt text plus any uploaded file information
3. Returns the created Message object with its ID

**4.2 Agent Run Record Creation**

AgentBackgroundService.submitAgentRun() initiates the agent execution:

1. Generates a unique agentRunId
2. Creates an entry in the agent\_runs table with status "PENDING"
3. Adds the run to a queue for asynchronous execution
4. Returns immediately while processing continues in the background

**5. Streaming Response Initiation**

**5.1 Server-Sent Events (SSE) Setup**

Clients open an SSE connection to /api/agent/runs/{agentRunId}/stream:

1. AgentController.streamAgentRun() creates a new SseEmitter with a long timeout
2. The emitter is registered with AgentRedisHelper.subscribeToRunStream(agentRunId, messageHandler)
3. Connection lifecycle events (completion, timeout, error) are handled with appropriate cleanup

The controller creates an SSE emitter with a long timeout and registers it with the AgentRedisHelper to subscribe to the run stream. When messages are received, they're sent through the emitter to the client, with appropriate error handling for connection lifecycle events.

**6. Agent Execution Loop**

**6.1 Background Processing**

AgentRunnerService.executeAgentRun() is the core agent execution function:

1. Retrieves the agent configuration and resources (project, thread, model)
2. Registers tools via AgentRunManager.registerAgentTools()
3. Enters a loop that continues until completion or interruption

**6.2 Tool Registration**

ToolRegistry dynamically registers available tools:

1. ToolRegistry.registerTool() registers each tool with its schema
2. XML and OpenAPI schemas are extracted and stored
3. Tools are organized by name and mapping to java methods via reflection
4. Returns a complete registry of available tools for the agent

**6.3 The Main Agent Loop**

Each iteration of the agent loop follows this process:

Each iteration of the agent loop follows this process:

1. Check for [todo.md](http://todo.md/) updates and other control signals
2. Manage context window using summarization if needed
3. Construct the message for this iteration
4. Execute one turn of LLM reasoning via AgentExecutionHelper
5. Process results, detect completion or errors
6. Stream results to client via Redis

**7. LLM Interaction**

**7.1 Conversation and Tool Execution Flow**

AgentExecutionHelper.executeIteration() handles each LLM turn:

1. Constructs the LLM prompt with messages, system instructions, and tools
2. Calls ThreadManager.runThread() which orchestrates the LLM call and tool execution
3. Receives, processes, and potentially streams responses

**7.2 LLM API Call**

OpenAILlmService.makeLlmApiCall() handles communication with the LLM API:

1. Formats the request body with messages, tools, and parameters
2. Makes an HTTP request to the LLM provider (OpenAI) via OkHttp
3. For streaming responses, processes chunks as they arrive
4. Records usage and costs via BillingServiceFacade

The OpenAILlmService formats the request body with messages, tools, and parameters, then makes an HTTP request to the LLM provider. For streaming responses, it processes chunks as they arrive, while for standard responses, it parses the complete response and records usage costs.

**7.3 Response Processing**

ResponseProcessor parses the LLM responses and tool calls:

1. For streaming responses, accumulates text and detects tool calls incrementally
2. For tool calls, extracts parameters using XML or JSON parsing
3. Handles multi-modal content and specialized formats

**8. Tool Execution**

**8.1 Tool Call Processing**

When a tool call is detected:

1. ToolRegistry.getTool() or getXmlTool() retrieves the tool by name
2. ToolRegistry.invokeMethod() calls the appropriate method with arguments
3. Arguments are converted from Maps to proper types using Jackson's ObjectMapper
4. Results are converted back to JSON for the LLM's consumption

**8.2 Sandbox Tool Execution**

Tools that interact with the sandbox environment follow a pattern:

1. SandboxToolBase provides common functionality to all sandbox tools
2. SandboxToolBase.getSandboxFromProject() retrieves sandbox credentials
3. Tool-specific functions make HTTP calls to the Daytona API
4. Results are formatted for LLM consumption

Tools that interact with the sandbox environment follow a common pattern:

1. They retrieve sandbox credentials via SandboxToolBase
2. Make API calls to the Daytona service through specific service classes
3. Handle successful responses and errors appropriately
4. Format results for LLM consumption

**9. Real-time Updates and Streaming**

**9.1 Redis Streaming**

AgentRedisHelper handles real-time streaming to clients:

1. publishMessageToRedisStream() publishes messages to Redis channels
2. Redis Pub/Sub enables real-time transmission of agent messages
3. Redis Streams enable persisted, ordered message delivery

The AgentRedisHelper publishes messages to Redis channels and persists them in Redis streams, enabling real-time transmission of agent messages to subscribed clients.

**9.2 Stream Handling**

For streaming responses from the LLM:

1. OpenAILlmService processes the streamed chunks incrementally
2. ResponseProcessor.processStreamingResponse() accumulates tokens and detects tool calls
3. Partial updates are published to Redis for real-time client updates

**10. Completion and Cleanup**

**10.1 Detecting Completion**

AgentRunManager.isAgentTaskComplete() determines when the agent has finished:

1. Checks for explicit completion signals in messages
2. Monitors for timeouts or maximum iterations
3. Detects terminal error conditions

**10.2 Finalizing the Run**

When the agent completes:

1. AgentBackgroundService.updateAgentRunStatus() updates the status to "COMPLETED"
2. Any final responses are sent to the client via Redis
3. Resources are marked for cleanup or preservation as needed

**10.3 Error Handling**

If errors occur during execution:

1. Exceptions are caught and logged at appropriate levels
2. AgentBackgroundService.updateAgentRunStatus() updates status to "FAILED" with error details
3. Error messages are streamed to the client
4. Resources are cleaned up appropriately

**11. Database Interaction Pattern**

Throughout the entire flow, the application uses an asynchronous database interaction pattern:

Throughout the entire flow, the application uses an asynchronous database interaction pattern, leveraging CompletableFuture with thenApply, thenCompose, and exceptionally methods to manage complex async flows.

**12. Complete Request-Response Cycle Diagram**

Client AgentController AgentRunnerService OpenAILlmService Tools

| | | | |

|-- POST /api/agent/runs --------------->| | | |

| | | | |

| |-- Auth & validate ---------------+| | |

| |-- Create account/project --------+| | |

| |-- Provision sandbox -------------+| | |

| |-- Create thread -----------------+| | |

| |-- Upload files ------------------->| | |

| | | | |

|<-- 200 OK (agentRunId, threadId) ------| | | |

| | | | |

|-- GET /runs/{id}/stream -------------->| | | |

| |-- Setup SSE emitter ------------->| | |

| | | | |

| |-- submitAgentRun --------------->+| | |

| | |-- executeAgentRun() ------------->| |

| | |-- Register tools ---------------->| |

| | | | |

| | |-- Execute iteration ------------->| |

| | | |-- LLM API call ----------------->|

| | | |<-- LLM Response -----------------|

| | | | |

| | | |-- Parse tool calls ------------->|

| | | | |-- Execute tool call ----+

| | | | |<- Tool result ----------+

| | | |<-- Tool execution result --------|

| | |<-- AgentLoopResult ---------------| |

| | | | |

| | |-- Publish to Redis ------------->+| |

|<-- SSE Event (message) ---------------| | | |

| | | | |

| | |-- [Repeat loop until complete] -->| |

| | | | |

|<-- SSE Event (completion) ------------| | | |

| | | | |

+----------------------------------------+-----------------------------------+-----------------------------------+----------------------------------+

**13. Technical Implementation Patterns**

**13.1 Asynchronous Design**

The system uses CompletableFuture extensively for non-blocking operations:

1. Database queries return CompletableFuture<List<Map<String, Object>>>
2. External API calls are wrapped in CompletableFuture<T>
3. Composition patterns like thenApply(), thenCompose(), and exceptionally() manage complex async flows
4. .join() is used strategically when synchronous results are required

**13.2 Dependency Injection**

Spring's dependency injection is used throughout:

1. Controllers, services, and helpers are injected via constructor injection
2. Configuration properties are injected with @Value annotations
3. Tools receive service dependencies at registration time

**13.3 Streaming Pattern**

The streaming implementation follows a reactive pattern:

1. LLM responses are processed as they arrive
2. Redis pub/sub enables real-time updates
3. SSE emitters provide a push mechanism to clients

**13.4 Error Handling**

Robust error handling is implemented at multiple levels:

1. Global exception handlers for REST endpoints
2. CompletableFuture.exceptionally() for async error handling
3. Fallback mechanisms for degraded but functional operation
4. Detailed logging for observability

**Sandbox and Tool Architecture**

This document provides an in-depth technical explanation of the Sandbox and Tool architecture in the [Suna.so](http://suna.so/) Java Agent. It covers how sandboxes are provisioned, how tools are implemented and executed, and how the agent interacts with external systems.

**1. Sandbox Architecture Overview**

**1.1 Core Components**

The sandbox system consists of several key components:

1. **SandboxService**: Core service for sandbox operations and management
2. **SandboxFileService**: Handles file operations within sandboxes
3. **SandboxToolBase**: Base class for all sandbox tools
4. **HttpClientService**: Handles HTTP communications with the Daytona API
5. **WorkspaceService**: Manages workspace resources within sandboxes
6. **ProcessService**: Handles command execution within sandboxes

**1.2 High-Level Architecture**

┌───────────────────┐ ┌─────────────────┐ ┌────────────────────┐

│ │ │ │ │ │

│ AgentController │────▶│ SandboxService │────▶│ HttpClientService │──┐

│ │ │ │ │ │ │

└───────────────────┘ └─────────────────┘ └────────────────────┘ │

│ │

▼ │

┌───────────────────┐ ┌─────────────────┐ │

│ │ │ │ │

│ SandboxToolBase │◀────│ Tool Registry │ │

│ │ │ │ │

└───────────────────┘ └─────────────────┘ │

│ │

▼ ▼

┌────────────────────────────────┐ ┌────────────────────┐

│ │ │ │

│ FileTool ProcessTool Etc. │────────────────────────▶│ Daytona API │

│ │ │ │

└────────────────────────────────┘ └────────────────────┘

**Sandbox and Tool Architecture**

This document provides an in-depth technical explanation of the Sandbox and Tool architecture in the [Suna.so](http://suna.so/) Java Agent. It covers how sandboxes are provisioned, how tools are implemented and executed, and how the agent interacts with external systems.

**1. Sandbox Architecture Overview**

**1.1 Core Components**

The sandbox system consists of several key components:

1. **SandboxService**: Core service for sandbox operations and management
2. **SandboxFileService**: Handles file operations within sandboxes
3. **SandboxToolBase**: Base class for all sandbox tools
4. **HttpClientService**: Handles HTTP communications with the Daytona API
5. **WorkspaceService**: Manages workspace resources within sandboxes
6. **ProcessService**: Handles command execution within sandboxes

**1.2 High-Level Architecture**

┌───────────────────┐ ┌─────────────────┐ ┌────────────────────┐

│ │ │ │ │ │

│ AgentController │────▶│ SandboxService │────▶│ HttpClientService │──┐

│ │ │ │ │ │ │

└───────────────────┘ └─────────────────┘ └────────────────────┘ │

│ │

▼ │

┌───────────────────┐ ┌─────────────────┐ │

│ │ │ │ │

│ SandboxToolBase │◀────│ Tool Registry │ │

│ │ │ │ │

└───────────────────┘ └─────────────────┘ │

│ │

▼ ▼

┌────────────────────────────────┐ ┌────────────────────┐

│ │ │ │

│ FileTool ProcessTool Etc. │────────────────────────▶│ Daytona API │

│ │ │ │

└────────────────────────────────┘ └────────────────────┘

**2. Sandbox Service Implementation**

**2.1 Core SandboxService**

The SandboxService handles the lifecycle of sandboxes:

* **Dependencies**: Requires HttpClientService, DBConnection, and ObjectMapper
* **Configuration**: Uses Daytona API URL and API key from application properties
* **Key Methods**:
  + createSandbox: Creates a new sandbox for a project
  + getSandbox: Retrieves sandbox details by ID
  + executeCommand: Runs a command in a sandbox
  + deleteSandbox: Removes a sandbox when no longer needed

This service communicates with the Daytona API to provision and manage sandbox environments. When creating a sandbox, it stores the sandbox information in the project record in the database, ensuring the sandbox can be accessed later by tools and other services.

**2.2 SandboxFileService Implementation**

The SandboxFileService handles file operations within sandboxes:

* **Dependencies**: Requires HttpClientService and ObjectMapper
* **Configuration**: Uses Daytona API URL and API key from application properties
* **Key Methods**:
  + readFile: Reads file content from a sandbox
  + writeFile: Writes content to a file in a sandbox
  + listFiles: Lists files in a sandbox directory
  + deleteFile: Removes a file from a sandbox

This service provides file system operations for the agent, allowing it to read, write, and manipulate files within the sandbox environment. It handles path encoding and proper error handling for all file operations.

**3. Tool Architecture**

**3.1 SandboxToolBase Class**

All sandbox tools extend SandboxToolBase, which provides common functionality:

* **Properties**:
  + projectId: The ID of the project the tool operates on
  + Service dependencies: SandboxService, SandboxFileService, etc.
* **Key Methods**:
  + Setter methods for dependency injection
  + getSandboxFromProject: Retrieves sandbox details from project record

The SandboxToolBase class handles common operations like retrieving sandbox credentials and managing service dependencies. It defines a SandboxDetails inner class to encapsulate sandbox ID, password, and organization ID.

When retrieving sandbox details, it attempts to get the information from the project record in the database. If not found, it falls back to a default sandbox ID based on the project ID.

**3.2 FileTool Implementation**

The FileTool provides file system operations to the agent:

* **Methods**:
  + readFile: Reads content from a file
  + writeFile: Writes content to a file
  + listFiles: Lists files in a directory
  + deleteFile: Deletes a file
  + createDirectory: Creates a directory
  + searchFiles: Searches for files matching a pattern
* **Schema Definition**:
  + Provides OpenAPI schemas for each method
  + Defines required parameters and their types
  + Includes descriptions for methods and parameters

The FileTool follows a consistent pattern for all operations:

1. Retrieves sandbox details using getSandboxFromProject
2. Calls the appropriate method on SandboxFileService
3. Handles success and error cases with proper logging
4. Returns formatted results or error messages

For operations not directly supported by the SandboxFileService (like creating directories or searching), it uses executeCommand from the SandboxService to run shell commands.

**3.3 ProcessTool Implementation**

The ProcessTool allows the agent to execute commands:

* **Methods**:
  + executeCommand: Executes a command in the sandbox
  + Overloaded version with working directory parameter
  + startLongRunningProcess: Starts a background process
* **Schema Definition**:
  + Provides OpenAPI schemas for each method
  + Defines command parameters and optional working directory
  + Includes descriptions for different execution modes

The ProcessTool allows the agent to run shell commands and processes within the sandbox. It handles command execution, captures output and exit codes, and formats the results for the agent to interpret.

Long-running processes are managed through Daytona's process session API, allowing the agent to start services or background tasks that continue running even after the command completes.

**3.4 ExposeTool Implementation**

The ExposeTool provides port exposure and URL generation:

* **Methods**:
  + exposePort: Makes a sandbox port accessible externally
  + getPreviewUrl: Generates a URL for accessing exposed ports
* **Schema Definition**:
  + Defines port parameters and exposure options
  + Specifies return value formats for URLs

This tool allows the agent to expose services running in the sandbox (like web servers) to the outside world, and generate URLs that can be used to access these services. It's essential for demonstrating web applications or services created by the agent.

**3.5 WorkspaceTool Implementation**

The WorkspaceTool manages workspace operations:

* **Methods**:
  + createWorkspace: Creates a new workspace
  + cloneRepository: Clones a Git repository into the workspace
  + installDependencies: Installs project dependencies
* **Schema Definition**:
  + Parameters for workspace creation and repository operations
  + Options for dependency management

This tool handles higher-level operations related to workspace setup and management, allowing the agent to create development environments, clone code, and prepare projects for use.

**4. Tool Registration and Discovery**

**4.1 ToolRegistry Integration**

Tools are registered with the ToolRegistry at runtime:

* Registration occurs in AgentRunnerService.executeAgentRun
* Each tool is instantiated with the project ID
* Required services are injected after instantiation
* Tools are registered with the registry using registerTool

**4.2 Schema Extraction**

Tool schemas are extracted and made available to the LLM:

* Each tool implements getSchemas() to define its capabilities
* Schemas can be OpenAPI or XML format
* The ToolRegistry collects all schemas during registration
* getOpenApiSchemas() and getXmlSchemas() provide schemas to the LLM

**4.3 Method Invocation**

When the LLM calls a tool, the ToolRegistry handles method invocation:

* Tool method is located by name
* Arguments are converted to expected types
* Method is invoked via reflection
* Results are converted back to string or serializable format
* Error handling ensures robust execution

**5. Sandbox Environment Management**

**5.1 Sandbox Lifecycle**

Sandboxes follow a defined lifecycle:

1. **Creation**: Initiated by AgentController.startAgent()
2. **Configuration**: Files uploaded, environment variables set
3. **Active Phase**: Used by tools for agent operations
4. **Cleanup**: Eventual deletion by cleanup jobs

**5.2 Resource Management**

The sandbox system includes resource management:

* File storage with quotas and cleanup
* Process monitoring and termination
* Port allocation and release
* Memory and CPU limits

**5.3 Security Considerations**

Security is handled through several mechanisms:

* Isolated execution environments
* Access control through API keys
* Resource limits to prevent abuse
* Network isolation for sensitive operations

**6. Integration with Daytona**

**6.1 Daytona API Interface**

The system interfaces with Daytona through its HTTP API:

* Workspace management (create, delete)
* File operations (read, write, list)
* Command execution and process management
* Port exposure and URL generation

**6.2 Data Models**

Key data models for Daytona integration:

* **DaytonaWorkspace**: Represents a sandbox environment
* **DaytonaCommandExecution**: Result of command execution
* **DaytonaProcessSession**: Long-running process handle
* **FileInfo**: Metadata about files in the sandbox

**6.3 Error Handling**

The integration includes robust error handling:

* Network error detection and logging
* API error status interpretation
* Fallback mechanisms for transient failures
* Detailed error reporting for diagnostics

**Agent Execution Loop**

This document provides an in-depth technical breakdown of the Agent Execution Loop, which is the core processing engine of the [Suna.so](http://suna.so/) Java Agent. It details how the agent processes requests, interacts with the LLM, executes tools, and manages its execution flow.

**1. Agent Loop Architecture**

**1.1 Core Components**

The agent execution loop involves several key components working together:

1. **AgentRunnerService**: Orchestrates the overall execution flow
2. **AgentExecutionHelper**: Handles individual iteration steps
3. **ThreadManager**: Manages message history and LLM interactions
4. **ContextManager**: Handles context window management and summarization
5. **ToolRegistry**: Provides tool discovery and execution capabilities
6. **ResponseProcessor**: Parses LLM responses and extracts tool calls
7. **LlmService**: Communicates with the language model provider

**1.2 High-Level Flow Diagram**

┌─────────────────────────────────────────┐

│ AgentRunnerService │

│ │

│ ┌─────────┐ ┌──────────┐ ┌──────────┐ │

│ │ Init & │ │ Process │ │ Finalize │ │

│ │ Setup ├─►│ Iterations├─►│ Result │ │

│ └─────────┘ └──────────┘ └──────────┘ │

└───────────────┬─────────────────────────┘

│

▼

┌───────────────────────────────────────────────────────────┐

│ AgentExecutionHelper │

│ │

│ ┌─────────────┐ ┌───────────┐ ┌───────────────────┐ │

│ │ Construct │ │ Execute │ │ Process Results │ │

│ │ Messages ├─►│ LLM Call ├─►│ & Tool Execution │ │

│ └─────────────┘ └───────────┘ └───────────────────┘ │

└───────────────────────────────────────────────────────────┘

**2. AgentRunnerService - The Main Loop**

**2.1 Loop Initialization**

The entry point for agent execution initializes key variables and resources, registers tools for the agent run, and starts the execution in a separate thread using CompletableFuture.runAsync(). It handles initialization of the user ID, system prompt, model resolution, and timestamp tracking. It also includes error handling to update the agent run status and log any exceptions.

**2.2 Tool Registration Process**

Tool registration creates and configures various tools the agent can use, including:

* Sandbox tools like FileTool for file operations
* ProcessTool for command execution
* DataProviderTool for external API access
* Other specialized tools as needed

Each tool is instantiated with the project ID, configured with appropriate services, registered with the tool registry, and added to the list of available tools.

**2.3 The Main Execution Loop**

The main execution loop runs until the agent has completed its task or is interrupted. During each iteration, it:

1. Checks for interruption signals
2. Checks for updates to the [todo.md](http://todo.md/) file
3. Manages the context window, performing summarization if needed
4. Constructs a temporary message for this iteration
5. Executes one turn of LLM reasoning via the AgentExecutionHelper
6. Processes the results, adding new messages to the collection
7. Streams results to the client via Redis
8. Updates the [todo.md](http://todo.md/) file if necessary
9. Checks for completion signals, errors, or continuation flags

Once the loop exits, it updates the final status of the agent run to either STOPPED or COMPLETED.

**3. Agent Execution Helper - Single Iteration Processing**

**3.1 Iteration Structure**

Each iteration of the agent loop is handled by the AgentExecutionHelper.executeIteration() method, which:

1. Retrieves tool schemas and XML examples from the tool registry
2. Configures the processor with settings for native tool calling and XML tool calling
3. Calls ThreadManager.runThread() to execute the iteration with the LLM
4. Processes the resulting messages to check for completion signals or errors
5. Wraps the results in an AgentLoopResult object with messages, termination status, and continuation flags
6. Includes error handling to create appropriate error messages if something goes wrong

The method returns a CompletableFuture that resolves to the AgentLoopResult, allowing for asynchronous execution.

**3.2 Thread Manager Execution**

The ThreadManager.runThread() method is at the heart of each iteration, and performs these key steps:

1. Retrieves the formatted thread messages for context
2. Creates a system message with the provided prompt
3. Adds a user message if there's user input for this iteration
4. Prepends the system message to the conversation history
5. Makes the LLM API call with the prepared messages, model, and configuration
6. Processes the LLM response, extracting content and tool calls
7. Saves all new messages to the thread in the database
8. Returns a CompletableFuture that resolves to the list of messages

This method uses CompletableFuture composition with thenCompose() to chain asynchronous operations together in a clean, readable way.

**4. Tool Execution Flow**

**4.1 Detecting and Parsing Tool Calls**

The ResponseProcessor.processResponse() method handles the LLM response and executes any tool calls:

1. Creates an assistant message with the LLM's response content
2. Processes any tool calls included in the response:
   * Extracts the tool name and arguments
   * Looks up the tool in the registry
   * Executes the tool with the provided arguments
   * Creates a tool result message with the output
   * Handles errors if the tool is not found or execution fails
3. Also processes XML-formatted tool calls if enabled
4. Returns a CompletableFuture with all the resulting messages

This method is responsible for the critical step of converting LLM tool calls into actual tool executions and capturing the results.

**4.2 XML Tool Call Parsing**

XML tool calls are parsed using specialized methods:

1. The extractXmlToolCalls method:
   * Uses regex to find all XML tags in the LLM's response content
   * Extracts the tool name and full XML content for each match
   * Parses the XML to extract parameters
   * Creates a ToolCall object with the extracted information
   * Returns a list of all identified tool calls
2. The parseXmlToolCall method:
   * Extracts the root element name
   * Looks up the XML schema for the tool
   * For each parameter mapping in the schema:
     + Gets the parameter name, node type, and path
     + Extracts the parameter value from the XML content
     + Adds the parameter to the result map
   * Returns a map of parameter names to values

These methods allow the agent to process XML-formatted tool calls, which provide a more structured way for LLMs to specify tool invocations.

**5. Context Management**

**5.1 Token Counting and Context Window Management**

The ContextManager.checkAndSummarizeIfNeeded method manages the context window:

1. Gets the current token count for the thread
2. If the token count is below the threshold, no summarization is needed
3. If summarization is needed:
   * Gets the messages that need to be summarized
   * Creates a summary using the LLM
   * Adds the summary message to the thread
   * Returns true if summarization was performed
4. Includes error handling to log issues and return false if summarization fails

This method ensures that the conversation history stays within the LLM's context window limits by periodically creating summaries of older messages.

**5.2 Summarization Process**

The summarization process creates a concise version of the conversation:

1. Creates a system message with detailed summarization instructions:
   * The summary must preserve key information and decisions
   * Include tool usage and results
   * Maintain chronological order
   * Be structured with clear sections
   * Include only factual information from the conversation
   * Be concise but detailed enough for continuation
2. Makes an LLM API call to generate the summary
   * Uses a special prompt format with clear markers
   * Sets a target token limit for the summary
   * Configures the LLM for summarization (low temperature)
3. Formats the resulting summary with clear delimiters
   * Adds header and footer markers
   * Includes a transition note for context

This approach uses the LLM's own capabilities to create high-quality summaries that preserve the important context while reducing token usage.

**6. Streaming Response Handling**

**6.1 Streaming Implementation**

For streaming responses, the flow changes to process chunks incrementally:

1. The processStreamingResponse method creates a Flux of messages:
   * Maintains accumulators for content and tool call parameters
   * Processes each chunk as it arrives from the LLM
   * Emits message updates as content and tool calls are received
   * Detects when tool calls are complete and ready for execution
   * Executes tools when all parameters have been received
   * Handles errors and completes the stream appropriately
2. For content updates:
   * Detects if this is the first chunk or a continuation
   * Creates appropriate message objects with update flags
   * Accumulates content for later XML tool call detection
3. For tool call updates:
   * Tracks each tool call by ID and index
   * Accumulates tool name and arguments as they arrive
   * Marks tool calls as complete when finished
   * Executes tools when all parameters are available
4. For XML tool calls:
   * Periodically checks accumulated content for XML patterns
   * Extracts and processes any complete XML tool calls
   * Marks them as ready for execution

This streaming approach allows for real-time updates to the client, showing the agent's "thoughts" and tool progress incrementally.

**7. Todo File Handling**

[Suna.so](http://suna.so/) agents maintain a todo.md file to track progress:

1. The updateTodoFileFromMessage method:
   * Checks for Markdown code blocks in assistant messages
   * Extracts [todo.md](http://todo.md/) content using regex patterns
   * Writes the updated content to the [todo.md](http://todo.md/) file in the sandbox
   * Returns success/failure status
2. The [todo.md](http://todo.md/) file serves as:
   * A record of completed tasks
   * A list of planned tasks
   * A communication channel between agent iterations
   * A way for users to track progress

This file-based approach provides a persistent record of the agent's work plan and progress across iterations, which is useful for both the agent and human users.

**LLM Integration**

This document provides a detailed technical overview of how the [Suna.so](http://suna.so/) Java Agent integrates with Large Language Models (LLMs). It covers the service architecture, request/response handling, streaming implementation, model configuration, and billing aspects.

**1. LLM Service Architecture**

**1.1 Core Components**

The LLM integration is built around several key components:

1. **LlmService Interface**: A common interface that defines the contract for LLM interactions
2. **OpenAILlmService**: The primary implementation that communicates with OpenAI's API
3. **LlmServiceFactory**: Factory that creates appropriate LLM service implementations
4. **LlmServiceAdapter**: Adapter pattern implementation that normalizes various LLM provider APIs
5. **ResponseProcessor**: Processes raw LLM responses into usable format with tool calls
6. **BillingServiceFacade**: Tracks token usage and costs for billing purposes

**1.2 High-Level Architecture**

┌───────────────────┐ ┌─────────────────┐ ┌────────────────────┐

│ │ │ │ │ │

│ ThreadManager │─────▶ LlmService │─────▶ OpenAILlmService │

│ │ │ Interface │ │ │

└───────────────────┘ └─────────────────┘ └────────────────────┘

│

▼

┌───────────────────┐ ┌─────────────────┐ ┌────────────────────┐

│ │ │ │ │ │

│ ResponseProcessor │◀────│ LlmResponse & │◀────│ OkHttp Client │

│ │ │ LlmMessage │ │ │

└───────────────────┘ └─────────────────┘ └────────────────────┘

│ │

▼ ▼

┌───────────────────┐ ┌────────────────────┐

│ │ │ │

│ ToolRegistry │ │ BillingService │

│ │ │ │

└───────────────────┘ └────────────────────┘

**2. LLM Service Interface**

The LlmService interface defines a contract for different LLM provider implementations. It declares a makeLlmApiCall method that accepts all necessary parameters for communicating with an LLM API, including messages, model selection, temperature, token limits, tool schemas, streaming options, and tracking parameters for billing. The method returns a CompletableFuture that resolves to an LLM response, allowing for asynchronous execution.

**3. OpenAI Implementation**

**3.1 Configuration and Initialization**

The OpenAILlmService is a Spring service that implements the LlmService interface for communicating with OpenAI's API. It's configured with API credentials and endpoints through Spring's @Value annotation. The service uses OkHttpClient for HTTP communication, Jackson's ObjectMapper for JSON processing, and a BillingServiceFacade for tracking usage costs. Dependencies are injected through Spring's constructor injection.

**3.2 Request Construction**

When preparing a request to the OpenAI API, the service constructs a request body with all necessary parameters:

1. Basic parameters like model, temperature, max\_tokens, and stream flag
2. Formatted messages converted from the internal LlmMessage format to OpenAI's expected format
3. Special handling for different content types, including strings and multimodal content
4. Optional fields like name and tool\_call\_id for specific message types
5. Tool schemas that define available tools, with tool\_choice set to "auto"
6. Function schemas for backward compatibility with older API versions

This method ensures that all messages and parameters are properly formatted according to OpenAI's API specifications.

**3.3 Making the API Call**

The makeLlmApiCall method implements the LlmService interface to make actual API calls to OpenAI:

1. It wraps the entire operation in CompletableFuture.supplyAsync() for asynchronous execution
2. Creates the request body with all necessary parameters
3. Logs the API call at debug level (avoiding sensitive information in production logs)
4. Builds an HTTP request with proper headers, including authentication
5. Executes the request and handles the response
6. Provides specialized handling for streaming vs. non-streaming responses
7. Includes comprehensive error handling for API errors and exceptions
8. Returns either parsed API responses or streaming responses based on the stream parameter

This method serves as the core communication point with the OpenAI API, handling all the details of HTTP requests, authentication, and response processing.

**3.4 Response Parsing**

Response parsing is a critical part of the OpenAI integration:

1. The parseApiResponse method:
   * Extracts the response body and converts it to a JSON object
   * Navigates the JSON structure to find the message content and metadata
   * Constructs a LlmResponse object with the model name and content
   * Handles tool calls by extracting and parsing them
   * Supports backward compatibility with function calls
   * Records usage statistics for billing purposes
2. The parseToolCalls method:
   * Takes a list of tool call objects from the API response
   * Converts each to an internal ToolCall representation
   * Extracts the ID, type, and function details
   * Handles the function name and arguments specifically
   * Returns a list of structured ToolCall objects

Together, these methods transform the raw API response into structured objects that can be used by the rest of the application.

**4. Streaming Implementation**

**4.1 Streaming Response Handling**

For streaming responses, the service uses a specialized handler:

1. The handleStreamingResponse method:
   * Processes Server-Sent Events (SSE) from the OpenAI API
   * Reads and accumulates chunks as they arrive
   * Maintains builders for constructing tool calls incrementally
   * Tracks the model and other metadata
   * Handles different types of deltas (content, tool calls)
   * Detects completion markers and finish reasons
   * Builds a final LlmResponse when streaming is complete
   * Includes error handling for parsing issues and IO exceptions
2. The streaming flow:
   * Processes SSE lines that start with "data: "
   * Handles special markers like "[DONE]"
   * Extracts and accumulates content deltas
   * Builds tool calls incrementally as parts arrive
   * Finalizes tool calls when finish\_reason indicates completion
   * Estimates token usage for billing (since streaming responses don't include usage stats)

This approach enables real-time processing of LLM outputs, allowing for immediate updates to the client as the model generates text.

**4.2 Reactive Streaming Implementation**

For a truly reactive streaming implementation, the service uses Spring WebFlux:

1. The streamLlmApiCall method:
   * Creates a Flux using the create() operator
   * Prepares the request body with streaming enabled
   * Sets up an asynchronous HTTP request with OkHttp's enqueue method
   * Handles success and failure callbacks
   * Processes the streaming response line by line
   * Parses and emits chunks as they arrive
   * Completes the stream when the "[DONE]" marker is received
2. The parseStreamingChunk method:
   * Converts raw JSON chunks to structured LlmResponseChunk objects
   * Extracts the model information if available
   * Processes choice deltas for content and tool calls
   * Creates structured Delta objects for tool calls
   * Marks tool calls as finished when the finish\_reason indicates completion
   * Returns a complete chunk object for each delta

This reactive approach integrates well with Spring WebFlux applications, allowing for non-blocking processing of streaming responses all the way from the LLM to the client.

**5. Model Configuration and Selection**

**5.1 Model Resolution**

Model resolution is an important part of the service, handling various ways users might specify models:

1. Default handling:
   * If no model is specified, uses "gpt-4o-mini" as the default
   * Maintains a list of officially supported models
2. Model name normalization:
   * Accepts variations of model names (with or without hyphens, case insensitive)
   * Maps unofficial shorthand names to their official equivalents
   * Converts variations like "gpt4" to "gpt-4-turbo" or "haiku" to "claude-3-haiku"
3. Fallback behavior:
   * If an unknown model is requested, logs a warning
   * Returns the default model as a fallback
   * Provides clear logging about the substitution

This approach makes the service more user-friendly while ensuring that only supported models are used.

**5.2 Model-Specific Configurations**

Different LLM providers may require specific parameter transformations:

1. The applyModelSpecificConfig method:
   * Modifies the request body based on the target model
   * Handles Claude-specific parameters for Anthropic models
   * Renames parameters as needed (e.g., max\_tokens to max\_tokens\_to\_sample)
   * Reformats tool definitions to match the provider's expected format
2. The convertToolsForClaude method:
   * Transforms OpenAI-style tool definitions to Claude's format
   * Maps function names, descriptions, and parameter schemas
   * Handles differences in parameter naming and structure
   * Ensures compatibility with Claude's API expectations

This flexibility allows the agent to work with multiple LLM providers while maintaining a consistent interface for the rest of the application.

**6. Billing and Usage Tracking**

**6.1 Token Counting and Billing**

Usage tracking is critical for billing and resource management:

1. The BillingServiceFacade:
   * Records token usage for each LLM call
   * Calculates costs based on model-specific pricing
   * Maintains usage records for billing periods
   * Provides methods to query usage statistics
2. Usage recording process:
   * Extracts token counts from LLM response usage data
   * For streaming responses, estimates token usage
   * Associates usage with user ID, run ID, and timestamp
   * Calculates cost based on prompt and completion tokens
   * Persists usage records in the database
3. Cost calculation:
   * Uses model-specific rates for prompt and completion tokens
   * Accounts for different pricing tiers and models
   * Handles special cases like summarization or system messages
   * Provides accurate billing information for user invoicing

This comprehensive tracking ensures that usage is accurately measured and billed, while also providing valuable insights into resource utilization.

**7. Error Handling and Resilience**

The LLM integration includes robust error handling:

1. API error handling:
   * Detects and logs HTTP errors from the LLM provider
   * Includes detailed error information for debugging
   * Provides appropriate error messages for client feedback
   * Implements retry logic for transient errors
2. Timeout handling:
   * Sets appropriate timeouts for API calls
   * Detects and handles connection timeouts
   * Provides fallback behavior for timeout scenarios
   * Logs timeout information for monitoring
3. Resilience patterns:
   * Uses circuit breakers for API call protection
   * Implements backoff strategies for retries
   * Provides graceful degradation options
   * Ensures system stability during API outages

These measures ensure that the system remains operational even when external services experience issues, providing a reliable experience for users.

**8. Security Considerations**

Security is a critical aspect of LLM integration:

1. API key management:
   * Securely stores API keys in environment variables
   * Never logs or exposes keys in responses
   * Rotates keys periodically for security
   * Uses separate keys for different environments
2. Data protection:
   * Avoids sending sensitive information to the LLM
   * Implements content filtering for user inputs
   * Sanitizes responses to prevent data leakage
   * Logs minimal information for compliance
3. User authentication:
   * Verifies user permissions before making API calls
   * Associates usage with specific user accounts
   * Enforces rate limits and quotas per user
   * Prevents unauthorized access to LLM capabilities

These security measures protect both the system and user data while allowing appropriate access to LLM functionality.