Hathr External API

Hostnames

oAuth authentication: **hathr.auth-fips.us-gov-west-1.amazoncognito.com**
API: **api.hathr.ai**

Authentication

All requests must include an oAuth 2.0 bearer token. A client ID and client secret will be provided. These should be used to obtain the bearer token. When requesting the bearer token, the scope 'hathr/llm' must be used.

Working with Documents

There are two ways to include documents in your chat requests:

Inline Documents (via content array)

Include a base64-encoded document directly in your message using the `content` array format. The entire document is included in the model's context, allowing comprehensive analysis of the full content.

Best for:

- One-off document analysis (e.g., "summarize this PDF")
- Tasks requiring full document context (e.g., comprehensive summaries, cross-referencing sections)
- Small to medium documents (up to 4.5MB)
- Documents that don't need to be reused across multiple requests

Limitations:

- Maximum 1 document per request
- Document must be re-sent with each request
- Larger payload sizes increase latency
- Uses more tokens since the entire document is in context

Uploaded Documents (via RAG)

Upload documents once via `/v1/document/upload`, then reference them by name in `/v1/document/chat`. When you send a message, a semantic search is performed within the uploaded documents and only the most relevant sections are included in the context.

Best for:

- Documents referenced repeatedly across many requests
- Large documents where you need to find specific information
- Building a persistent knowledge base
- Production applications where the same documents are queried frequently

Trade-offs:

- Requires upload and processing wait time before first use
- Only relevant chunks are included in context - may miss information if it doesn't match the search query
- Not suitable for tasks requiring full document analysis (e.g., complete summaries)

Methods

chat

Endpoint: `/v1/chat` (POST)

Makes a chat request to Hathr AI. Conversation history should be sent in the messages array, oldest first.

Request Body

Messages can be sent in two formats: the legacy text format or the new content array format.

Legacy Format (text only):

```
{
  "messages": [
    {
      "role": "user|assistant",
      "text": "string"
    }
  ],
  "temperature": "float",
  "toolConfig": {}
}
```

Content Array Format (supports inline documents):

```
{
  "messages": [
    {
      "role": "user|assistant",
      "content": [
        { "type": "text", "text": "string" },
        { "type": "document", "format": "pdf", "name": "string", "data":
"base64-encoded-string" }
      ]
    }
  ],
  "temperature": "float",
  "toolConfig": {}
}
```

- `messages`: Array of message objects (Required)
  - `role`: String - Either "user" or "assistant"
  - `text`: String - The message content (legacy format)
  - `content`: Array of content blocks (new format) - cannot be combined with `text`
    - Text block: `{ "type": "text", "text": "string" }`
    - Document block: `{ "type": "document", "format": "string", "name": "string", "data": "string" }`
      - `format`: One of: pdf, csv, doc, docx, xls, xlsx, html, txt, md
      - `name`: Alphanumeric identifier (with underscores/hyphens), 1-200 characters

        - `data`: Base64-encoded document content (max 4.5MB decoded)
    - Each message using `content` must include at least one text block
    - Document blocks are only allowed in user messages
    - Maximum 1 document per request (to stay under API Gateway size limits)
- `temperature`: Float - Controls randomness (Optional, default: 0.2, range: 0-2.0)
- `toolConfig`: Object - Configuration for tools the model can use (Optional)

```
{
  "tools": [
    {
      "toolSpec": {
        "name": "string", // Name of the tool
        "description": "string", // Description of the tool
        "inputSchema": { // JSON schema for the tool's input
          "json": {}
        }
      }
    }
  ],
  "toolChoice": { "auto": {} } // Optional: exactly one of "auto": {}, "any": {} or "tool": {"name": "string"}
}
```

Response

```
{
  "code": 200,
  "message": "Success",
  "response": {
    "usage": {
      "inputTokens": 0,
      "outputTokens": 0,
      "totalTokens": 0
    },
    "text": "string", // Response text
    "stop_reason": "string", // Reason the model stopped generating text - end_turn or tool_use
    "toolUse": { // Optional: If a tool was used by the model
      "toolUseId": "string",
      "name": "string",
      "input": {}
    }
  }
}
```

Error Responses

- `400` - Invalid request format or parameters
- `401` - Missing or invalid bearer token
- `404` - No account exists for the authenticated client
- `500` - Internal server error

upload_url

Endpoint: `/v1/document/upload` (POST)

Returns a pre-signed S3 URL for uploading documents which can be used in future chat requests.

Request Body

```
{
  "filename": "string",
  "type": "string"
}
```

- `filename`: The name of the file to be uploaded
- `type`: The MIME type of the file to be uploaded, for example application/pdf. This may be omitted only when the filename has a supported extension.

Response

```
{
  "code": 200,
  "message": "Document embedding URL created",
  "response": {
    "signedUrl": "string",
    "fileKey": "string",
    "headers": {
      "Content-Type": "application/pdf"
    }
  }
}
```

When `headers` includes `Content-Type`, send it with the S3 `PUT` request if available. The upload URL does not require `Content-Type` for signature compatibility, so clients that omit it can still upload. The document processor also inspects the uploaded bytes with `python-magic`, so extensionless files with valid supported content, such as PDFs, can still be processed.

Error Responses

- `400` - Unsupported document type, a document with this filename already exists, or the stored-document limit (40,000) has been reached
- `401` - Missing or invalid bearer token
- `404` - No account exists for the authenticated client
- `500` - Internal server error

document/complete

Endpoint: `/v1/document/complete` (GET)

Checks the status of document processing.

Response

```
{
  "code": "integer",
  "message": "string",
  "response": {
    // Processing status details
  }
}
```

- `code`: 202 while any documents are still being processed or newly uploaded files are awaiting processing
- `code`: 200 once nothing is pending (`response.state` is `done` or `error`)
- `message`: Status description
- `response.state`: `done`, `pending`, or `error`. When `error` is returned, `response.errors` contains the failed resource keys and processing messages, and `response.error_count` the number of failures.
- `response.ocr_in_progress`: Boolean - true when pending documents are undergoing OCR. In that case `response.ocr_documents` gives the count and `response.message` explains that processing may take a few minutes longer.

document/list

Endpoint: `/v1/document/list` (GET)

Retrieves a list of all available documents.

Response

```
{
```

```
  "code": "integer",
  "message": "string",
  "response": {
    "documents": [
      {
        "id": "string",
        "name": "string"
      }
    ]
  }
}
```

Each document entry includes an `id` that can be used to delete the document.

document/delete

Endpoint: `/v1/document/{documentId}` (DELETE)

Deletes the specified document and any embeddings associated with it.

Path Parameters

- `documentId`: String - Identifier returned by `/v1/document/list`

Response

```
{
  "code": 200,
  "message": "Document and embeddings associated with it have been deleted",
  "response": null
}
```

Error Responses

- `400` - Document identifier is missing
- `404` - Document not found
- `409` - Document is still processing; retry once `/v1/document/complete` reports completion

document/chat

Endpoint: `/v1/document/chat` (POST)

Makes a chat request to Hathr AI using specific documents as context (via RAG).

Request Body

Messages can be sent in two formats: the legacy text format or the new content array format.

Legacy Format:

```
{
  "documents": ["string"],
  "messages": [
    {
      "role": "user|assistant",
      "text": "string"
    }
  ],
  "temperature": "float",
  "toolConfig": {}
}
```

Content Array Format (supports inline documents):

```
{
  "documents": ["string"],
  "messages": [
    {
      "role": "user|assistant",
      "content": [
        { "type": "text", "text": "string" },
        { "type": "document", "format": "pdf", "name": "string", "data":
"base64-encoded-string" }
      ]
    }
  ],
  "temperature": "float",
  "toolConfig": {}
}
```

- `documents`: Array of document names to use as RAG context, as returned in the `name` field of `/v1/document/list` (Required)
- `messages`: Array of message objects
    - `role`: String - Either "user" or "assistant"
    - `text`: String - The message content (legacy format)

    - `content`: Array of content blocks (new format) - see `/v1/chat` for full specification
- `temperature`: Float - Controls randomness (Optional, default: 0.2, range: 0-2.0)
- `toolConfig`: Object - Configuration for tools the model can use (Optional) - see `/v1/chat` for full specification

Response

Same shape as `/v1/chat`:

```
{
  "code": 200,
  "message": "Success",
  "response": {
    "usage": {
      "inputTokens": 0,
      "outputTokens": 0,
      "totalTokens": 0
    },
    "text": "string", // Response text
    "stop_reason": "string",
    "toolUse": {} // Optional: If a tool was used by the model
  }
}
```

embeddings

Endpoint: `/v1/embeddings` (POST)

Creates an embedding for the given text content.

Request Body

```
{
  "content": "string"
}
```

- `content`: String - The text content to create an embedding for (Required)

Response

```
{
  "code": 200,
  "message": "Created embedding vectors",
  "response": [
    // Array of floats representing the embedding (1024 dimensions, normalized)
  ]
}
```

Error Responses

- `500` - Internal server error, including a missing or invalid `content` field