ChatGPT

NetSuite SuiteScript `N/llm` Module Reference

Overview of the N/llm Module

The `N/llm` SuiteScript module enables generative AI (Large Language Model, LLM) capabilities in NetSuite scripts [1]. It allows SuiteScript (server-side) code to send prompts to supported large language models and receive AI-generated text responses for use in custom automations [1]. Key features include: conversational chat support (with message history), retrieval-augmented generation using reference documents, and even image-based question answering (with vision-capable models).

Availability & Usage Modes: LLM support is currently available only in certain NetSuite data center regions [2]. By default, accounts can use a limited **free usage pool** of LLM calls per month (NetSuite currently allocates ~**1,000 free LLM requests per month** per account [3]). Once the free monthly quota is exhausted – or if the script is running from a managed SuiteApp (which *cannot* use the free pool) – you must configure an Oracle Cloud Infrastructure (OCI) Generative AI Service for **unlimited usage mode** [4] [5]. In unlimited mode, you supply OCI credentials (either via script parameters or via the SuiteScript AI Preferences in NetSuite) to route LLM requests through your Oracle Cloud account [5] [6]. SuiteApps installed in target accounts are **prevented from using the free usage pool** and must use OCI configuration for LLM access [4]. All LLM requests consume script governance units (see **Governance** below) and have a default timeout of 30 seconds if not specified [7].

Supported Models: As of 2024.2+, NetSuite supports a few model families (via the `llm.ModelFamily` enum): the default Cohere Command models and an Oracle-provided Meta Llama model that supports vision (image input) [8] [9]. By default, if no model is specified, the system uses the latest Cohere *Command-R* model [10]. Other models (e.g. OpenAI GPT) are not directly available; developers must use the provided model options.

Script Types: The `N/llm` module is available to server-side SuiteScript 2.x scripts (Suitelet, Map/Reduce, Scheduled, Workflow actions, etc.), using SuiteScript 2.1 syntax [11]. It is not supported in client-side (browser) scripts.

Enumerations

`llm.ChatRole` (Enum)

Represents the author role of a chat message in a conversation. Use this to specify whether a message is from the user (prompt) or the AI chatbot (response) [12] [13].

- **Values:**
- `llm.ChatRole.USER` – Indicates the message is from the user (the prompt/question) [14].
- `llm.ChatRole.CHATBOT` – Indicates the message is from the AI/assistant (the LLM’s response) [15].

These values are used when creating chat messages (see `llm.createChatMessage`) to build a chat history. For example, a user prompt would have role USER, and the LLM reply would have role CHATBOT [13].

`llm.ModelFamily` **(Enum)**

Defines which large language model to use for a request [16]. This enum's values correspond to specific model families/versions supported by NetSuite.

- **Values:**
- `llm.ModelFamily.COHERE_COMMAND_R` – Uses the latest supported Cohere *Command-R* model (this is the **default** if no modelFamily is specified) [10].
- `llm.ModelFamily.COHERE_COMMAND_R_PLUS` – Uses the latest Cohere *Command-R Plus* model (a larger variant of the Cohere model) [17].
- `llm.ModelFamily.META_LLAMA` – Uses the latest Meta **Llama 3.2 90B Vision** model (supports image input in prompts) [9].

All values are strings under the hood. For example, `COHERE_COMMAND_R` corresponds to `"cohere.command-r-08-2024"` (always mapping to the latest version) [10].

Note: If you use a model that does not support a given feature (e.g. using `META_LLAMA` but providing a preamble, which only Cohere supports), the API will throw an error – see error codes like `MODEL_DOES_NOT_ACCEPT_PREAMBLE` below [18].

Object Types (Data Structures)

`llm.ChatMessage` Object

Represents a single chat message in a conversation (either a user prompt or an AI response). A ChatMessage has the following properties:

- `role` (string): The role/author of the message – typically `llm.ChatRole.USER` for user prompts or `llm.ChatRole.CHATBOT` for LLM-generated messages [19] [13].
- `text` (string): The content of the message (the actual text) [20].

ChatMessage objects are primarily used when supplying a `chatHistory` to the LLM (see `llm.generateText` methods). You can create them using `llm.createChatMessage()` or by constructing an object with the required structure. For example, to maintain context in a conversation, you would push user and chatbot messages into an array of ChatMessages, preserving the order of dialogue.

`llm.Document` Object

Represents a reference document used for retrieval-augmented generation (RAG). Each Document contains:

- `id` (string): An identifier for the document [21]. This can be any unique string you choose (e.g. `"doc1"`, `"KB_article_123"`). Documents in an array must have unique IDs (the call will error if there are duplicates) [22].

- `data` (string): The content of the document (text to be used as reference) [23].

Documents are provided to the LLM via the `documents` parameter in `llm.generateText`/ `llm.generateTextStreamed`. The LLM can then use these documents' content to ground its response (for instance, answering questions based on the provided text) [24]. This mechanism is often called retrieval-augmented generation. If documents are used, the LLM may return **citations** indicating which document and snippet were used in forming the answer.

> **Note:** You can create Document objects using `llm.createDocument()`, but this is not strictly required – you may also directly pass plain JavaScript objects with `id` and `data` fields in the `documents` array [25].

`llm.Citation` Object

Represents a citation/reference in the LLM's response, used when retrieval-augmented generation is employed. A Citation object includes:

- `documentIds` (string[]): The IDs of the document(s) from which the cited text originates [26]. (Typically this is a single ID, but the array allows multiple if the excerpt spans documents.)
- `start` (number): The starting character index of the cited text within the source document [27].
- `end` (number): The ending character index of the cited text within the source document [27].
- `text` (string): The exact text excerpt from the document that was used or quoted [28].

These citations are returned as part of the LLM response to help identify which parts of which documents were relevant. You can use the `documentIds` along with `start`/`end` to highlight or display the source content to end-users.

`llm.Response` Object

The Response object represents a full response returned by the LLM for non-streaming calls. It contains:

- `text` (string): The complete text output from the LLM (the answer or completion) [29].
- `model` (string): The identifier of the model that generated this response [30] (e.g., `"cohere.command-r-08-2024"` or similar).
- `documents` (llm.Document[]): The list of Document objects that were provided as context when generating this response [31]. (This mirrors what you passed in the request; useful if you need to know which docs were used.)
- `citations` (llm.Citation[]): An array of Citations highlighting which parts of the documents were used for the response (if any) [31]. This may be empty if no docs or no specific snippet was cited.
- `chatHistory` (llm.ChatMessage[]): The chat message history relevant to this response [32]. Typically, if you provided a `chatHistory` plus a new prompt, this array will include those and possibly the latest question/answer. Essentially, it can contain the prompt as a USER message and the response as a CHATBOT message. (This can be used to continue the conversation in subsequent calls by feeding it back in.)

`llm.StreamedResponse` **Object**

The StreamedResponse object is similar to Response, but is returned by the streaming API variants (`llm.generateTextStreamed`). It allows you to iterate through the response as it is being generated. Its properties are mostly the same as Response:

- `text` (string): The content generated so far (this string accumulates tokens as the LLM produces output) [33] . Once the stream is complete, this will contain the full response text.
- `model` (string): The model identifier used for the generation [34] .
- `documents` (llm.Document[]): Any Document objects that were provided as context (same as in Response) [35] .
- `citations` (llm.Citation[]): Any citations returned (same as in Response) [36] .
- `chatHistory` (llm.ChatMessage[]): The chat history messages (similar to Response) [37] .

In addition, StreamedResponse provides a way to retrieve tokens incrementally: - `iterator()` – Returns an `Iterator` object that can be used to step through each token of the response stream as it arrives [38] [39] . You can call `iter = streamedResponse.iterator()` and then use `iter.each(function(token){ ... })` to process tokens one by one. Each `token` has a `.value` property containing the token text [40] . During iteration, the `streamedResponse.text` property is updated to include all tokens up to that point [41] . This allows, for example, logging partial output or sending real-time updates to a user interface while the LLM is still generating the rest of the text.

Usage: The StreamedResponse is useful for handling long responses or providing feedback as the response is being generated. Keep in mind that using the iterator will block until the LLM has finished (it processes tokens in sequence). If you do not iterate at all, the StreamedResponse's `text` will contain the full reply once generation completes (just like a normal Response, but you would have waited until the end without processing intermediate tokens).

Methods

Below are the functions provided by the `N/llm` module, including their purpose, parameters, return values, governance limits, and relevant error codes. All methods below are available only in server-side SuiteScript 2.x (2.1) scripts.

`llm.createChatMessage(options)`

Description: Creates a new chat message object, given a role and text [42] . This is a convenience method to build a `llm.ChatMessage` which can be included in a chat history array for conversation context. Supported roles for the message are defined by the `llm.ChatRole` enum (USER or CHATBOT) [42] .

- **Returns:** a `llm.ChatMessage` object representing the message [43] [44] .
- **Governance:** *None* (no usage units consumed) [45] .
- **Since:** 2024.1 [46] .

Parameters: (All properties of the `options` argument)
- `options.role` (`string`, **required**): The author role of the message (who is speaking). Use `llm.ChatRole` values – e.g. `llm.ChatRole.USER` for a user prompt, or `llm.ChatRole.CHATBOT` for

an AI response [47].
- `options.text` (`string`, **required**): The text content of the message [48].

There are no special error conditions for this method (it will throw a general error if required fields are missing). Once created, the returned ChatMessage can be added to an array and passed as `chatHistory` to an LLM generation call.

`llm.createDocument(options)`

Description: Creates a new document object with a specified ID and content [49]. This is used to supply reference text (e.g. knowledge base articles, FAQs, etc.) to the LLM so it can perform retrieval-augmented generation. The returned `llm.Document` can be included in the `documents` parameter of LLM calls.

You can use this method to pre-construct Document objects, but it's optional – the LLM functions will accept plain `{id, data}` objects as well [25]. The purpose is to ensure your documents have the proper structure or to reuse documents across multiple requests.

- **Returns:** a `llm.Document` object with the given id and data [50] [51].
- **Governance:** *None* (no usage cost) [52].
- **Since:** 2025.1 [53].

Parameters:
- `options.id` (`string`, **required**): The unique identifier for the document [54]. This ID is used by the LLM to refer to the document (for example, in citations). Choose a descriptive ID if possible. *Important:* All documents in a single call must have distinct IDs – if there are duplicates, the call will throw an error `DOCUMENT_IDS_MUST_BE_UNIQUE` [22]. - `options.data` (`string`, **required**): The content of the document (the text body) [55]. This should be a reasonably sized string (to fit within token limits of the models). There is no fixed length limit per document mentioned, but the total tokens of all documents + prompt must stay within model limits.

No special errors are documented for `createDocument` itself aside from general argument validation. If successfully created, you can pass the resulting Document in the `documents` array to `llm.generateText` or `llm.generateTextStreamed` calls. Providing documents enables the LLM to **cite and use their content** when formulating a response [24].

`llm.generateText(options)`

Description: Sends a prompt to the LLM and returns the generated response [56]. This is the primary method to get a text completion or answer from the AI. You can include various parameters to shape the AI's response (such as model choice, temperature, etc.) and to provide context (previous chat messages or reference documents). The method returns once the LLM has finished generating the complete response.

- **Returns:** a `llm.Response` object containing the LLM's response text (and any associated metadata like citations) [57].
- **Governance:** 100 units per call [58]. This is a relatively high usage cost, so each call counts significantly against script usage limits. (Both synchronous and promise versions consume the same 100 units [59].)

- **Since:** 2024.1 [60].
- **Supported Scripts:** Server-side scripts only [11].

Aliases: `llm.chat(options)` is an alias for this method [61]. It has identical behavior and parameters, just a different name.

Parameters: (All properties of the `options` object are listed below)

- `options.prompt` (`string`, **required**): The prompt or question to send to the LLM [62]. This is the main user query or instruction that you want the AI to respond to. *(Ensure this string is within a reasonable length, as very large prompts count toward token limits.)*

- `options.chatHistory` (`llm.ChatMessage[]`, optional): An array of prior chat messages (conversation history) to provide context [63]. Include recent user and AI messages here if you want the LLM to continue a conversation or have awareness of previous exchanges. Each entry should be a ChatMessage object (with role and text). If not provided, the LLM will treat the prompt as a single-turn query with no context aside from the prompt itself.

- `options.documents` (`llm.Document[]`, optional): A list of Document objects to use as additional context for the LLM [64]. These documents enable retrieval-augmented generation: the LLM can draw facts from the documents and may provide citations. If provided, the LLM will consider the content of these documents when formulating its answer (for example, answering questions about their content). This is useful for grounding the AI's response in factual or custom data. *(This feature was added in NetSuite 2025.1)* [65]. Remember each document needs a unique `id`. Models that do not support RAG will throw an error if documents are supplied (see error `MODEL_DOES_NOT_ACCEPT_DOCUMENTS`) [66].

- `options.image` (`file.File`, optional): An image file object to send to the LLM for analysis [67]. This allows you to ask questions about the content of an image or request descriptions of the image. For example, you could prompt: "What is described in this image?" and supply an image of a chart or picture. **Note:** Image processing is only supported by the Meta Llama vision model (`ModelFamily.META_LLAMA`) [68]. If you provide an image with a model that doesn't support it, the call will fail with `MODEL_DOES_NOT_ACCEPT_IMAGE` [69]. To use this, create or load a NetSuite file (N/file module) and pass it in. The LLM can return: detailed captions, answers about the image's content, or information extracted from images (like reading charts) [70] [71].

- `options.modelFamily` (`enum`, optional): Specifies which LLM model to use [72]. Use one of the values from the `llm.ModelFamily` enum, such as `llm.ModelFamily.COHERE_COMMAND_R` or others. If omitted, the system will use the default model (Cohere Command-R) [73]. Provide this if you want to select a non-default model (for example, the Llama model for image support, or the Command-R Plus for potentially higher quality). *(Support for specifying modelFamily was added in 2024.2)* [74].

- `options.modelParameters` (`Object`, optional): A set of generation parameters to customize the LLM's output [75]. These allow tuning the "personality" or format of the response. If not provided, model-specific defaults are used. Recognized fields within this object include:

- `maxTokens` (`number`, optional): The maximum number of tokens the LLM is allowed to generate [76]. This caps the length of the response. (Note: roughly 3 tokens ≈ 1 word [77].) Each model has limits on `maxTokens` (e.g., the sum of prompt+output tokens must be ≤ model's context length). If you set a value not allowed by the model, an error occurs (`INVALID_MAX_TOKENS_VALUE`) [78].
- `temperature` (`number`, optional): Controls randomness/creativity in output [79]. Range is typically 0 to 1 (sometimes higher). Lower values make output more deterministic and focused (good for factual answers), higher values produce more varied or creative text [80]. An invalid value (e.g. out of range) will throw `INVALID_TEMPERATURE_VALUE` [78].
- `topK` (`number`, optional): Limits consideration to the top K most likely tokens at each generation step [81]. For example, `topK: 3` means the next word is chosen from the 3 most probable options. This is an alternate sampling control to use alongside or instead of `topP`. (If an invalid value is given, you get `INVALID_TOP_K_VALUE`) [82].
- `topP` (`number`, optional): Limits consideration to tokens that cumulatively represent probability `p` at each step [83]. For instance, `topP: 0.7` considers the smallest set of tokens whose probabilities add up to 70%. This can be used with or instead of `topK`. If both are set, the model will apply `topK` first then `topP` on that subset [84]. (Invalid values throw `INVALID_TOP_P_VALUE`) [82].
- `frequencyPenalty` (`number`, optional): A value (usually 0 to 1 or 2) that penalizes tokens that appear frequently in the generated text [85]. This discourages the LLM from repeating the same lines or words. A higher number means more penalty for repetition of any token proportional to frequency [86]. Use either this or `presencePenalty` as needed (see note below). An invalid value yields `INVALID_FREQUENCY_PENALTY_VALUE` [87].
- `presencePenalty` (`number`, optional): A penalty to discourage *any* usage of tokens that have already appeared, regardless of frequency [88]. This encourages the model to bring in new words not seen before. It's similar to frequencyPenalty but treats all prior occurrences the same (not weighted by count) [89]. Do not use a non-zero presencePenalty *together with* a non-zero frequencyPenalty on the Cohere models – they are mutually exclusive in that context [90]. If both are set > 0 for Cohere, the call throws `MUTUALLY_EXCLUSIVE_ARGUMENTS` (you must pick one form of penalty) [90]. An invalid numeric value will throw `INVALID_PRESENCE_PENALTY_VALUE` [87].

Note: The valid ranges for these parameters depend on the model. If you supply a parameter value outside the allowed range or of the wrong type, the call may throw an error like `INVALID_*_VALUE` (see Errors below) [78] [82]. For details on recommended or max values, see Oracle's OCI Generative AI documentation on model parameters [91].

- `options.ociConfig` (`Object`, optional): OCI configuration credentials, required only for unlimited usage mode (bypassing the free tier) [5] [92]. In most cases you will either set these here or (preferably) configure them once in the "AI Preferences" in NetSuite UI [6] [93]. If both are provided, the explicit values here override the account-level preferences [93]. This object's fields correspond to your Oracle Cloud tenancy and user API key details:
- `tenancyId` (`string`): Your Oracle Cloud Tenancy OCID [94].
- `userId` (`string`): The Oracle Cloud User OCID that the API key belongs to [95].
- `compartmentId` (`string`): The OCID of the compartment in OCI where the AI service is enabled [96].
- `endpointId` (`string`, optional): The OCI Generative AI **Endpoint ID** to use (if you have a dedicated AI endpoint/cluster) [97]. Most use-cases won't require this unless using a custom deployment.

- `fingerprint` (`string`, **required if ociConfig is used**): The fingerprint of the public key for your OCI API key [98]. **Important:** This must be provided as a NetSuite Secret ID (e.g., `custsecret_mykey_fingerprint`), not the raw fingerprint itself [98]. If you put a raw fingerprint here, NetSuite will throw `ONLY_API_SECRET_IS_ACCEPTED` error – it expects a reference to a stored secret.
- `privateKey` (`string`, **required if ociConfig is used**): The *private key* for your OCI API user, which must also be stored as a NetSuite Secret and referenced by its ID (e.g., `custsecret_mykey_private`) [99]. Do not embed the raw private key text here – that will trigger the `ONLY_API_SECRET_IS_ACCEPTED` error [100]. Instead, create a Secret (with scope = RESTlet/ SuiteScript) containing the key and use its script ID.

These credentials are needed only if you want to use your Oracle Cloud account for LLM requests (unlimited mode) or if the free tier is not available. If using the free tier, you can omit `ociConfig`. If using a SuiteApp or hitting the free limit, you must supply these. NetSuite will handle signing and routing the request to OCI. Unrecognized fields in this object will cause `UNRECOGNIZED_OCI_CONFIG_PARAMETERS` error [101] [102].

- `options.preamble` (`string`, optional): A preamble string to prime the model with an initial context or persona before the prompt [103]. This is like a system message or instruction that influences the style/tone of the response. For example, you might set: *"You are a helpful financial advisor. Answer in a formal tone."* – then the prompt question follows. **Note:** Only certain models support a custom preamble. In particular, this works for Cohere Command models, but **Meta Llama does not support preamble** overrides [104]. If you supply a preamble with a model that doesn't accept it, the call throws `MODEL_DOES_NOT_ACCEPT_PREAMBLE` [18]. Also, this parameter is ignored for models that inherently have a fixed system prompt. (For Cohere, the documentation notes it's valid to use preamble for customizing behavior.)

- `options.timeout` (`number`, optional): A timeout in milliseconds for the LLM request [7]. Defaults to **30000 ms (30 seconds)** if not set [7]. If the LLM does not produce a result in this time, the call may be terminated with an error. You can increase this if expecting long responses, but be mindful of script governance and user experience. The maximum allowed timeout may be capped by NetSuite.

Errors: The `llm.generateText` call can throw a variety of errors if inputs are missing, invalid, or inconsistent. Below are the specific error codes and their meanings:

- `SSS_MISSING_REQD_ARGUMENT` – Thrown if a required argument is missing. For example, if you do not provide the `options.prompt` (prompt is required) [105]. *(Meaning: You must include all required parameters, e.g. ensure the prompt is provided.)*

- `MUTUALLY_EXCLUSIVE_ARGUMENTS` – Thrown if two parameters that cannot be used together are both set [90]. This specifically happens if **both** `presencePenalty` and `frequencyPenalty` are set to non-zero values while using the Cohere Command model (which doesn't allow combining those penalties) [90]. *(Meaning: You should use either frequencyPenalty or presencePenalty, not both, for Cohere models.)*

- `UNRECOGNIZED_MODEL_PARAMETERS` – Thrown if one or more unrecognized/unexpected fields are present in `options.modelParameters` [101]. *(Meaning: You included a parameter in the* `modelParameters` *object that the system doesn't support or recognize – check for typos or unsupported parameter names.)*

- `UNRECOGNIZED_OCI_CONFIG_PARAMETERS` – Thrown if one or more unrecognized fields are present in `options.ociConfig` [102]. *(Meaning: Your OCI config object has an invalid property name – check that you only use the documented keys like userId, tenancyId, etc.)*

- `ONLY_API_SECRET_IS_ACCEPTED` – Thrown if the `ociConfig.privateKey` or `ociConfig.fingerprint` is not provided as a NetSuite API secret ID [100]. *(Meaning: You passed the OCI key or fingerprint directly as text, which is not allowed. You must store these in NetSuite's Secrets and use the secret IDs instead.)*

- `INVALID_MODEL_FAMILY_VALUE` – Thrown if `options.modelFamily` is set to a value that is not valid [106]. This could happen if there's a typo or an unsupported model name. *(Meaning: The specified model family is not recognized. Use the* `llm.ModelFamily` *enum values.)*

- `MODEL_DOES_NOT_ACCEPT_PREAMBLE` – Thrown if you provided a `preamble` but the chosen model does not support preambles (for example, you set a preamble while using the Meta Llama model, which doesn't allow it) [18]. *(Meaning: Remove the preamble or switch to a model that supports it, like Cohere.)*

- `MODEL_DOES_NOT_ACCEPT_DOCUMENTS` – Thrown if you provided `documents` but the model doesn't support retrieval-augmented generation [66]. Currently, Cohere models support RAG, but if this error appears it likely means the model in use (e.g. a hypothetical model that doesn't do RAG) cannot utilize documents. *(Meaning: The model cannot use the provided documents – remove* `documents` *or use a model that supports RAG.)*

- `MODEL_DOES_NOT_ACCEPT_IMAGE` – Thrown if `options.image` is provided but the model doesn't support image inputs [69]. (For instance, you tried to send an image to Cohere, which can't process images.) *(Meaning: Only the Meta Llama vision model can handle images. Use that model or remove the image parameter.)*

- `DOCUMENT_IDS_MUST_BE_UNIQUE` – Thrown if you provided multiple documents with the same `id` in the `options.documents` array [22]. *(Meaning: Ensure each Document has a unique id string – no duplicates.)*

- `INVALID_MAX_TOKENS_VALUE` – Thrown if the `modelParameters.maxTokens` value is not acceptable for the model [78]. Each model has a maximum context length and possibly specific allowed ranges for maxTokens. *(Meaning: The maxTokens you set is too high, too low, or otherwise not allowed. Adjust it according to model limits.)*

- `INVALID_TEMPERATURE_VALUE` – Thrown if the `modelParameters.temperature` value is out of range or invalid [78]. *(Meaning: Temperature must typically be between 0 and 1 (or in some cases up to, say, 2). Use a valid value.)*

- `INVALID_TOP_K_VALUE` – Thrown if `modelParameters.topK` is set to an invalid value for the model [82]. *(Meaning: The topK value might be negative or beyond what the model expects. Use a non-negative integer, usually within a reasonable range.)*

- `INVALID_TOP_P_VALUE` – Thrown if `modelParameters.topP` is set to an invalid value [82]. *(Meaning: TopP should be between 0 and 1 (probability). Check that value.)*

- `INVALID_FREQUENCY_PENALTY_VALUE` – Thrown if the `modelParameters.frequencyPenalty` value is invalid or out of range [87]. *(Meaning: Check the allowed range for frequencyPenalty for the model, e.g., 0.0 to 1.0, and provide a valid number.)*

- `INVALID_PRESENCE_PENALTY_VALUE` – Thrown if the `modelParameters.presencePenalty` value is invalid or out of range [87]. *(Meaning: Check the allowed range for presencePenalty and provide a correct value.)*

- `MAXIMUM_PARALLEL_REQUESTS_LIMIT_EXCEEDED` – Thrown if you attempt to execute more than 5 LLM requests in parallel [107]. NetSuite limits concurrency for LLM calls to at most 5 at the same time. *(Meaning: Too many LLM calls are running concurrently. Ensure no more than 5 parallel calls; otherwise queue or throttle additional requests.)*

Usage Notes: Each call to `llm.generateText` consumes a significant amount of governance units (100). Use them judiciously, and monitor `Script.getRemainingUsage()` if needed. You can also check how many free calls remain in the month using `llm.getRemainingFreeUsage()` (see below) if operating in free mode. If the free quota is exhausted (or not available), make sure to configure `ociConfig` (or set up the AI Preferences) to avoid errors. The response from this function includes the text and possibly citations. If you provided a `chatHistory`, you might want to append the new question and answer to that history for future calls (the returned Response's chatHistory can be used for this).

Asynchronous Usage: If your script supports promises (SuiteScript 2.1), you can use the promise variant: `llm.generateText.promise(options)`. This returns a JavaScript Promise that resolves to the same `llm.Response`. Internally, it still consumes 100 governance units, but it allows other code to run while waiting. Usage example:

```
llm.generateText.promise({ prompt: "Hello" }).then(response => {
    // handle response
});
```

This can be helpful in SuiteScript Map/Reduce or other contexts. The parameters and errors for the promise version are identical [108].

`llm.generateTextStreamed(options)`

Description: Similar to `llm.generateText`, but returns a **streamed** response object that you can use to retrieve the output incrementally [109]. This allows you to start processing the generated text before the entire response is finished. Essentially, the LLM begins streaming tokens back as they are generated. This method is useful for handling long responses or giving real-time feedback (for example, writing output progressively to a file or UI).

- **Returns:** a `llm.StreamedResponse` object [38] [110]. You can use this object's `iterator()` to iterate over tokens as described in the StreamedResponse section. The StreamedResponse also ultimately contains the full text once generation is done (in its `text` property).
- **Governance:** 100 units per call (same as non-streamed) [111].
- **Since:** 2025.1 [112].
- **Supported Scripts:** Server-side only (SuiteScript 2.1).

Aliases: `llm.chatStreamed(options)` is an alias name for this method [113], behaving identically.

Parameters: The input `options` for `generateTextStreamed` are **the same as for** `llm.generateText` (described above). All the same fields are supported: `prompt` (required) [114], optional `chatHistory` [115], `documents` (since 2025.1) [116], `image` (since 2025.1) [117], `modelFamily` [118], `modelParameters` (with all sub-parameters) [119] [120], `ociConfig`, `preamble`, and `timeout`. The constraints and behaviors of these parameters are exactly as documented for `generateText` above. (If a parameter is invalid or not supported by the model, the same error will occur.) For example, streaming an image query requires using the Meta Llama model as well, and streaming with an unsupported parameter will throw the same errors.

Because the parameters are identical in meaning, refer to the explanations under `llm.generateText` for details on each. Notably, `options.timeout` (default 30s) applies to the streaming call as well [121].

Using the Stream: After calling `llm.generateTextStreamed(options)`, you will immediately get back a StreamedResponse object. You can then do something like:

```
let resp = llm.generateTextStreamed({ prompt: "Your prompt here", ... });
resp.iterator().each(function(token) {
   // This callback runs for each token as it arrives
   log.debug("Got token:", token.value);
   log.debug("Current text so far:", resp.text);
   return true; // continue iteration
});
```

In the above pattern, the `.each` will iterate through each generated token in sequence [40]. Inside the loop, `token.value` is the latest token, and `resp.text` accumulates the text up to that point [41]. This allows processing or outputting partial results (e.g., streaming to client via response or writing gradually to a file). The iteration continues until the LLM indicates completion. If you break out of the loop early (by returning false), you could theoretically stop reading further tokens (though the generation might still complete in the background).

If you prefer not to handle token-by-token, you can also simply wait until the generation is done and then use `resp.text` for the full output (just as you would with a normal Response). The streaming call is especially useful when integrated with long-running Map/Reduce or Suitelet that can flush output to a client as it arrives.

Errors: `llm.generateTextStreamed` can throw all the **same errors as** `llm.generateText`, since it does all the same work, just in a streaming fashion [122]. This means the error codes listed above (missing prompt, invalid parameters, model limitations, etc.) all apply here too [123] [124] [125]. In addition, the same parallel limit of 5 concurrent calls applies (`MAXIMUM_PARALLEL_REQUESTS_LIMIT_EXCEEDED`) [107]. Ensure to handle these errors similarly. Once you have a StreamedResponse, using the iterator does not produce further special error codes – any issues (like a dropped connection to the AI service) would likely manifest as a thrown error during iteration, but such low-level errors are not explicitly documented.

Asynchronous Usage: There is also a promise version: `llm.generateTextStreamed.promise(options)`, which returns a Promise that resolves to a StreamedResponse. This can be used if you want to `await` the streamed result. However, note that even with a promise, you would still need to iterate through tokens on the resolved StreamedResponse to actually consume the data. The promise just offloads the waiting for the initial response object, not the token streaming itself.

`llm.getRemainingFreeUsage()`

Description: Returns the number of free LLM requests remaining in the current month for the account [126]. This is useful to programmatically check how many free uses of the generative AI you have left. If the account is in unlimited/OCI mode (or in a SuiteApp that can't use free requests), this number may always be 0 or irrelevant.

- **Returns:** a `number` – the count of free requests still available this month [126]. For example, if the account has a limit of 1000 and you've used 25, this might return 975.
- **Governance:** *None* (checking usage does not consume any units) [127].
- **Since:** 2025.1 (feature introduced alongside the AI services).

Parameters: None. Simply call `llm.getRemainingFreeUsage()` with no arguments.

This function does not throw documented special errors (beyond general issues). It should return an integer. You might use it to decide whether to route a request to OCI (if free usage is depleted) or to simply log remaining usage for analytics. Keep in mind the count resets each month (on the first day of the month, presumably).

An asynchronous version is available as `llm.getRemainingFreeUsage.promise()`, which returns a Promise for the number [128], but in practice this is rarely needed due to the simplicity/quickness of the call.

Note: There is a corresponding `llm.getRemainingFreeEmbedUsage()` for the free embedding request count [129], but since embedding methods are outside our current scope, it's not detailed here.

Governance and Limits

- **Usage Units:** Each call to generate text (or evaluate a prompt) costs 100 governance units [58] [59]. This is a significant cost – scripts are limited in total units (e.g., 1000 units per scheduled script execution). Plan usage accordingly (for instance, avoid calling in tight loops without yield). Simple helper calls like creating chat messages/documents or checking usage are free (0 units) [45] [52].

- **Free vs Paid Mode:** NetSuite provides a free allocation of LLM calls per month (approximately 1000 calls as of 2025.2 [3]). These reset monthly and are shared by all scripts in the account. When using free mode, responses are generated by NetSuite's provided service endpoints. If you exceed this, further calls will likely fail or be blocked unless you switch to unlimited mode with OCI. NetSuite doesn't charge script usage units for the calls themselves beyond governance, but after the free quota you *must* have an OCI setup (which may have its own costs through Oracle Cloud).

- **Parallel Calls:** You can only have **5 LLM requests in flight at the same time** per account [107]. The 6th simultaneous call will immediately throw `MAXIMUM_PARALLEL_REQUESTS_LIMIT_EXCEEDED`. "In flight" means the script has initiated the call and not yet received a result. For example, if using promises or asynchronous contexts, be careful not to fire off too many at once. Throttling or queuing might be necessary for high-volume scenarios.

- **Timeouts:** By default, each call times out after 30 seconds if no response (you can adjust `options.timeout` up to some limit) [7]. A call that times out likely still consumes the governance units. If you anticipate long responses, consider using the streaming API, which allows you to get partial data earlier and possibly handle longer generation times by processing tokens as they come.

- **Region Limitations:** As noted, only certain data center regions have the generative AI feature enabled [2]. If you attempt to use `N/llm` in an unsupported region, you may get an error indicating the module or feature is unavailable. Check Oracle's documentation for "Generative AI Availability in NetSuite" for the list of supported regions [2].

- **Model and Feature Support:** Not all models support all features (e.g., only Llama supports vision, only Cohere supports preamble customization). Use the `ModelFamily` appropriate for your use case and handle errors for unsupported parameters. In future releases, Oracle may add more models or capabilities, so keep an eye on release notes for new `ModelFamily` values or changed limits.

By understanding these methods, parameters, and limits, developers can integrate powerful AI features into NetSuite customizations, while providing meaningful error messages to end users when things go wrong. Always handle exceptions from these calls and consider logging `Response.model` and other metadata for auditing how AI is used in your account. With proper use of context (chat history, documents) and configuration, the `N/llm` module can enable advanced AI-driven functionality in the NetSuite platform.

Sources: The above information is compiled from the official Oracle NetSuite Help Center documentation on the N/llm module and related SuiteScript API references [130] [56] [24] [13] (Oracle, SuiteCloud Platform – SuiteScript 2.x API). All parameters, return types, and error codes have been documented to assist with accurate implementation of NetSuite AI features.

---

[1] [2] [19] [20] [21] [23] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [108] [126] [129] [130] NetSuite Applications Suite - N/llm Module
https://docs.oracle.com/en/cloud/saas/netsuite/ns-online-help/article_9123730083.html

[3] SuiteAPI | Documentation
https://suiteapi.com/documentation/

[4] [5] [7] [11] [18] [22] [56] [57] [58] [60] [61] [62] [63] [64] [65] [66] [67] [68] [69] [70] [71] [72] [73] [74] [75] [76] [77] [79] [80] [81] [83] [84] [85] [86] [88] [89] [90] [91] [92] [94] [95] [96] [97] [98] [99] [100] [101] [102] [103] [104] [105] [106] NetSuite Applications Suite - llm.generateText(options)
https://docs.oracle.com/en/cloud/saas/netsuite/ns-online-help/article_1014032554.html

[6] [93] NetSuite Applications Suite - llm.evaluatePrompt(options)
https://docs.oracle.com/en/cloud/saas/netsuite/ns-online-help/article_0115064704.html

[8] [9] [10] [16] [17] NetSuite Applications Suite - llm.ModelFamily
https://docs.oracle.com/en/cloud/saas/netsuite/ns-online-help/article_1014101247.html

[12] [13] [14] [15] NetSuite Applications Suite - llm.ChatRole
https://docs.oracle.com/en/cloud/saas/netsuite/ns-online-help/article_1015044805.html

[24] [25] [49] [50] [51] [52] [53] [54] [55] NetSuite Applications Suite - llm.createDocument(options)
https://docs.oracle.com/en/cloud/saas/netsuite/ns-online-help/article_79091440431.html

[38] [39] [40] [41] [78] [82] [87] [107] [109] [110] [111] [112] [113] [114] [115] [116] [117] [118] [119] [120] [121] [122] [123] [124] [125] NetSuite Applications Suite - llm.generateTextStreamed(options)
https://docs.oracle.com/en/cloud/saas/netsuite/ns-online-help/article_46075557997.html

[42] [43] [44] [45] [46] [47] [48] NetSuite Applications Suite - llm.createChatMessage(options)
https://docs.oracle.com/en/cloud/saas/netsuite/ns-online-help/article_1014104320.html

[59] [127] [128] NetSuite Applications Suite - SuiteScript 2.x API Governance
https://docs.oracle.com/en/cloud/saas/netsuite/ns-online-help/section_157072844224.html