📖 Standard Operating Procedures

AI Agents Platform - Tools, Features & API Usage Guide

Version 2.5

Table of Contents

ⓘ 1. System Overview

The AI Agents Platform is a comprehensive system for creating, managing, and deploying AI agents with advanced capabilities including tool calling, MCP integration, and RAG-based knowledge retrieval.

Key Features:

- Multi-agent management with user-scoped access
- Tool execution system with 16+ built-in tools (including video & speech)
- Video generation: text_to_video, image_to_video, combine_video_audio
- Text-to-speech (AWS Polly, synchronous) for narration and demo videos
- Conversation context passed to tools—previous conversation and current query used to fill tool parameters (e.g. text_prompt, url)
- MCP (Model Context Protocol) server integration
- RAG-based knowledge retrieval from training data
- Conversation memory and context management (streaming and non-streaming)
- API key authentication (user-level and agent-level)
- REST API credit deduction: /api/v1/chat and /api/v1/query bill the agent owner per api_call
- Real-time agent testing and monitoring
- Immediate conversation_id return for context preservation
- UTF8MB4 database support for emoji and special characters
- URL resolver normalizes URLs (adds https:// when scheme is missing)
- Social platform integration (social.oshaani.com)
- Footer social links configurable in Django Admin (Social Links model)
- Public share URLs for published agents
- Email-based agent sharing with expiration
- Welcome email system for new users
- Tutorial button on chat page; video tutorial on intro page
- Pay-As-You-Go credits model with free credits for new users

Oshaani Ecosystem:

> The Oshaani platform is part of a growing ecosystem of AI-powered solutions:
>
> - **AI Agents Platform (oshaani.com):** Create, train, and deploy AI agents
> - **Oshaani Social (social.oshaani.com):** Connect, share, and collaborate with AI agents in a social environment
> - **Developer Tools:** Advanced APIs, SDKs, and developer resources (coming soon)

🧩 2. Core Components

3.1 Agent Model

The `Agent` model represents an AI agent with the following attributes:

- **Status:** draft → training → testing → published
- **Configuration:** JSON field storing agent settings, enabled/disabled tools, RAG config
- **Model:** Reference to AIModel (Bedrock or Ollama)
- **User:** Owner of the agent (for user-scoped access)
- **API Keys:** Agent-level API keys for authentication

3.2 AgentLoop

The `AgentLoop` class implements the core agent execution logic:

- Manages conversation context and memory
- Handles tool calling iterations (max 10 iterations)
- Integrates with LLM providers (Bedrock/Ollama)
- Retrieves relevant context via RAG
- Processes tool results and generates final responses

3.3 ToolExecutor

The `ToolExecutor` manages all available tools:

- **Default Tools:** web_search, url_resolver, code_executor, transcription, text_to_image, text_to_video, image_to_video, text_to_speech, combine_video_audio, svg_diagram, ocr, summarization, question_answering, translation, read_file, write_file
- **Custom Tools:** User-defined HTTP-based tools
- **MCP Tools:** Tools from connected MCP servers
- Filters tools based on agent configuration (enabled/disabled)
- When video tools are enabled, url_resolver, text_to_speech, and combine_video_audio are auto-enabled for the video-with-sound flow

3.4 MCP Integration

MCP (Model Context Protocol) integration allows agents to use external tools:

- HTTP and STDIO transport support
- Automatic tool discovery and registration

- User API key authentication
- Tool prefixing to avoid conflicts

3. Authentication & Authorization

5.1 Authentication Methods

- **User API Keys:** From UserProfile or UserAPIKey models (for MCP and API access)
- **Agent API Keys:** From Agent model (for agent-specific API access)
- **Session Authentication:** For web dashboard access

5.2 Authorization

- Users can only access their own agents
- All data queries are scoped to the authenticated user
- MCP servers use user API keys for authentication

> **Security Best Practices:**
> - API keys are hashed using SHA-256
> - Keys are shown only once when generated
> - Last used timestamps are tracked
> - Keys can be revoked at any time

4. Tools Configuration & Feature Enabling

4.1 Available Tools

Agents have access to multiple types of tools:

✓ Built-in Tools (16+ tools)

- **read_file** - Read files from storage
- **write_file** - Write files to storage
- **web_search** - Search the web
- **url_resolver** - Resolve and fetch URLs (auto-adds https:// if missing)
- **code_executor** - Execute Python code
- **transcription** - Transcribe audio (AWS Transcribe)
- **text_to_image** - Generate images (AWS Bedrock)
- **text_to_video** - Generate video from text (Bedrock Luma/ Nova)
- **image_to_video** - Generate video from image
- **text_to_speech** - Generate speech from text (AWS Polly, synchronous)
- **combine_video_audio** - Mux video + audio for demo videos with sound
- **svg_diagram** - Generate SVG diagrams for buildings/ architecture
- **ocr** - Extract text from images/PDFs (AWS Textract)
- **summarization** - Summarize content (AWS Bedrock)
- **question_answering** - Answer questions (AWS Bedrock)
- **translation** - Translate text (AWS Translate)

✓ Custom Tools

- User-defined HTTP-based tools
- Up to 10 custom tools per agent
- Configurable endpoints and methods
- Custom authentication headers

✓ MCP Tools

- Tools from MCP servers
- Automatically discovered
- Prefixed with server name
- Dynamic tool registration

4.2 Enabling and Disabling Tools

You can control which tools are available to your agent by enabling or disabling them. This allows you to customize agent capabilities based on your needs.

4.2.1 Via Dashboard (Recommended)

Steps to Enable/Disable Tools:

1. Navigate to your agent's training page: `/dashboard/{agent_id}/train/`
2. Scroll to the "Tools Configuration" section
3. You'll see three categories:
    - **Default Tools** - Built-in system tools
    - **Custom Tools** - Your custom HTTP-based tools
    - **MCP Tools** - Tools from connected MCP servers

4. Toggle tools on/off using the checkboxes
5. Click "Save Tool Configuration" to apply changes

4.2.2 Via API

Update agent configuration to enable/disable tools:

Enable Specific Tools:

```
PATCH /api/agents/{agent_id}/
Authorization: Bearer YOUR_USER_TOKEN
Content-Type: application/json

{
  "configuration": {
    "enabled_tools": ["web_search", "url_resolver", "code_executor
    "disabled_tools": []
  }
}
```

Disable Specific Tools:

```
PATCH /api/agents/{agent_id}/
Authorization: Bearer YOUR_USER_TOKEN
Content-Type: application/json

{
  "configuration": {
    "disabled_tools": ["text_to_image", "transcription"]
  }
}
```

4.2.3 Tool Configuration Logic

How Tool Filtering Works:

- **If `enabled_tools` is empty:** All tools are available (except those in `disabled_tools`)

- **If `enabled_tools` has items:** Only tools listed in `enabled_tools` are available
- **If a tool is in `disabled_tools`:** It's always disabled (unless also in `enabled_tools`)
- **Priority:** `enabled_tools` takes precedence over `disabled_tools`

4.2.4 Example Configurations

Example 1: Enable Only Web Search and URL Resolver

```
{
  "enabled_tools": ["web_search", "url_resolver"]
}
```

Example 2: Disable Image Generation Tools

```
{
  "disabled_tools": ["text_to_image"]
}
```

Example 3: Enable All Tools Except Code Execution

```
{
  "disabled_tools": ["code_executor"]
}
```

4.3 Feature Enabling

Beyond tools, you can enable/disable various features for your agent:

4.3.1 RAG (Retrieval-Augmented Generation)

Enable RAG:

```
{
  "configuration": {
    "use_rag": true,
    "rag_top_k": 5,
    "rag_threshold": 0.7
  }
}
```

- `use_rag` : Enable/disable RAG feature
- `rag_top_k` : Number of relevant chunks to retrieve (default: 5)
- `rag_threshold` : Similarity threshold for retrieval (default: 0.7)

4.3.2 Tool Calling

Enable/Disable Tool Calling:

```
{
  "configuration": {
    "tools_enabled": true,
    "max_tool_iterations": 10
  }
}
```

- `tools_enabled` : Enable/disable tool calling capability
- `max_tool_iterations` : Maximum tool call iterations (default: 10)

4.3.3 Conversation Memory

Configure Memory:

```
{
  "configuration": {
    "use_memory": true,
    "max_history_messages": 50
  }
}
```

- `use_memory` : Enable conversation memory
- `max_history_messages` : Maximum messages to keep in context

4.4 Checking Tool Status

View Enabled Tools:

- **Via Dashboard:** Go to agent training page → Tools Configuration section
- **Via API:** `GET /api/agents/{agent_id}/` → Check `configuration.enabled_tools` and `configuration.disabled_tools`

Important Notes:

- Tool configuration changes take effect immediately for new requests
- Published agents may need to be republished to apply configuration changes
- Disabled tools won't appear in tool schemas sent to the LLM
- Custom tools and MCP tools follow the same enable/disable rules

</> 5. API Usage Guide

5.1 API Types Overview

The platform provides multiple API types to suit different use cases:

REST API

- Standard REST endpoints
- Django REST Framework
- Full CRUD operations
- Session or API key auth

REST API v1

- Simple chat/query endpoints
- Agent API key auth
- Streamlined interface
- Quick integration

MCP API

- Model Context Protocol
- Tool discovery
- Server management
- User API key auth

5.2 REST API Endpoints

The REST API provides comprehensive access to all platform features.

5.2.1 Agent Management

Base URL: `/api/agents/`

Authentication: User API Key or Session

Endpoints:

- `GET /api/agents/` - List all user's agents
- `POST /api/agents/` - Create new agent
- `GET /api/agents/{id}/` - Get agent details
- `PATCH /api/agents/{id}/` - Update agent
- `DELETE /api/agents/{id}/` - Delete agent
- `POST /api/agents/{id}/publish/` - Publish agent
- `POST /api/agents/{id}/unpublish/` - Unpublish agent

5.2.2 Agent Interaction (Requires Agent API Key)

Base URL: `/api/agents/{id}/`

Authentication: Agent API Key

Endpoints:

- `POST /api/agents/{id}/chat/` - Chat with agent
- `POST /api/agents/{id}/query/` - Query agent
- `POST /api/agents/{id}/invoke/` - Invoke agent action
- `GET /api/agents/{id}/status/` - Get agent status
- `GET /api/agents/{id}/history/` - Get interaction history

5.2.3 Example: Create Agent via REST API

```
POST /api/agents/
Authorization: Bearer YOUR_USER_API_KEY
Content-Type: application/json

{
  "name": "Customer Support Bot",
  "description": "Helps customers with common questions",
  "model_id": 1,
  "agent_type": "quick_bot",
  "configuration": {
    "system_prompt": "You are a helpful customer support assistant
    "enabled_tools": ["web_search", "url_resolver"],
    "use_rag": true
  }
}
```

5.2.4 Example: Chat with Agent via REST API

```
POST /api/agents/{agent_id}/chat/
Authorization: ApiKey YOUR_AGENT_API_KEY
Content-Type: application/json

{
  "message": "What is the weather today?",
  "conversation_id": "optional-conversation-id"
}
```

Response:

```
{
  "response": "I'll search for today's weather information...",
  "tool_calls": [
    {
      "tool_name": "web_search",
      "parameters": {"query": "weather today"},
      "result": "..."
    }
  ],
  "conversation_id": "conv-123",
  "iterations": 2
}
```

5.3 REST API v1 Endpoints

Simplified API endpoints for quick integration.

5.3.1 Simple Endpoints

Base URL: `/api/v1/`

Authentication: Agent API Key

Endpoints:

- `POST /api/v1/chat` - Simple chat endpoint
- `POST /api/v1/query` - Simple query endpoint
- `GET /api/v1/agents` - List accessible agents
- `GET /api/v1/agents/{id}` - Get agent info

5.3.2 Example: REST API v1 Chat

```
 POST /api/v1/chat
Authorization: ApiKey YOUR_AGENT_API_KEY
Content-Type: application/json

{
  "message": "Hello, how can you help me?"
}
```

Response:

```
{
  "response": "Hello! I can help you with...",
  "tool_calls": [],
  "iterations": 1,
  "conversation_id": "conv-abc123"
}
```

5.3.3 REST API Endpoints

Base URL: `/api/`

Authentication: Agent API Key

Endpoints:

- `POST /api/create_conversation` - Create new conversation
- `POST /api/continue_conversation` - Continue existing conversation
- `GET /api/get_answer` - Get answer by request_id
- `POST /api/find_conversation` - Find conversation details
- `PUT /api/upload_file` - Upload files
- `GET /api/download_file` - Download files
- `POST /api/create_upload_url` - Create upload URL
- `POST /api/get_download_url` - Get download URL

5.3.4 Example: Create Conversation

```
POST /api/create_conversation
Authorization: ApiKey YOUR_AGENT_API_KEY
Content-Type: application/json

{
  "message": "What is machine learning?"
}
```

Response (Async):

```
{
  "request_id": "req-550e8400-e29b-41d4-a716",
  "conversation_id": "conv-550e8400-e29b-41d4-a716",
  "status": "pending",
  "message": "Request queued for processing. Use /api/get_answer wi
}
```

Key Points:

- `conversation_id` is returned immediately - use it to continue the conversation
- `request_id` is used to poll for the initial response
- You can use `conversation_id` with `/api/continue_conversation` to maintain chat context

Poll for Results:

```
GET /api/get_answer?request_id=req-550e8400-e29b-41d4-a716
Authorization: ApiKey YOUR_AGENT_API_KEY
```

Continue Conversation:

```
POST /api/continue_conversation
Authorization: ApiKey YOUR_AGENT_API_KEY
Content-Type: application/json

{
  "conversation_id": "conv-550e8400-e29b-41d4-a716",
  "message": "Tell me more about that"
}
```

5.4 MCP API Endpoints

API endpoints for managing MCP servers and discovering tools.

5.4.1 MCP Server Management

Base URL: `/api/mcp-servers/`

Authentication: User API Key or Session

Endpoints:

- `GET /api/mcp-servers/` - List MCP servers
- `POST /api/mcp-servers/` - Create MCP server
- `GET /api/mcp-servers/{id}/` - Get server details
- `PUT /api/mcp-servers/{id}/` - Update server
- `DELETE /api/mcp-servers/{id}/` - Delete server
- `POST /api/mcp-servers/{id}/test_connection/` - Test connection
- `GET /api/mcp-servers/list_tools/?agent_id={id}` - List available tools

5.4.2 Example: Create MCP Server

```
 POST /api/mcp-servers/
Authorization: Bearer YOUR_USER_API_KEY
Content-Type: application/json

{
  "agent": 1,
  "name": "filesystem",
  "description": "Local filesystem access",
  "transport_type": "stdio",
  "command": "npx",
  "args": ["-y", "@modelcontextprotocol/server-filesystem", "."],
  "auto_connect": true
}
```

5.5 Authentication Methods

5.5.1 User API Key Authentication

Use Case: Managing agents, accessing user resources
Header Format:

```
Authorization: Bearer YOUR_USER_API_KEY
```

Where to Get: User Profile → API Keys section

5.5.2 Agent API Key Authentication

Use Case: Interacting with specific agents
Header Format:

```
Authorization: ApiKey YOUR_AGENT_API_KEY
```

Alternative:

```
X-API-Key: YOUR_AGENT_API_KEY
```

Where to Get: Agent Detail Page → API Access section (only for published agents)

5.5.3 Session Authentication

Use Case: Web dashboard access
Method: Django session cookies (automatic in browser)
Note: Only works for web requests, not API clients

5.6 API Response Formats

5.6.1 Success Response

```
{
  "success": true,
  "data": { ... },
  "message": "Operation completed successfully"
}
```

5.6.2 Error Response

```
{
  "error": "Error message here",
  "detail": "Additional error details"
}
```

5.6.3 Async Response

```
{
  "request_id": "req-550e8400-e29b-41d4-a716",
  "conversation_id": "conv-550e8400-e29b-41d4-a716",
  "status": "pending",
  "message": "Request queued for processing. Use /api/get_answer wi
}
```

Note: The `conversation_id` is now returned immediately when creating a conversation, allowing you to maintain chat context across multiple API calls.

5.7 Complete API Examples

5.7.1 Python Example: Full Agent Workflow

```
 import requests

# Configuration
BASE_URL = "https://your-domain.com/api"
AGENT_API_KEY = "your-agent-api-key"

# Chat with agent
response = requests.post(
    f"{BASE_URL}/v1/chat",
    headers={
        "Authorization": f"ApiKey {AGENT_API_KEY}",
        "Content-Type": "application/json"
    },
    json={
        "message": "What is artificial intelligence?"
    }
)

result = response.json()
print(f"Response: {result['response']}")
print(f"Tool calls: {result.get('tool_calls', [])}")
```

5.7.2 JavaScript Example: Async Conversation

```
 const API_KEY = "your-agent-api-key";
const BASE_URL = "https://your-domain.com/api";

// Create conversation (async)
const createResponse = await fetch(`${BASE_URL}/create_conversatio
  method: 'POST',
  headers: {
    'Authorization': `ApiKey ${API_KEY}`,
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({
    message: "Hello, agent!"
  })
});

const { request_id, conversation_id } = await createResponse.json(

// Store conversation_id for maintaining context
console.log('Conversation ID:', conversation_id);

// Poll for results
const pollAnswer = async () => {
  const response = await fetch(
    `${BASE_URL}/get_answer?request_id=${request_id}`,
    {
      headers: { 'Authorization': `ApiKey ${API_KEY}` }
    }
  );
  return response.json();
};

// Poll until completed
let result = await pollAnswer();
while (result.status === 'pending' || result.status === 'processin
  await new Promise(resolve => setTimeout(resolve, 1000));
  result = await pollAnswer();
}

console.log('Final response:', result.response);

// Continue the conversation using conversation_id
const continueResponse = await fetch(`${BASE_URL}/continue_convers
```

```
  method: 'POST',
  headers: {
    'Authorization': `ApiKey ${API_KEY}`,
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({
    conversation_id: conversation_id,
    message: "Tell me more about that"
  })
});

const continueResult = await continueResponse.json();
console.log('Continued conversation response:', continueResult);
```

Benefits of conversation_id:

- Maintains chat context across multiple API calls
- Available immediately after creating a conversation
- Enables multi-turn conversations with context preservation

5.7.3 cURL Example: Multiple API Types

REST API:

```
curl -X POST https://your-domain.com/api/agents/1/chat/ \
  -H "Authorization: ApiKey YOUR_AGENT_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{"message": "Hello"}'
```

REST API v1:

```
curl -X POST https://your-domain.com/api/v1/chat \
  -H "Authorization: ApiKey YOUR_AGENT_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{"message": "Hello"}'
```

MCP API:

```
curl -X GET https://your-domain.com/api/mcp-servers/list_tools/?ag
  -H "Authorization: Bearer YOUR_USER_API_KEY"
```

API Best Practices:

- Always use HTTPS in production
- Store API keys securely (environment variables, secrets manager)
- Handle errors gracefully with retry logic
- Use appropriate API type for your use case
- Monitor API usage and rate limits
- Keep API keys rotated regularly

6. File Upload and Download

6.1 File Upload

> **Endpoint:** `PUT /api/upload_file`
> **Authentication:** Agent API Key
> **Max File Size:** 100MB
> **Supported Formats:** All file types (PDF, images, text, etc.)

6.2 File Download

> **Endpoint:** `GET /api/download_file?file_id=<file_id>`
> **Purpose:** Get download URL for uploaded or generated files
> **Authentication:** Agent API Key

6.3 Generated Files

Files generated by tools (e.g., images from `text_to_image`) are automatically registered and accessible via the download_file endpoint. The `file_id` is included in tool call results.

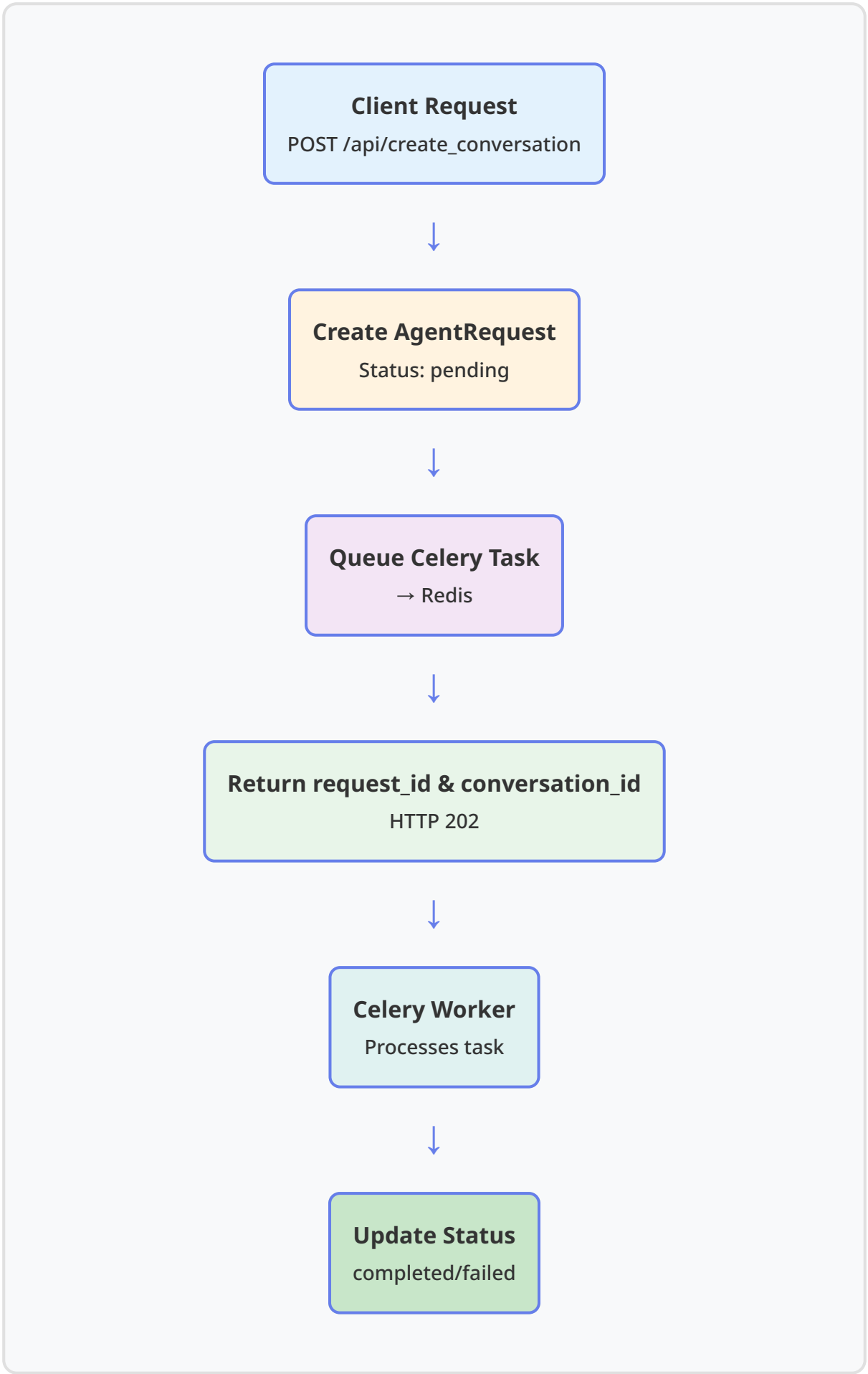
> **Tools that generate files:**
> - `text_to_image` - Generates images (returns `file_id`)
> - `ocr` - May extract images from PDFs (returns `file_id` if applicable)
> - `transcription` - May save audio files (returns `file_id` if applicable)

🔄 7. Async Processing with Celery

7.1 Overview

The agent API uses asynchronous processing with Celery and Redis for non-blocking request handling. All conversation requests return immediately with a `request_id` that can be used to poll for results, and a `conversation_id` that can be used to maintain chat context across multiple API calls.

7.2 Architecture


Client Request
POST /api/create_conversation
Create AgentRequest
Status: pending
Queue Celery Task
→ Redis
Return request_id & conversation_id
HTTP 202
Celery Worker
Processes task
Update Status
completed/failed

7.3 Request Status

- **pending** - Request queued, waiting for worker
- **processing** - Worker is processing the request
- **completed** - Request completed successfully
- **failed** - Request failed (check `error_message`)

7.3.1 Conversation ID in Async Responses

> **Important:** When creating a conversation via `/api/create_conversation`, the response includes both:
>
> - `request_id` - Used to poll for the initial response via `/api/get_answer`
> - `conversation_id` - Returned immediately, used to continue the conversation via `/api/continue_conversation`
>
> The `conversation_id` is available immediately, allowing you to start a new conversation thread while the initial request is still being processed asynchronously.

7.4 Setup Requirements

> **Required Services:**
>
> - Redis server (message broker)
> - Celery worker process
> - Django application server

> **Monitoring:**
>
> - Check Celery worker status: `ps aux | grep celery`
> - Check Redis connection: `redis-cli ping`
> - Monitor queue length: `redis-cli LLEN celery`
> - View requests in Django Admin → Agent Requests

8. RAG (Retrieval Augmented Generation)

9.1 Overview

RAG enables agents to retrieve relevant information from training data using semantic search, improving response accuracy and relevance.

9.2 Features

- **Automatic Chunking:** Training data is automatically chunked into smaller pieces
- **Embedding Generation:** Uses AWS Bedrock Titan Embeddings or Ollama embeddings
- **Semantic Search:** Retrieves only relevant chunks based on query similarity
- **Vector Store Backends:** Supports both in-memory and Qdrant vector databases

9.3 Vector Store Configuration

In-Memory (Default)

- Fast but data is lost on server restart
- Suitable for development and testing

Qdrant (Production)

- Persistent vector database
- Better scalability
- Setup: `docker run -p 6333:6333 qdrant/qdrant`

9.4 Using RAG

1. **Index Training Data:** `POST /api/agents/{id}/index_rag/`
2. **Check RAG Status:** `GET /api/agents/{id}/rag_status/`
3. **Enable/Disable RAG:** Set `use_rag: true/false` in agent configuration

RAG Notifications:

- Success notification sent when indexing completes with chunk count
- Failure notification sent if indexing fails with error details
- Optional email notifications (configure `SEND_RAG_NOTIFICATIONS=true`)

9. MCP Integration

10.1 Overview

Agents can be enhanced with MCP servers to access additional tools and capabilities beyond the built-in tools.

10.2 Supported Transports

- **STDIO:** Run MCP servers as subprocesses (e.g., `npx -y @modelcontextprotocol/server-filesystem`)
- **HTTP/SSE:** Connect to remote MCP servers (coming soon)

10.3 Adding MCP Servers

Step 1: Create MCP Server Configuration

`POST /api/mcp-servers/`

```
{
  "agent": 1,
  "name": "filesystem",
  "description": "Local filesystem access",
  "transport_type": "stdio",
  "command": "npx",
  "args": ["-y", "@modelcontextprotocol/server-filesystem", "."],
  "auto_connect": true
}
```

Step 2: Test Connection

`POST /api/mcp-servers/{id}/test_connection/`

Step 3: List Available Tools

`GET /api/mcp-servers/list_tools/?agent_id=1`

10.4 MCP Tools

- Tools are automatically discovered from connected MCP servers

- Tool names are prefixed with server name (e.g., `mcp_filesystem_read_file`)
- Tools are integrated into the agent's tool system
- MCP tools work alongside built-in tools seamlessly

10. Deployment & Operations

11.1 System Requirements

Core Services

- Python 3.9+
- MySQL database
- Redis (for Celery & caching)

Optional Services

- Qdrant (for vector storage)
- AWS Bedrock access
- Ollama (local LLM)

11.2 Key Services

- **Django Application:** Main web server and API
- **Celery Workers:** Background task processing for async requests
- **Redis:** Message broker for Celery and response caching
- **Qdrant Server:** Vector database for RAG (optional)

11.3 Database Models

Key Models:

- `Agent` - Agent configuration and status
- `Conversation` - Conversation threads
- `ConversationMessage` - Individual messages
- `AgentRequest` - Async request tracking
- `ConversationFile` - Uploaded/generated files
- `TrainingData` - Agent training data
- `MCPServer` - MCP server configurations

- `ToolCall` - Tool execution tracking

11.3.1 Database Character Encoding

UTF8MB4 Support:

- All text fields use UTF8MB4 encoding to support emojis and 4-byte UTF-8 characters
- Database and Django settings are configured for UTF8MB4 by default
- Migration `0020_fix_toolcall_utf8mb4` ensures `ToolCall.result_content` and `ToolCall.error` columns support emojis
- This enables proper storage of tool results containing emoji characters (e.g., ✓, 📄, 🔧)

11.4 Caching

Redis-based Caching:

- Agent responses are cached for identical queries
- Cache key includes: agent_id, query, system_prompt, model_id, training_data_hash
- Cache is automatically invalidated when training data or configuration changes
- Cache is used for new conversations (0 messages), skipped for ongoing conversations

11.5 Security

Security Features:

- **API Keys:** Hashed using SHA-256 before storage
- **Authentication:** Agent API keys are tightly bound to specific agents
- **Permissions:** Agents can only access their own conversations and files
- **HTTPS:** Required in production environments

Monitoring & Logging:

- All components use Python logging
- Tool execution is logged with results
- Agent conversations are stored in database
- Error tracking for debugging
- Request status visible in Django Admin

11. Connectors & OAuth Callback URLs

12.1 Overview

The platform supports integration with multiple external services through OAuth 2.0 connectors. Each connector requires a callback URL to be registered with the OAuth provider.

Supported Connector Types:

- **JIRA** - Atlassian JIRA Cloud (OAuth 2.0 3LO)
- **Confluence** - Atlassian Confluence (OAuth 2.0 3LO)
- **GitLab** - GitLab repositories (OAuth 2.0)
- **GitHub** - GitHub repositories (OAuth 2.0)
- **Google** - Google Workspace (Drive, Docs, Sheets, Gmail) (OAuth 2.0)
- **Microsoft** - Microsoft 365 (OneDrive, SharePoint, Outlook, Teams) (OAuth 2.0)
- **Slack** - Slack workspace (OAuth 2.0) - *Coming soon*

12.2 Callback URL Format

All connectors use the same callback URL pattern:

Callback URL Pattern:

`https://your-domain.com/api/connectors/{connector_id}/oauth/callback/`

- **Replace** `{connector_id}` with the actual connector ID after creating the connector
- The callback URL is provided in the API response when creating a connector
- The URL must match **EXACTLY** (including protocol, domain, path, and trailing slash)

12.3 Getting Callback URLs

Callback URLs are automatically generated when you create a connector. You can get the callback URL in two ways:

1. **When Creating a Connector:** The API response includes the `callback_url` field
2. **When Initiating OAuth:** The `POST /api/connectors/{connector_id}/oauth/initiate/` endpoint returns the callback URL

12.4 Connector-Specific Callback URL Registration

⚠ JIRA Connector

OAuth Provider: Atlassian Developer Console

OAuth Type: OAuth 2.0 (3-Legged OAuth / 3LO)

Registration Steps:

1. Go to Atlassian Developer Console
2. Click **Create** → **New app**
3. Choose **OAuth 2.0 (3LO)** integration type
4. Fill in app details (name, logo, description)
5. Navigate to **APIS AND FEATURES** → **OAuth 2.0 (3LO)**
6. Add **Authorization callback URL**: `https://your-domain.com/api/connectors/{connector_id}/oauth/callback/`
7. Add **Jira platform REST API** under **APIS AND FEATURES**
8. Configure required scopes:
   - `read:jira-work` - Read JIRA issues and work items
   - `read:jira-user` - Read user information
   - `offline_access` - Required for refresh tokens
9. Copy the **Client ID** and **Client Secret**

Important Notes:

- This connector is designed for **JIRA Cloud** (atlassian.net domains)
- JIRA Server/Data Center requires OAuth 1.0a, which is not currently supported
- Each connector requires its own callback URL - add multiple URLs for different connectors

- The callback URL must match **EXACTLY** (including protocol, domain, path, and trailing slash)

⚠ Confluence Connector

OAuth Provider: Atlassian Developer Console

OAuth Type: OAuth 2.0 (3-Legged OAuth / 3LO)

Registration Steps:

1. Go to Atlassian Developer Console
2. Click **Create** → **New app**
3. Choose **OAuth 2.0 (3LO)** integration type
4. Fill in app details (name, logo, description)
5. Navigate to **APIS AND FEATURES** → **OAuth 2.0 (3LO)**
6. Add **Authorization callback URL**: `https://your-domain.com/api/connectors/{connector_id}/oauth/callback/`
7. Add **Confluence REST API** under **APIS AND FEATURES**
8. Configure required scopes:
   - `read:confluence-content.all` - Read all Confluence content
   - `read:confluence-space.summary` - Read space information
   - `offline_access` - Required for refresh tokens
9. Copy the **Client ID** and **Client Secret**

Important Notes:

- This connector is designed for **Confluence Cloud** (atlassian.net domains)
- Each connector requires its own callback URL
- The callback URL must match **EXACTLY** (including protocol, domain, path, and trailing slash)

⚠ GitLab Connector

OAuth Provider: GitLab Application Settings

OAuth Type: OAuth 2.0

Registration Steps:

1. Go to your GitLab instance (e.g., `https://gitlab.com` or your self-hosted instance)
2. Navigate to **User Settings** → **Applications** (for user-level apps)
3. Or go to **Admin Area** → **Applications** (for instance-wide apps)
4. Click **Add new application**
5. Fill in the application details:
    - **Name:** Your application name
    - **Redirect URI:** `https://your-domain.com/api/connectors/{connector_id}/oauth/callback/`
    - **Scopes:** Select the following scopes:
        - `read_api` - Read API data
        - `read_repository` - Read repository contents
        - `read_user` - Read user information (optional)
4. Click **Save application**
5. Copy the **Application ID** (Client ID) and **Secret** (Client Secret)

Important Notes:

- Works with both GitLab.com and self-hosted GitLab instances
- For self-hosted instances, use your instance URL as the base URL
- Each connector requires its own callback URL

⚠ GitHub Connector

OAuth Provider: GitHub Developer Settings

OAuth Type: OAuth 2.0

Registration Steps:

1. Go to GitHub Developer Settings
2. Click **New OAuth App** (or **OAuth Apps** → **New OAuth App**)
3. Fill in the application details:
    - **Application name:** Your application name
    - **Homepage URL:** Your application homepage (e.g., `https://your-domain.com`)

- **Authorization callback URL:** `https://your-domain.com/api/connectors/{connector_id}/oauth/callback/`

4. Click **Register application**
5. Copy the **Client ID** and click **Generate a new client secret**
6. Save the **Client Secret** immediately (it's only shown once)

OAuth Scopes: The connector automatically requests the following scopes:

- `repo` - Full control of private repositories (if accessing private repos)
- `read:org` - Read org and team membership (if accessing organization repos)

Important Notes:

- For **GitHub Enterprise Server**, use your instance's OAuth app settings (usually at `https://your-ghe-instance.com/settings/developers`)
- Each connector requires its own callback URL
- The callback URL must match **EXACTLY** (including protocol, domain, path, and trailing slash)

⚠ Google Connector

OAuth Provider: Google Cloud Console

OAuth Type: OAuth 2.0

Registration Steps:

1. Go to Google Cloud Console
2. Select your project or create a new one
3. Enable the required APIs:
   - Navigate to **APIs & Services** → **Library**
   - Enable **Google Drive API** (for Drive access)
   - Enable **Google Docs API** (for Google Docs)
   - Enable **Google Sheets API** (for Google Sheets)
   - Enable **Gmail API** (for Gmail access, if needed)
4. Navigate to **APIs & Services** → **Credentials**
5. Click **Create Credentials** → **OAuth client ID**

6. If prompted, configure the OAuth consent screen first:
    - Select **External** or **Internal** user type
    - Fill in app information (name, support email, developer contact)
    - Add scopes: `https://www.googleapis.com/auth/drive.readonly`, `https://www.googleapis.com/auth/documents.readonly`, etc.
7. Select **Web application** as application type
8. Add **Authorized redirect URIs**: `https://your-domain.com/api/connectors/{connector_id}/oauth/callback/`
9. Click **Create**
10. Copy the **Client ID** and **Client Secret**

OAuth Scopes: The connector uses the following scopes:

- `https://www.googleapis.com/auth/drive.readonly` - Read Google Drive files
- `https://www.googleapis.com/auth/documents.readonly` - Read Google Docs
- `https://www.googleapis.com/auth/spreadsheets.readonly` - Read Google Sheets
- `https://www.googleapis.com/auth/gmail.readonly` - Read Gmail messages (if enabled)

Important Notes:

- Each connector requires its own callback URL
- The OAuth consent screen must be configured before creating OAuth credentials
- For production use, you may need to verify your app with Google

⚠ Microsoft Connector

OAuth Provider: Azure Portal (Azure Active Directory)

OAuth Type: OAuth 2.0 (Microsoft Identity Platform)

Registration Steps:

1. Go to Azure Portal

2. Navigate to **Azure Active Directory → App registrations**
3. Click **New registration**
4. Fill in the application details:
    - **Name:** Your application name
    - **Supported account types:** Select appropriate option:
        - **Accounts in this organizational directory only** - Single tenant
        - **Accounts in any organizational directory** - Multi-tenant
        - **Accounts in any organizational directory and personal Microsoft accounts** - Multi-tenant + personal
    - **Redirect URI:** Select **Web** and enter `https://your-domain.com/api/connectors/{connector_id}/oauth/callback/`
4. Click **Register**
5. Copy the **Application (client) ID** from the Overview page
6. Create a **Client secret**:
    - Go to **Certificates & secrets**
    - Click **New client secret**
    - Add description and select expiration
    - Click **Add** and **copy the secret value immediately** (it's only shown once)
7. Configure API permissions:
    - Go to **API permissions**
    - Click **Add a permission**
    - Select **Microsoft Graph**
    - Select **Delegated permissions**
    - Add the following permissions:
        - `Files.Read` - Read files in OneDrive
        - `Sites.Read.All` - Read items in SharePoint
        - `Mail.Read` - Read mail in Outlook
        - `ChannelMessage.Read.All` - Read Teams messages (if needed)
        - `offline_access` - Required for refresh tokens
    - Click **Add permissions**
    - Click **Grant admin consent** if you have admin rights (recommended)

OAuth Scopes: The connector uses Microsoft Graph API with the following scopes:

- `Files.Read` - Read OneDrive files
- `Sites.Read.All` - Read SharePoint sites and documents
- `Mail.Read` - Read Outlook mail
- `ChannelMessage.Read.All` - Read Teams channel messages
- `offline_access` - Required for refresh tokens

Important Notes:

- Each connector requires its own callback URL
- Admin consent may be required for organization-wide access
- The callback URL must match **EXACTLY** (including protocol, domain, path, and trailing slash)

⚠ Slack Connector

Status: *Coming Soon* - OAuth implementation in progress

OAuth Provider: Slack API

OAuth Type: OAuth 2.0

Note: The Slack connector is defined in the system but OAuth integration is not yet implemented. Check back for updates.

12.5 Common Callback URL Issues

Common Errors and Solutions:

- **"Invalid callback URL" or "Redirect URI mismatch"**
  - Ensure the callback URL matches exactly (including `https://`, domain, path, and trailing `/`)
  - Check for typos in the URL
  - Verify the connector ID is correct

- **"The app's callback URL is invalid" (Atlassian)**
  - Get the exact callback URL from the API response when creating the connector
  - Copy the URL exactly as shown (including trailing slash)
  - Ensure you're registering it in the correct OAuth app
- **Multiple Connectors**
  - Each connector has a unique callback URL based on its ID
  - You must register each callback URL separately with the OAuth provider
  - Some providers support wildcard patterns, but exact URLs are recommended

12.6 Example Callback URLs

Example Callback URLs (replace with your actual domain and connector IDs):

- `https://example.com/api/connectors/1/oauth/callback/` (JIRA connector #1)
- `https://example.com/api/connectors/2/oauth/callback/` (GitHub connector #2)
- `https://example.com/api/connectors/3/oauth/callback/` (Google connector #3)
- `https://example.com/api/connectors/4/oauth/callback/` (Microsoft connector #4)

12.7 API Endpoints

Key Endpoints:

- **Create Connector:** `POST /api/connectors/` (returns callback_url)
- **Initiate OAuth:** `POST /api/connectors/{connector_id}/oauth/initiate/` (returns callback_url)
- **OAuth Callback:** `GET /api/connectors/{connector_id}/oauth/callback/` (handled automatically)

Best Practices:

- Always get the callback URL from the API response when creating a connector
- Register the callback URL in your OAuth provider before initiating the OAuth flow
- Use HTTPS in production environments
- Keep track of which callback URLs are registered for which connectors
- Test the OAuth flow after registering the callback URL

12. Agent Sharing & Public URLs

12.1 Overview

Published agents can be shared with others through two methods: email-based sharing and public share URLs. Both methods allow users to access and use your agents without requiring direct access to your account.

Email-Based Sharing

- Share with specific users via email
- Requires recipient to accept invitation
- Optional expiration dates
- Track acceptance status
- Can revoke access anytime

Public Share URLs

- Generate shareable link
- Anyone with URL can access
- No email required
- Track access count
- Can deactivate anytime

12.2 Email-Based Sharing

How It Works:

1. Navigate to your published agent's detail page
2. Click "Share Agent (Email)" button
3. Enter recipient's email address

4. Optionally add a message and set expiration
5. An email with a unique share link is sent to the recipient
6. Recipient clicks the link and accepts the share
7. They can now access and use your agent

12.2.1 API Endpoints

Share Agent:

`POST /dashboard/{agent_id}/share/`

Parameters:

- `email` - Recipient's email address (required)
- `message` - Optional message to include
- `expires_days` - Optional expiration in days

12.2.2 Managing Shares

Available Actions:

- **Resend Email:** Resend share invitation email
- **Withdraw Share:** Remove pending invitations
- **Revoke Access:** Remove access from accepted shares

12.3 Public Share URLs

How It Works:

1. Navigate to your published agent's detail page
2. Click "Public Share URL" button
3. Optionally set expiration date
4. Copy the generated share URL
5. Share the URL with anyone (via email, social media, website, etc.)
6. Anyone with the URL can access your agent (after login/registration)

12.3.1 Generating Public Share URLs

Via Dashboard:

1. Go to agent detail page: `/dashboard/{agent_id}/`
2. Click "Public Share URL" button
3. Set optional expiration (in days)
4. Click "Generate Share URL"
5. Copy the generated URL

Via API:

`POST /dashboard/{agent_id}/public-share/`

```
{
  "expires_days": 30  // Optional: expiration in days
}
```

12.3.2 Public Share URL Format

URL Pattern:

`https://your-domain.com/dashboard/public/{token}/`

Features:

- Unique token for each share
- Accessible by anyone with the URL
- Requires user login/registration to use
- Automatically redirects to agent chat after authentication

12.3.3 Managing Public Shares

View Active Share:

- Active public share URL is displayed on agent detail page
- Shows access count and last accessed time
- Shows expiration date if set

Deactivate Share:

- Click "Deactivate" button on agent detail page

- Or use API: `POST /dashboard/{agent_id}/public-share/deactivate/`
- Deactivated URLs no longer grant access

12.4 Access Control

Important Security Notes:

- Only **published** agents can be shared
- Shared users can **use** the agent but cannot modify it
- Agent owner retains full control and can revoke access anytime
- Public share URLs can be deactivated instantly
- Expired shares automatically lose access
- Usage is tracked and billed to the agent owner

12.5 Use Cases

Email-Based Sharing:

- Sharing with specific team members
- Client access to custom agents
- Controlled access with expiration
- Tracking who has accepted shares

Public Share URLs:

- Sharing on social media
- Embedding in websites or blogs
- Demo links for potential clients
- Public showcase of agent capabilities
- Community sharing and collaboration

12.6 API Examples

Generate Public Share URL:

```
POST /dashboard/{agent_id}/public-share/
Authorization: Session or User API Key
Content-Type: application/json

{
  "expires_days": 30
}
```

Response:

```
{
  "success": true,
  "share_url": "https://your-domain.com/dashboard/public/{token}/",
  "expires_at": "2024-02-15T00:00:00Z"
}
```

13. Billing & Subscription Plans

13.1 Overview

The platform uses a flexible billing system with multiple subscription tiers and a Pay-As-You-Go model for usage-based pricing.

Subscription Plans

- Free Tier
- Starter Plan
- Professional Plan
- Enterprise Plan
- Pay-As-You-Go

Usage Tracking

- Messages sent
- Tool calls executed
- API calls made
- Storage used (MB)
- Agents created

💰 Credits System

- Pay-As-You-Go billing (credits-only)
- Credit balance and ledger (UserCredits, CreditLedger)
- Automatic deduction on message, API call, tool, video
- Free credits for new users (configurable amount)
- Low-balance email notifications

13.2 Subscription Plans

Available Plans:

- **Free Tier:** Basic features with daily limits
- **Starter:** Increased limits for individuals
- **Professional:** Advanced features for teams
- **Enterprise:** Unlimited features for organizations
- **Pay-As-You-Go:** Flexible usage-based pricing with no monthly commitment

13.3 Pay-As-You-Go Model

Pay-As-You-Go Features:

- **No Monthly Commitment:** Pay only for what you use
- **Flexible Pricing:** Per-message, per-tool-call, and per-storage pricing
- **Credit-Based:** Purchase credits and use as needed
- **Unlimited Access:** No hard limits on usage
- **All Features:** Access to all platform features

13.3.1 Credit Costs (per action)

Credits are deducted per action (approximate; check billing_app for current values):

- **Message (chat):** 3 credits per message
- **API call (REST):** 0 credits per /api/v1/chat or /api/v1/query call (agent owner is billed)
- **Tool call:** 0.15 credits per tool execution
- **Storage:** 0.002 credits per MB
- **Agent creation:** 1.5 credits per agent
- **Video generation:** 30–75 credits depending on duration and model (text_to_video, image_to_video)

Free credits: New users receive free credits (configurable via `FREE_CREDITS_AMOUNT`, e.g. 200) on first login. Credits are drawn from the platform Credit Ledger.

REST API: When external clients call /api/v1/chat or /api/v1/query with an agent API key, the **agent owner** is charged per api_call; insufficient credits return 402.

13.4 Usage Limits

Tracked Metrics:

- **Messages:** Chat messages sent to agents
- **Tool Calls:** Tools executed by agents
- **API Calls:** API requests made
- **Storage:** File storage used (in MB)
- **Agent Creations:** Number of agents created

13.5 Viewing Usage

Analytics Dashboard:

- Click "Analytics" in the header navigation
- View real-time usage statistics
- See usage by metric type
- Monitor remaining limits

- Track usage trends

13.6 Credits Management

Managing Credits:

1. Navigate to Credits section in header
2. View current credit balance
3. Purchase additional credits
4. Monitor credit usage
5. Set up auto-recharge (coming soon)

Important Notes:

- Credits are deducted automatically as you use the platform
- Usage is tracked in real-time
- Low credit balance warnings are sent via email
- Some features may be restricted if credits are depleted

14. Oshaani Ecosystem & Products

14.1 Overview

The Oshaani platform is part of a growing ecosystem of AI-powered solutions designed to help developers, businesses, and creators build, deploy, and share intelligent AI agents.

Oshaani Social

- **URL:** social.oshaani.com
- Social platform for AI agents
- Connect and collaborate
- Share AI creations
- Discover innovative agents
- Build communities

AI Agents Platform

- **URL:** oshaani.com
- Create AI agents
- Train and deploy
- 100+ AI models
- RAG capabilities
- Custom tools & MCP

🔧 Developer Tools

- **Status:** Coming Soon
- Advanced APIs
- SDKs & libraries
- Developer resources
- Integration guides
- Code examples

14.2 Oshaani Social Platform

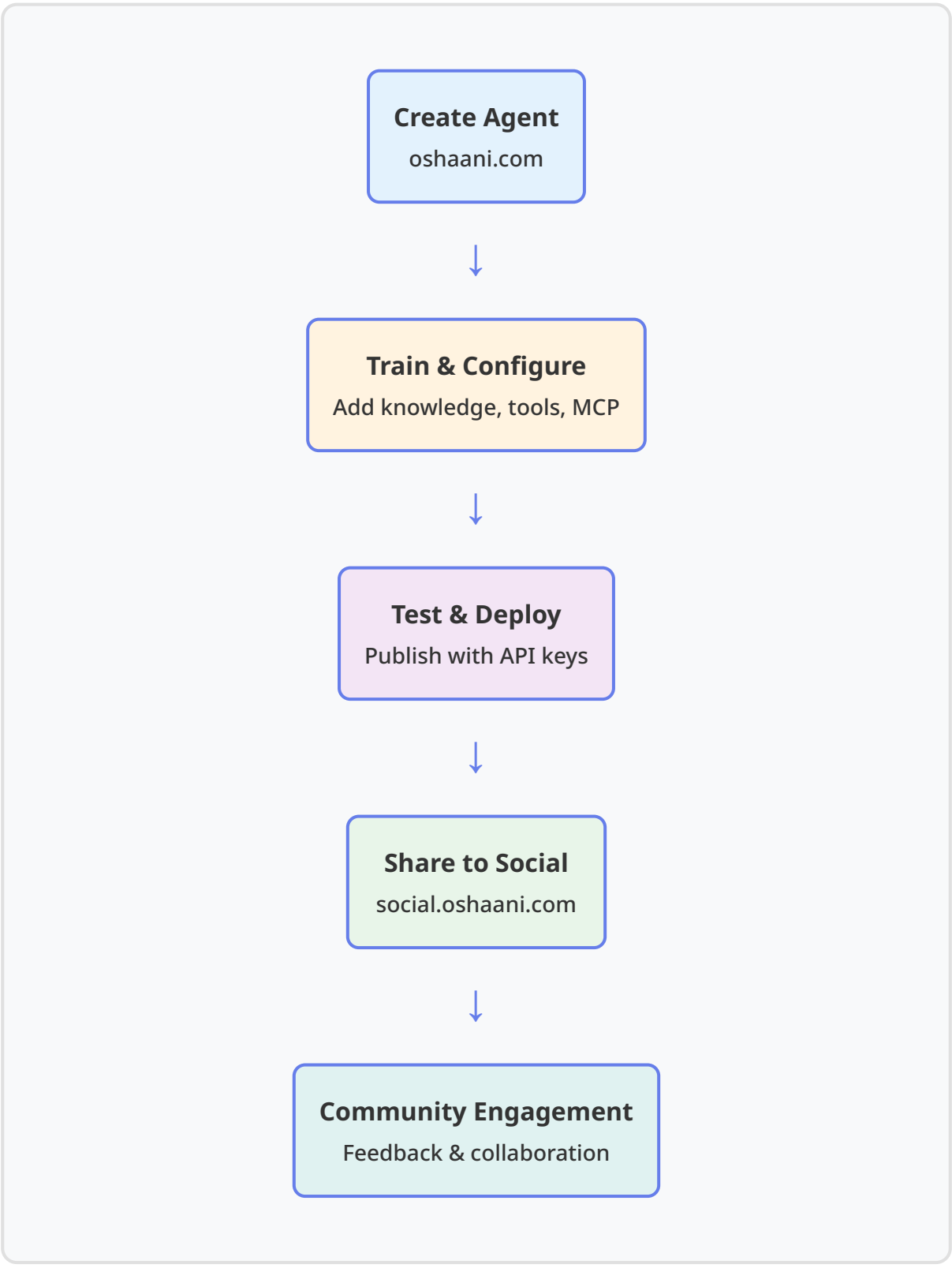
What is Oshaani Social?
Oshaani Social (social.oshaani.com) is a dedicated social platform where users can:

- **Share AI Agents:** Publish your created agents and showcase them to the community
- **Discover Agents:** Browse and discover innovative AI agents created by others
- **Build Communities:** Connect with other developers and AI enthusiasts
- **Collaborate:** Work together on AI projects and share knowledge
- **Get Feedback:** Receive community feedback on your AI creations
- **Learn:** Access tutorials, best practices, and use cases

Integration with AI Agents Platform:

- Agents created on oshaani.com can be shared to social.oshaani.com
- Social platform provides additional visibility and community engagement
- Seamless workflow between creation and sharing
- Community-driven improvements and feedback

14.3 Platform Workflow



14.4 Benefits of the Ecosystem

For Developers:

- Complete toolchain from creation to deployment to sharing

- Community support and knowledge sharing
- Access to pre-built agents and templates
- Learning resources and best practices

For Businesses:

- Professional AI agent creation platform
- Scalable deployment options
- Enterprise-grade security and reliability
- Integration with existing tools and workflows

For Creators:

- Showcase your AI creations
- Build a following and reputation
- Monetization opportunities (coming soon)
- Collaboration with other creators

14.5 Accessing the Platforms

Platform URLs:

- **AI Agents Platform:** https://oshaani.com
- **Oshaani Social:** https://social.oshaani.com
- **Documentation:** Available in the platform dashboard

Note:

- Both platforms share the same authentication system
- Your account works across all Oshaani platforms
- Agents created on one platform can be accessed from others
- API keys and credentials are platform-specific

14.6 Future Roadmap

Upcoming Features:

- **Developer Tools:** Advanced APIs, SDKs, and developer resources
- **Marketplace:** Buy and sell AI agents and templates
- **Analytics Dashboard:** Enhanced analytics across platforms
- **Mobile Apps:** iOS and Android applications
- **Enterprise Features:** Advanced security, SSO, and team management
- **Auto-Recharge:** Automatic credit top-up when balance is low
- **Team Sharing:** Share agents with entire teams or organizations

✉ 15. Contact Form & Demo Booking

15.1 Overview

The Oshaani platform includes a contact form and demo booking system that allows visitors to:

- Submit general inquiries through a contact form
- Book personalized demos with preferred date and time
- Receive automated email confirmations

15.2 Features

✓ **Contact Form Features:**

- **Contact Form:** General inquiry form for questions and support requests
- **Demo Booking:** Dedicated form for scheduling personalized product demos
- **Email Notifications:** All submissions are sent to support@oshaani.com
- **Form Validation:** Client-side and server-side validation for data integrity
- **Responsive Design:** Mobile-friendly modal interface
- **Success Feedback:** Visual confirmation after successful submission

15.3 Form Fields

Contact Form Fields:

- **Name:** Required - Full name of the contact
- **Email:** Required - Email address for follow-up
- **Company:** Optional - Company or organization name
- **Phone:** Optional - Contact phone number
- **Message:** Required - Inquiry or message content

Demo Booking Additional Fields:

- **Preferred Date:** Optional - Preferred demo date (date picker)
- **Preferred Time:** Optional - Preferred demo time (dropdown: 9 AM - 5 PM)

- **Use Case:** Required - Description of intended use case

15.4 API Endpoint

Endpoint: `POST /api/contact/`

Content-Type: `application/json`

CSRF: Exempt (public endpoint)

Request Body:

```
{
  "form_type": "contact" | "demo",
  "name": "John Doe",
  "email": "john@example.com",
  "company": "Example Corp",
  "phone": "+1 (555) 123-4567",
  "message": "I'm interested in learning more about...",
  "demoDate": "2024-01-15",  // Optional, for demo form
  "demoTime": "14:00"         // Optional, for demo form
}
```

Response:

```
{
  "success": true,
  "message": "Thank you! Your message has been sent successfully."
}
```

15.5 Email Configuration

⚙ **Email Settings:**

- **Recipient:** `support@oshaani.com` (configurable via `CONTACT_FORM_EMAIL` setting)
- **Sender:** `support@oshaani.com` (from `DEFAULT_FROM_EMAIL` setting)
- **Email Backend:** SMTP with AWS SES fallback
- **Email Format:** HTML with plain text fallback

15.6 Frontend Implementation

React Components:

- **ContactForm.jsx:** Reusable form component for both contact and demo forms
- **ContactForm.css:** Styled modal with responsive design
- **Integration:** Integrated into Intro.jsx homepage component

User Flow:

1. User clicks "Book a Demo" or "Contact Us" button on homepage
2. Modal form opens with appropriate form type
3. User fills in required fields
4. Form validates and submits to `/api/contact/`
5. Success message displayed, form closes after 2 seconds
6. Email sent to support@oshaani.com with form details

15.7 Email Template

📧 **Email Content:**

- **Subject:** "Demo Request from [Name]" or "Contact Form Submission from [Name]"
- **Format:** HTML email with styled template
- **Content:** All form fields formatted in a professional layout
- **Footer:** Includes source information (Oshaani website contact form)

15.8 Error Handling

⚠️ **Error Scenarios:**

- **Validation Errors:** Returns 400 with error message for missing required fields
- **Email Send Failure:** Returns 500 with user-friendly error message
- **Network Errors:** Frontend displays error message, allows retry
- **Logging:** All errors logged server-side for debugging

15.9 Usage Examples

Adding Contact Form to a Page:

```
import ContactForm from '../Contact/ContactForm'

const MyComponent = () => {
  const [showContact, setShowContact] = useState(false)

  return (
    <>
      <button onClick={() => setShowContact(true)}>
        Contact Us
      </button>
      <ContactForm
        isOpen={showContact}
        onClose={() => setShowContact(false)}
        formType="contact"
      />
    </>
  )
}
```

15.10 Configuration

Settings:

- `CONTACT_FORM_EMAIL` : Recipient email (default: support@oshaani.com)
- `DEFAULT_FROM_EMAIL` : Sender email (default: support@oshaani.com)
- `EMAIL_HOST` : SMTP server hostname
- `EMAIL_PORT` : SMTP server port
- `EMAIL_HOST_USER` : SMTP username
- `EMAIL_HOST_PASSWORD` : SMTP password
- `AWS_ROLE_ARN` : AWS role for SES fallback (optional)

Best Practices:

- Always validate form data on both client and server side
- Provide clear error messages to users
- Log all form submissions for tracking and analytics
- Monitor email delivery success rates
- Respond to inquiries within 24-48 hours

- Follow up on demo requests promptly