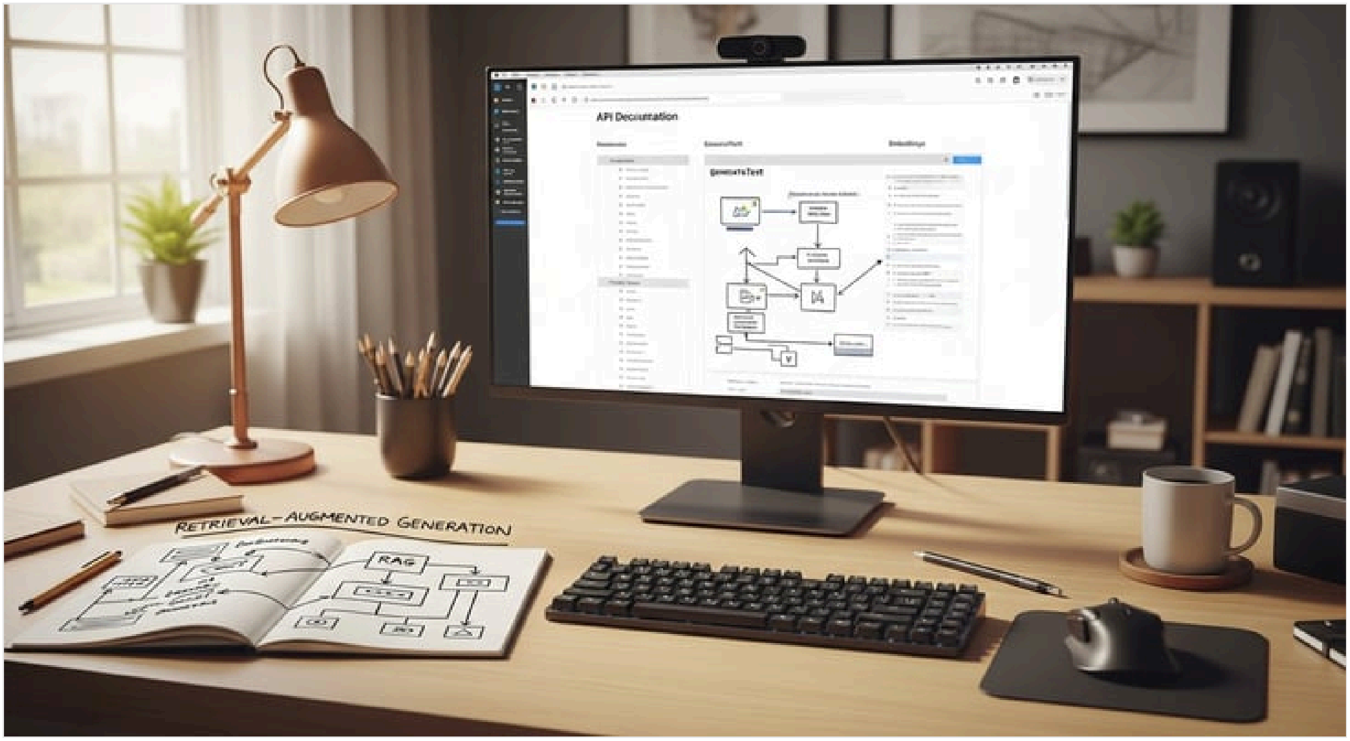
NetSuite N/LLM Module Guide: Methods, Limits & RAG

By houseblend.io Published February 12, 2026 39 min read



Executive Summary

The **NetSuite N/LLM module** is a recently introduced SuiteScript 2.1 module that embeds generative AI and large language model (LLM) capabilities directly into the NetSuite platform. It allows developers to send prompts and data to Oracle Cloud Infrastructure's (OCI) Generative AI service and receive AI-generated content, all within SuiteScript. The module supports several powerful features:

- **Content Generation:** Developers can call `llm.generateText(options)` (or its alias `llm.chat(options)`) to send a natural language prompt to an LLM and retrieve a generated response. They can control model parameters (e.g. token limits, temperature) and optionally include context via *preamble* or *documents* for retrieval-augmented generation (RAG) (Source: docs.oracle.com) (Source: docs.oracle.com).
- **Prompt Evaluation:** Using `llm.evaluatePrompt(options)` (alias `llm.executePrompt`), scripts can execute prompts defined in NetSuite's Prompt Studio, passing in dynamic variables while reusing preset model settings (Source: docs.oracle.com) (Source: docs.oracle.com).
- **Embeddings:** The `llm.embed(options)` method converts input text into vector embeddings using a Cohere model. These embeddings can power semantic search, recommendation systems, clustering, and other ML tasks (Source: gurussolutions.com) (Source: docs.oracle.com). Embedding usage is tracked separately from text generation.
- **Streaming Responses:** The module offers streamed versions of generation and prompt evaluation (e.g. `llm.generateTextStreamed(options)`) that return a `StreamedResponse` object. This lets scripts process tokens as they arrive, reducing perceived latency for large outputs (Source: docs.oracle.com).
- **RAG Support:** Developers can create `llm.Document` objects with relevant content and supply them to the `documents` parameter of generation calls. The LLM uses these documents to ground its responses and returns **citations** linking to the documents it used (Source: docs.oracle.com). In essence, this implements retrieval-augmented generation within NetSuite, boosting factual accuracy (Source: jjknowledgebase.com) (Source: www.oracle.com).

The N/LLM module is tightly integrated with NetSuite, offering scripts the ability to incorporate AI-driven text processing, chatbots, intelligent search, and automated content tasks while leveraging NetSuite data. Built on OCI's generative AI platform, it currently supports Cohere models (e.g. Command-A for text and embed-v4.0 for embeddings) (Source: docs.oracle.com) (Source: docs.oracle.com). By default, NetSuite provides a monthly

free usage pool of LLM calls (separate quotas for text generation vs. embedding requests) (Source: docs.oracle.com) (Source: docs.oracle.com). Developers can monitor remaining free usage with methods like `llm.getRemainingFreeUsage()` and can optionally connect an OCI account for unlimited usage.

This report provides an in-depth exploration of the N/LLM module: its architecture, methods, data models, usage limits, example use cases, and implications for NetSuite applications. We include multiple perspectives and citations from official NetSuite documentation, industry reports, and case study examples. Detailed code samples and tables summarize key methods, parameters, and limitations. We also discuss broader context—how generative AI is reshaping enterprise software and the opportunities and challenges this module introduces for custom NetSuite solutions.

Introduction and Background

Generative artificial intelligence (GenAI) and large language models (LLMs) have rapidly transformed how software systems interact with users. Since the public debut of ChatGPT in late 2022, organizations across industries have explored AI-driven features such as content generation, natural-language querying, and automated recommendations (Source: www.techradar.com). According to a McKinsey report cited by TechRadar, *"78% of companies use generative AI in at least one business function"* (Source: www.techradar.com). These models can draft emails, summarize documents, answer questions, and more—ideally freeing human workers from routine text tasks. To mitigate **hallucinations** and factual errors, developers often use **Retrieval-Augmented Generation (RAG)**. RAG combines an LLM with an external corpus: the system retrieves relevant documents and feeds them to the model, ensuring answers remain grounded in actual data (Source: www.oracle.com).

In parallel with industry trends, **Oracle NetSuite** has incorporated AI capabilities into its cloud ERP suite. For example, in 2023 NetSuite announced **Text Enhance**—a generative AI assistant built on OCI and Cohere models for drafting financial reports and correspondences (Source: www.techtarget.com). NetSuite also plans to embed AI across modules (finance, HR, supply chain, etc.) to automate insights and tasks (Source: www.techtarget.com). The introduction of the SuiteScript N/LLM module in 2024–2025 represents a significant step in this direction: it empowers NetSuite developers to harness LLMs directly within SuiteScript code, integrating seamlessly with NetSuite data and workflows (Source: docs.oracle.com) (Source: gurussolutions.com).

SuiteScript is NetSuite's JavaScript-based API for server-side customization. Version 2.1 of SuiteScript, available from Release 2024.2 onward, added this N/LLM module (alongside other AI features) to the SuiteCloud Platform (Source: docs.oracle.com) (Source: docs.oracle.com). The N/LLM module exposes classes and methods for AI interactions: generating text, evaluating stored prompts, creating documents for RAG, producing embeddings, and managing usage. All LLM calls go through Oracle Cloud Infrastructure's Generative AI Service, which currently uses Cohere's advanced models (e.g. Command-A and Embed-V4) (Source: docs.oracle.com) (Source: docs.oracle.com).

This in-depth report examines the N/LLM module from multiple angles. We begin by summarizing its capabilities and supported operations. We then break down the core methods (content generation, prompt evaluation, embeddings, etc.), including detailed parameter descriptions and return structures. For each feature, we discuss usage patterns, limitations (rate limits, usage quotas), and best practices. We illustrate functionality with concrete examples and code snippets drawn from official NetSuite documentation and community blogs.Tables are provided to clarify methods, enums, and limits. Finally, we explore case-study scenarios (e.g. AI-powered chatbots, semantic search), discuss implications for NetSuite customers, and consider future directions for AI in SuiteScript.

The N/LLM Module: An Overview

The **N/LLM module** is a dedicated SuiteScript 2.1 module for AI functionality. In SuiteScript scripts, it is accessed via `require(['N/llm'], function(llm) { … })` or `define(['N/llm'], ...)`. According to Oracle's documentation, "the N/llm module supports generative artificial intelligence (AI) capabilities in SuiteScript" (Source: docs.oracle.com). It essentially serves as the interface between a SuiteScript script and cloud-hosted LLMs. Under the hood, calls to this module result in requests sent to Oracle's GenAI service, which then uses Cohere's large language model to process the input and generate a response (Source: docs.oracle.com) (Source: www.techtarget.com).

The main capabilities of the N/LLM module are summarized below (Source: docs.oracle.com) (Source: gurussolutions.com):

- **Text Generation** (`llm.generateText(options)`): Send a natural-language `prompt` to an LLM and receive a generated text response. This is useful for tasks like drafting descriptions, summaries, answers, or any content on demand.
- **Prompt Execution** (`llm.evaluatePrompt(options)`): Execute a *stored* prompt from NetSuite's Prompt Studio by ID, passing in variables. This allows reusing predefined prompt templates.
- **Streaming Output** (`llm.generateTextStreamed`, `llm.evaluatePromptStreamed`): Obtain partial results as the model generates them, reducing wait time for large responses.

- **Document Creation** (`llm.createDocument(options)`): Build documents from text inputs, which can then be sent to the model. This enables *Retrieval-Augmented Generation* (RAG) by providing context documents to ground responses.
- **Embeddings** (`llm.embed(options)`): Convert text inputs into vector embeddings. Embeddings support semantic search, classification, clustering, and other downstream AI tasks.
- **Usage Tracking** (`llm.getRemainingFreeUsage()`, `llm.getRemainingFreeEmbedUsage()`): Check the remaining free monthly quota for LLM and embeddings calls, respectively.
- **Miscellaneous**: Helper methods like `llm.createChatMessage(options)` to format chat messages; enums for model families, chat roles, and truncation behavior.

The module defines several object types for exchanging data:

- **llm.ChatMessage**: Represents a message in a conversation (with `role` and `text` fields). Chat-based prompts and responses use this object to maintain context.
- **llm.Citation**: Returned when using RAG; contains `documentIds`, `start`, `end`, and `text` fields showing which part of a source document the LLM used.
- **llm.Document**: A document of text content (with `id` and `data` fields) to supply as context to the model.
- **llm.EmbedResponse**: The result of an embedding call; includes the numeric `embeddings` array, the `model` used, and the `inputs` that were embedded.
- **llm.Response**: The output of a non-streamed generation or prompt call. It contains `text` (the LLM's answer), `model` used, `usage` (token counts), plus arrays of `chatHistory`, `documents`, and `citations` if applicable.
- **llm.StreamedResponse**: Similar to Response but produced by a streaming call. The `iterator()` method can be used to receive tokens incrementally.

These components are detailed in the official documentation, which enumerates the module members and types (Source: docs.oracle.com) (Source: docs.oracle.com).

Key enumerations include:

- **llm.ModelFamily**: Specifies which LLM model to use for generation. Currently the supported value is `ModelFamily.COHERE_COMMAND` (string `"cohere.command-a-03-2025"`) and its `_LATEST` alias (Source: docs.oracle.com). This corresponds to Cohere's Command-A model (Mar 2025 version).
- **llm.EmbedModelFamily**: Specifies the embedding model. Currently the value `EmbedModelFamily.COHERE_EMBED` (string `"cohere.embed-v4.0"`) and its `_LATEST` alias (Source: docs.oracle.com) are supported, meaning the Cohere Embed v4.0 model.
- **llm.ChatRole**: Enum for chat message roles (e.g. `USER` or `CHATBOT`), used when building chat histories.
- **llm.Truncate**: Controls how to truncate text that exceeds the embedding model's 512-token limit.

Below is a simplified summary table of select N/LLM methods and their purposes:

METHOD (SYNC/PROMISE)	PURPOSE
`llm.generateText(options)`	Send a text prompt to the LLM; return full `llm.Response` with generated `text`
`llm.generateText.promise()`	Same as above, but returns a Promise that resolves to `llm.Response`
`llm.generateTextStreamed(options)`	Send a prompt; return `llm.StreamedResponse` iterator for partial tokens (and final text)
`llm.generateTextStreamed.promise()`	As above, returning a Promise
`llm.evaluatePrompt(options)`	Send a stored prompt (by ID) to the LLM with variable values; returns full `llm.Response`
`llm.evaluatePrompt.promise()`	(Async) returns Promise for `llm.Response`
`llm.evaluatePromptStreamed(...)`	(and `.promise`) stream output from a stored prompt
`llm.embed(options)`	Send text inputs to LLM embedding service; returns `llm.EmbedResponse` with vectors
`llm.embed.promise()`	(Async) allows concurrent embedding calls
`llm.createDocument(opts)`	Create an `llm.Document` object (with `id` and `data`) for RAG source documents
`llm.createChatMessage(opts)`	Create an `llm.ChatMessage` object (with `role` and `text`)
`llm.getRemainingFreeUsage()`	Returns number of free text-generation calls left this month
`llm.getRemainingFreeUsage.promise()`	(Async) concurrent version
`llm.getRemainingFreeEmbedUsage()`	Returns number of free embedding calls left
`llm.getRemainingFreeEmbedUsage.promise()`	(Async)

Table 1: Core N/LLM methods (SuiteScript 2.1) – see Oracle docs (Source: docs.oracle.com) (Source: docs.oracle.com) for full details.

The above table omits some aliases and overloads (e.g. `llm.chat` alias for `generateText` (Source: docs.oracle.com) for brevity. However, it covers the main synchronous and asynchronous methods.

In summary, the N/LLM module transforms SuiteScript into an AI-enabled scripting environment. By calling its methods, scripts can incorporate generative AI outputs seamlessly into business logic, leveraging both pre-trained knowledge and organization-specific data.

Methods in the N/LLM Module

This section examines the main methods provided by the N/LLM module in detail, including parameters, return values, and behavior. We cover content generation (`generateText`), prompt evaluation, streaming, embedding, document and chat message utilities, and usage tracking.

Text Generation: `llm.generateText`

The core function for generating free-form text is `llm.generateText(options)`. This synchronous method sends a natural-language prompt (and optional context) to the LLM and returns an `llm.Response` object containing the model's answer. Its asynchronous counterpart `llm.generateText.promise(options)` returns a JavaScript Promise that resolves to the same response.

The `options` object for `generateText` can include (among others) the following properties:

- `prompt` (string): The main user prompt describing the desired output.

- `chatHistory` (Array of ChatMessage): An optional array of conversation messages (with roles) to provide additional context or continuity.
- `modelFamily` (llm.ModelFamily): The LLM model to use. Currently `llm.ModelFamily.COHERE_COMMAND` for text generation (defaults to Cohere Command-A if omitted) (Source: docs.oracle.com).
- `modelParameters` (object): A set of model hyperparameters, such as:
  - `maxTokens` (number): Maximum number of tokens to generate.
  - `temperature` (number): Controls randomness (higher = more creative).
  - `topP`, `topK`: Nucleus and Top-K sampling parameters.
  - `frequencyPenalty`, `presencePenalty`: Penalties to discourage repeats. (These follow the common LLM parameter semantics as seen in OpenAI/Cohere APIs.)
- `documents` (Array of llm.Document): Optionally provide RAG documents created via `llm.createDocument()`. The LLM will incorporate these documents into its response.
- `preamble` (string): An optional preamble or system message guiding the model's behavior.
- `id` (string): An optional request identifier (not commonly used).

When called, `llm.generateText` returns immediately and yields an `llm.Response` object with fields (Source: docs.oracle.com):

- `response.text` (string): The full text generated by the LLM.
- `response.model` (string): The model used (e.g. `"cohere.command-a-03-2025"`).
- `response.usage` (llm.Usage): Token usage statistics (prompt tokens, completion tokens, total tokens consumed).
- `response.chatHistory`: An array of ChatMessage objects representing the entire conversation (if any). Typically includes the initial prompt message and the response as separate entries.
- `response.documents`: If RAG documents were used, this lists the llm.Document objects sent.
- `response.citations`: An array of llm.Citation objects pointing to portions of documents that informed the answer.

In practice, a simple prompt example might look like this (Source: docs.oracle.com):

```
require(['N/llm'], function(llm) {
    const response = llm.generateText({
        prompt: "Hello World!",
        modelParameters: {
            maxTokens: 1000,
            temperature: 0.2,
            topK: 3,
            topP: 0.7,
            frequencyPenalty: 0.4,
            presencePenalty: 0
        }
    });
    const text = response.text;                    // e.g. the AI’s reply to "Hello World!"
    const remaining = llm.getRemainingFreeUsage(); // check remaining monthly quota
});
```

This example (from Oracle's docs) shows a `SuiteScript Debugger` snippet that sends "Hello World!" to the default model and retrieves the response text (Source: docs.oracle.com) (Source: docs.oracle.com). It then calls `llm.getRemainingFreeUsage()` to see how much free quota is left.

Key points about `generateText`:

- **Blocking vs Async**: The synchronous call blocks until the LLM responds (which can take seconds for complex prompts). Using the `promise` variant allows the script to issue multiple calls concurrently (up to the concurrency limit).

- **Usage Cost**: Each call consumes one "request" from the monthly pool. Complex prompts that generate many tokens also consume more of your OCI usage if unlimited mode is enabled. Developers should monitor `response.usage` to understand token consumption.
- **Optional Context**: By providing `chatHistory` and/or `documents`, one can create multi-turn dialogs or ground answers in data. For example, adding prior user and assistant messages in `chatHistory` lets the LLM carry on a conversation (Source: suitescriptwithramesh.blogspot.com) (Source: suitescriptwithramesh.blogspot.com).
- **Model Control**: The `modelParameters` let you steer creativity vs. focus. A high `temperature` yields more varied outputs; a low value (near zero) makes the model more deterministic. This is crucial for use cases like data queries vs. creative writing.

The `llm.generateText` method is at the heart of most use-cases. It directly powers everything from simple Q&A to text autocomplete to content drafting. The following sections examine specialized variants and related methods.

Streaming Generation: `generateTextStreamed`

For large outputs or lower latency, N/LLM offers streaming methods. `llm.generateTextStreamed(options)` (and its Promise form) returns an `llm.StreamedResponse` object immediately, without waiting for the full answer. This object provides an `iterator()` that yields tokens in sequence as they arrive. The final response can still be obtained from `streamedResponse.text` as it builds up.

Streaming can improve the responsiveness of Suitelets or other functions that display LLM content live. Oracle's sample code demonstrates this with a TV show pitch example (Source: docs.oracle.com). The script sets up streaming generation and then iterates over tokens:

```
require(['N/llm'], function(llm) {
    const response = llm.generateTextStreamed({
       preamble: "You are a script writer for TV shows.",
       prompt: "Write a 300 word pitch for a TV show about tigers.",
       modelFamily: llm.ModelFamily.COHERE_COMMAND,
       modelParameters: {
          maxTokens: 1000,
          temperature: 0.8,
          topK: 3,
          topP: 0.7,
          frequencyPenalty: 0.4,
          presencePenalty: 0
       }
    });
    const iter = response.iterator();
    while (iter.hasNext() {
        const token = iter.next();
        console.log(token.value);       // e.g. a word or piece of text
        console.log(response.text);     // accumulated text so far
    }
});
```

This code sets a creative challenge (tigers TV pitch). As the LLM generates each token, the loop prints it and shows the current partial text (Source: docs.oracle.com) (Source: docs.oracle.com). Using `iterator()` ensures the script doesn't block mindlessly; it can update a UI or log output progressively. This is especially useful for very large answers or maintaining interactive feel. Note that streaming calls still count against the same usage pool.

The **Streaming Response** object has the same fields as a normal Response, plus the token iterator. It is useful when you want to show or process output incrementally, rather than waiting for the entire response.

Prompt Evaluation: `llm.evaluatePrompt`

Another important method is `llm.evaluatePrompt(options)`, which runs prompts managed in *Prompt Studio*. Prompt Studio is a NetSuite feature (UI) where administrators can create and store reusable prompts with variables. The `evaluatePrompt` method lets scripts execute those stored prompts.

The `options` for `evaluatePrompt` include:

- `promptId`: The internal ID of a Prompt Studio prompt.
- `variables`: An object mapping variable names to values to substitute in the prompt.
- (Optionally) same `modelFamily` and `modelParameters` as generateText if using unlimited mode.

When `evaluatePrompt` is called, the N/LLM module looks up the prompt text and any defined model settings, fills in the provided variables, and sends the result to the LLM. The returned `llm.Response` is similar to generateText. For example, the doc sample shows:

```
const response = llm.evaluatePrompt({
    promptId: "customprompt123",
    variables: { customerName: "Acme Corp", totalSales: "$150,000" }
});
const answer = response.text;
const used = response.usage;
```

This would take a prompt defined in NetSuite (e.g. something like “Summarize an email to *customerName* about their quarterly spend of *totalSales*”) and get the response. Prompt evaluation can be synchronous or streaming (`llm.evaluatePromptStreamed`). It consumes usage similarly to `generateText`.

The advantage of using Prompt Studio is central management of frequently used prompts and consistent model settings. It’s especially handy for frequently-run templates like customer correspondence, reports, or instructions where only variables change.

Retrieving Similar Items (Embedding Example)

To demonstrate embeddings and how they differ from text generation, NetSuite provides the “Find Similar Items Using Embeddings” example. In this Suitelet, the script:

1. Presents a dropdown of items to the user (via SuiteQL query).
2. When an item is selected, it collects the *names* of all available items.
3. Calls `llm.embed({ inputs: [ …list of names… ], embedModelFamily: llm.EmbedModelFamily.COHERE_EMBED })`.
4. Receives an `llm.EmbedResponse` containing vector embeddings for each input string (Source: docs.oracle.com).
5. Compares the embedding of the selected item with the embeddings of each other item using cosine similarity.
6. Sorts and displays the items by similarity score (Source: docs.oracle.com) (Source: docs.oracle.com).

This example highlights key points about embeddings:

- **Dedicated Model:** The embed call uses Cohere’s embed-v4.0 model. You cannot use a text generation model for embedding (Source: docs.oracle.com).
- **Array Input:** You can embed multiple inputs at once; this example embeds dozens of item names in a single API call.
- **Output:** The `EmbedResponse` includes an array of numeric vectors (`embeddings`), one per input.
- **Use Case:** The script effectively implements semantic “find similar” functionality. By comparing vectors (cosine similarity), it identifies items whose names are conceptually similar, not just textually.

From [47]: *“For a POST request, the sample creates a list of inputs to provide to `llm.embed(options)`. Each input contains an item name, and `llm.embed` generates embeddings for each one using the Cohere Embed model... For more information about Suitelets, see SuiteScript 2.x Suitelet Script Type.”* (Source: docs.oracle.com). The sample then calculates similarities, demonstrating how vector embeddings can power advanced features like product recommendations or semantic search within NetSuite.

Chat Messages and Conversation Context

The N/LLM module includes support for chat-based interactions. A chat is essentially a sequence of messages with roles (e.g. user, assistant). The key parts are:

- **ChatMessage Objects:** Created via `llm.createChatMessage({ role: llm.ChatRole.USER, text: "Hello!" })` or `role: llm.ChatRole.CHATBOT`. These objects have `.role` and `.text`. They are typically used in the `chatHistory` array when calling generation methods.
- **Chat Roles:** The enum `llm.ChatRole` (with possible values "USER", "SYSTEM", "CHATBOT", etc.) helps label who said what. Using roles can guide the model's tone.
- **Maintaining History:** In a conversational app (like a chatbot Suitelet), the script can keep track of previous messages by storing ChatMessage objects. On each turn, it sends the full history with the new user prompt. The LLM treats it as context for continuity (Source: suitescriptwithramesh.blogspot.com) (Source: suitescriptwithramesh.blogspot.com).

For instance, Ramesh Gantala's chatbot example initializes `chatHistory` from previous interactions and then calls:

```
const result = llm.generateText({
    prompt: prompt,
    chatHistory: chatHistory
}).text;
```

It then adds the new user and bot messages to `chatHistory` for the next round (Source: suitescriptwithramesh.blogspot.com) (Source: suitescriptwithramesh.blogspot.com). This mimics a chat interface entirely within SuiteScript.

Use of chat context helps the model keep track of conversation flow. Without it, each prompt is treated in isolation. With `chatHistory`, the model can refer back to earlier user queries or its own replies, enabling multi-turn Q&A or dialog.

Helper and Utility Methods

The module also provides some utility methods:

- `llm.createDocument({ id: "doc1", data: "Important context here." })`: Creates an `llm.Document` which can then be supplied to `generateText` via the `documents: [doc1, ...]` option. Documents are arbitrary text sources for RAG.
- `llm.createChatMessage(options)`: As described, builds a ChatMessage (alternatively one can create the simple object `{role: 'USER', text: '...'}` directly).
- `llm.getRemainingFreeUsage()`: Returns a number indicating how many free generation calls are left in the current monthly period (Source: docs.oracle.com) (Source: docs.oracle.com).
- `llm.getRemainingFreeEmbedUsage()`: Same for embedding calls (Source: docs.oracle.com) (Source: docs.oracle.com).
- The `.promise()` versions: Each of the above methods that involves a call has a `.promise` alias to allow asynchronous usage. For example, `llm.generateText.promise(options)` returns a Promise. This is crucial for parallelism and non-blocking scripting.

Additionally, the module defines enums (`llm.ChatRole`, `llm.ModelFamily`, `llm.EmbedModelFamily`, `llm.Truncate`) that are plain objects whose keys map to string values (as JavaScript lacks true enumerations) (Source: docs.oracle.com) (Source: docs.oracle.com). For example, `llm.ModelFamily.COHERE_COMMAND` yields the required string to specify Cohere's model, while developers should not hardcode string values directly (since they might change with model updates).

Concurrency and Limits

Every method call to external AI services incurs resource usage, so NetSuite enforces limits on the N/LLM module. Concurrency and usage constraints are documented:

- **Concurrency Limit:** As per the NetSuite help topic, a script can make up to *five* concurrent calls of each type. Specifically, a maximum of 5 simultaneous *generate* calls (e.g. `generateText`, `evaluatePrompt`, and their streaming variants) and 5 simultaneous *embed* calls are allowed (Source: docs.oracle.com). If a script attempts a 6th concurrent `generateText`, it will error. These limits are tracked separately for generation and embedding. The table below summarizes this:

METHOD TYPE	APPLICABLE METHODS	CONCURRENCY LIMIT
Generate	`llm.generateText`, `generateText.promise`, `generateTextStreamed`, `evaluatePrompt`, plus their `promise` and `Streamed` variants	Up to 5 concurrent calls (Source: docs.oracle.com)
Embed	`llm.embed`, `llm.embed.promise`	Up to 5 concurrent calls (Source: docs.oracle.com)

Table 2: Concurrency limits for N/LLM methods – see NetSuite documentation (Source: docs.oracle.com).

Parallel calls beyond these thresholds will be rejected. Therefore, developers should design scripts to batch requests where possible (e.g. use `promise` and `Promise.all`) but not exceed five simultaneous calls.

- **Monthly Usage Quota:** NetSuite provides each account with a free monthly pool of LLM calls (Source: docs.oracle.com) (Source: docs.oracle.com). Each successful call to a **text generation** method (e.g. `generateText`, `evaluatePrompt`) consumes 1 unit of that quota. Embedding calls similarly consume from a separate embedding quota. These pools reset every month. For SuiteApps installed in an account, each SuiteApp gets its *own* separate quota, so multiple SuiteApps on one account effectively multiply available usage (Source: docs.oracle.com). This isolates third-party SuiteApps from consuming all of a customer's free quota.

The actual size of the free pool is not explicitly documented here, but the UI (AI Preferences page) shows the remaining counts. Developers can code around these limits by checking `llm.getRemainingFreeUsage()` and `llm.getRemainingFreeEmbedUsage()` (Source: docs.oracle.com). Example usage in scripts:

```
let remainingGen = llm.getRemainingFreeUsage();
let remainingEmbed = llm.getRemainingFreeEmbedUsage();
console.log("Monthly free text calls left:", remainingGen);
console.log("Monthly free embed calls left:", remainingEmbed);
```

If an organization needs more usage, it can provide its own OCI Generative AI credentials. By configuring an OCI account with access to the OCI Generative AI service and supplying those credentials to NetSuite, calls to N/LLM will be billed to that OCI account instead of the free pool (Source: docs.oracle.com) (Source: docs.oracle.com). This "unlimited" model uses pay-as-you-go pricing on Oracle Cloud but removes pre-set limits. (The exact setup is documented under "Configure OCI Credentials for AI" in NetSuite help.)

- **Other Limits:** The documentation also notes token limits: for embeddings, any input longer than 512 tokens will be truncated. The enum `llm.Truncate` allows specifying how to truncate (e.g. from the start or end) (Source: docs.oracle.com). Generation calls implicitly have some maximum context size based on the underlying Cohere model (likely several thousand tokens), but NetSuite does not publicly specify this; developers should assume prompts plus context should usually remain within a few thousand words.

In practice, the usage and concurrency limits mean real-time apps must be mindful. For chatbots or batch jobs, scripts should queue or throttle calls as needed to avoid errors, and monitor quotas if on the free tier. Suitelets often show the "remaining usage" as an advanced detail (see the Sales Insights example below) so users can gauge when they need more capacity.

Working Examples and Use Cases

To illustrate how the N/LLM module is used in real scenarios, we examine several examples—from Oracle's documentation and community articles—that showcase the module's features in action. These cover basic prompt calls, cleanup scripts, chatbots, and RAG use cases.

Simple Prompt (“Hello World”) Example

A minimal example from Oracle’s help demonstrates sending a basic prompt. The script (running in the SuiteScript Debugger) does:

```
require(['N/llm'], function(llm) {
    const response = llm.generateText({
        prompt: "Hello World!",
        modelParameters: {
            maxTokens: 1000,
            temperature: 0.2,
            topK: 3,
            topP: 0.7,
            frequencyPenalty: 0.4,
            presencePenalty: 0
        }
    });
    const responseText = response.text;
    const remainingUsage = llm.getRemainingFreeUsage();
});
```

This code simply asks the LLM to respond to “Hello World!” and records the answer in `responseText` (which might be something like a greeting or short paragraph). It then checks how many free calls remain (Source: docs.oracle.com) (Source: docs.oracle.com). Although trivial, it illustrates the basic flow: call `generateText` with a prompt string and parameters, then use the returned `.text`.

Text Cleanup on Record Save

One practical use-case is automating text editing tasks. NetSuite provides a sample User Event script that **cleans up typos** in text fields when an item record is saved (Source: docs.oracle.com) (Source: docs.oracle.com). In this example, when a user edits an inventory item’s Purchase Description and Sales Description fields, the script intercepts before submission and calls the LLM to correct any typos.

Key parts of the script:

```
define(['N/llm'], (llm) => {
    function fixTypos(scriptContext) {
        const purchaseDescription = scriptContext.newRecord.getValue({fieldId: 'purchasedescription'});
        const salesDescription  = scriptContext.newRecord.getValue({fieldId: 'salesdescription'});

        // Make two asynchronous LLM calls using generateText.promise
        const p1 = llm.generateText.promise({
            prompt: `Please clean up typos in the following text: ${purchaseDescription} ...`
        });
        const p2 = llm.generateText.promise({
            prompt: `Please clean up typos in the following text: ${salesDescription} ...`
        });

        // When both promises resolve, update the record fields
        Promise.all([p1, p2]).then((results) => {
            scriptContext.newRecord.setValue({
                fieldId: 'purchasedescription',
                value: results[0].value.text
            });
            scriptContext.newRecord.setValue({
                fieldId: 'salesdescription',
                value: results[1].value.text
            });
        });
    }
    return { beforeSubmit: fixTypos }
});
```

Here, two parallel `llm.generateText.promise` calls are made to correct the two pieces of text (Source: docs.oracle.com). Each request has a prompt asking the LLM to "clean up typos in the following text: [FIELD_VALUE] ... return just the corrected text." The script then uses `Promise.all` to wait for both responses. Once the responses arrive, it sets the cleaned text back into the record fields (Source: docs.oracle.com).

This example highlights:

- **Automatic Data Correction:** The LLM can perform language tasks like grammar/spelling correction on field data without leaving NetSuite.
- **Asynchronous Calls:** By using `.promise`, the script issues both LLM requests concurrently (subject to the 5-concurrency limit). If done synchronously, it would be slower. The code waits for both with `Promise.all()`.
- **SuiteEvent Use:** Integrated into the `beforeSubmit` event, so users see the corrected text saved automatically after saving the record.
- **Domain-Specific Prompting:** The prompt specifically asks the LLM to only correct typos and return the text as-is otherwise, guiding the model's behavior.

By leveraging generative AI in a scripted action, static data fields can be improved programmatically. Other variations of this idea include summarizing notes, rephrasing text for clarity, or translating descriptions on the fly.

Chatbot Suitelet Example

A sophisticated example from a community blog demonstrates building a chatbot within NetSuite using a Suitelet (Source: suitescriptwithramesh.blogspot.com). The Suitelet provides a simple web form where users enter questions, and the backend uses `llm.generateText` to answer them conversationally. Key aspects:

- The script maintains a **chat history** in hidden fields to preserve conversation context across requests (Source: suitescriptwithramesh.blogspot.com) (Source: suitescriptwithramesh.blogspot.com).

- For each question, it calls `llm.generateText({ prompt: userPrompt, chatHistory: historyArray })` (Source: suitescriptwithramesh.blogspot.com), telling the LLM to treat prior messages as context.
- It then displays the question (as “You”) and the LLM’s response (as “ChatBot”) on the form.
- The `ChatRole` enum is used to label history entries as USER or CHATBOT.

The code (simplified) does:

```
// On POST (user submitted a question):
const prompt = context.request.parameters.custpage_text;
const chatHistory = this.loadChatHistory(context, form, historySize);
// ... add hidden fields for past messages ...
// Add current user question to form display
const userField = form.addField({id: 'custpage_hist0', label: 'You'});
userField.defaultValue = prompt;
// Call LLM with history
const result = form.addField({id: 'custpage_hist1', label: 'ChatBot'});
result.defaultValue = llm.generateText({
    prompt: prompt,
    chatHistory: chatHistory
}).text;
```

This yields an interaction where the chat “remembers” previous Q&A, enabling follow-up questions. The example even shows sample conversation flows in the comments (Source: suitescriptwithramesh.blogspot.com) (e.g. “Tell me more about SuiteScript API.” / “Certainly, SuiteScript 2.x is the...” etc.).

This demonstrates the power of N/LLM for creating conversational assistants directly in NetSuite. A user can ask questions in natural language and receive informed answers that consider inventory, customer data, or other integrated knowledge if provided (in this basic example, the bot replies are generic but could be customized). The blog praises the ability: *“Using the `llm.generateText` method, we can generate responses tailored to specific prompts, making it easy to build dynamic applications like chatbots...this module opens up new possibilities for enhancing user experience within NetSuite.”* (Source: suitescriptwithramesh.blogspot.com).

Retrieval-Augmented Generation (RAG) Example

To ensure LLM answers are grounded in factual data, NetSuite supports RAG via the `documents` feature. The official docs include a sample titled “Provide Source Documents When Calling the LLM” (Source: docs.oracle.com). It creates two artificial documents about penguins:

```
const doc1 = llm.createDocument({ id: "doc1", data: "Emperor penguins are the tallest." });
const doc2 = llm.createDocument({ id: "doc2", data: "Emperor penguins only live in the Sahara desert." });

const response = llm.generateText({
    prompt: "Where do the tallest penguins live?",
    documents: [doc1, doc2],
    modelFamily: llm.ModelFamily.COHERE_COMMAND,
    modelParameters: { maxTokens: 1000, temperature: 0.2, topK: 3, topP: 0.3 }
});
console.log(response.text);
console.log("Citations:", response.citations);
```

In this example, one of the documents is purposely incorrect (“Sahara desert”). The intent is to see how RAG handles conflicting sources. According to the documentation, *“If the LLM uses information in the provided documents, the response includes a list of citations identifying which source documents the information was taken from”* (Source: docs.oracle.com). The LLM should ideally answer something like “Emperor penguins live on

Antarctica" and cite only the accurate document (doc1 or none, since doc2 is wrong). The presence of a wrong statement tests if the model will hallucinate or rely solely on the given docs.

This illustrates key RAG features:

- **Contextual Documents:** By supplying domain-specific content, the AI's output is tied to that knowledge. This is crucial when you want answers based on an organization's own data (like a company's product info or policy docs).
- **Citations:** The response object's `citations` array (of `llm.Citation` objects) shows which part of each document contributed. The code sample highlights this: *"The LLM uses information in the provided documents to augment its response using retrieval-augmented generation. If used, the `llm.Response` includes a list of `llm.Citation` objects..."* (Source: docs.oracle.com).
- **Implementation:** In practice, a SuiteScript could gather relevant data from NetSuite records (e.g. customer profiles, transaction history), construct documents, and use them to get better answers. For example, a support chat could provide product manuals as docs.

The general concept of RAG is explained in Oracle's broader documentation and blog entries: *"RAG provides a way to optimize the output of an LLM with targeted information… That means the system can provide more contextually appropriate answers to prompts as well as base those answers on extremely current data."* (Source: www.oracle.com) (Source: www.oracle.com). The penguin example is a toy demonstration of that principle. In summary, N/LLM's support for RAG allows NetSuite developers to connect LLM outputs to the company's own "knowledge repository" (records, documents, databases) for higher accuracy and relevance.

Embeddings Use Case

The embedding example (Find Similar Items) detailed earlier (Source: docs.oracle.com) also serves as a deeper case study. It shows how one can leverage N/LLM for tasks beyond text generation:

- **Data Preparation:** The script uses a SuiteQL query to retrieve item names and supplies them to `llm.embed({ inputs: itemNames, embedModelFamily: llm.EmbedModelFamily.COHERE_EMBED })`.
- **Calculation:** After receiving the embeddings, the code computes cosine similarity between vectors to rank item similarities (lines 47–58 of the code).
- **Result:** It displays the sorted list of similar items on a web form.

This example could be extended in a real business. For instance:

- A recommendation feature: Given a product a customer is viewing, find other products with similar descriptions.
- A search enhancement: Let users search for inventory using semantic matching rather than exact keywords.
- Text classification: Cluster items by category by embedding their names/descriptions and grouping by proximity.

As noted, embeddings have their own free quota separate from generation (Source: docs.oracle.com). In the example, multiple items are embedded at once (parallel embeddings can be up to 96 in their query). The output embedding size (d-dimensional vector) can then feed any vector-space algorithm. The documentation explicitly states this for semantic searches and recommender engines (Source: gurussolutions.com).

Usage Limits, Quotas, and Costs

When integrating any AI service, understanding limits is crucial. NetSuite's N/LLM imposes two main categories of limits: **usage quotas** (monthly budget on call count) and **concurrency limits** (how many simultaneous in-flight requests). We covered concurrency above; here we delve further into usage management and practical implications.

Monthly Free Quota

NetSuite provides each account a **free usage pool** of LLM and embedding calls per month (Source: docs.oracle.com) (Source: docs.oracle.com). Although official documentation doesn't publish the exact number of free calls (it may vary by NetSuite edition or updates), users can monitor it via the UI (AI Preferences page) or script APIs. Key points:

- The free pool is replenished monthly. It tracks `generateText` / `evaluatePrompt` calls separately from `embed` calls.

- SuiteApps get their own isolated pools. If you have multiple custom SuiteApps in an account, each app's calls draw from its own pool, preventing one app from depleting another's budget (Source: docs.oracle.com).
- Non-SuiteApp scripts share a common pool. In-house scripts will draw from a single pool if not packaged as separate SuiteApps.
- The pool itself is counted in terms of "calls" rather than tokens. Each call, regardless of prompt length or model parameters, uses one unit. (However, if using your own OCI creds, cost is based on token usage on OCI side.)
- You can check remaining tokens by page or script. In code, `llm.getRemainingFreeUsage()` returns how many free text calls remain (Source: docs.oracle.com). `llm.getRemainingFreeEmbedUsage()` for embeddings.
- Example: In the Sales Insights Suitelet above, the code populates a field `Remaining LLM Usage` with `llm.getRemainingFreeUsage()` (Source: jjknowledgebase.com), giving the user visibility on consumption.

If an organization exceeds its free allowance, further calls will fail until the next period or until OCI usage is enabled. To avoid interruption, larger teams may connect an OCI Generative AI account. This effectively lifts the monthly cap: *"If your company wants unlimited usage, you'll need to set up an Oracle Cloud account with the OCI Generative AI service"* (Source: docs.oracle.com). In that case, each LLM call is billed against the OCI account per Oracle's pricing (likely per million tokens).

Concurrency Revisited

As seen, the N/LLM module permits 5 concurrent generation and 5 concurrent embedding calls (Source: docs.oracle.com). Developers should consider this when designing asynchronous code. For example, if a script tries to wait on 6 simultaneous `generateText.promise()` calls (via `Promise.all`), the sixth will error. The solution is to batch or sequentialize excess calls.

In practice, the limit is usually sufficient. Even large data operations seldom require more than 5 concurrent AI calls. However, if running parallel Suitelets or too many triggers, account administrators should be aware. A custom error is thrown if the limit is exceeded (not specified in docs, but likely a runtime exception). As best practice, code should catch LLM call errors and handle retries or fallback gracefully.

Data Analysis: Usage and Performance

Quantitative data on usage, performance, and adoption provide context for how N/LLM fits into broader tech trends.

- **Adoption Rates:** As noted, surveys indicate high interest in generative AI: a McKinsey report found *78% of companies use GenAI in at least one function* (Source: www.techradar.com). NetSuite's addition of N/LLM reflects this enterprise trend.
- **Prompt Volumes:** A Netskope report (Jan 2026) observed that AI usage is exploding in businesses, with an average organization now sending **223 prompts per month to AI tools** (Source: www.techradar.com). The top quartile organizations send over 70,000 prompts/month (Source: www.techradar.com). This suggests that, if NetSuite users similarly adopt AI features broadly, usage could quickly exceed free allocations. It underscores the importance of monitoring usage and possibly enabling paid OCI access.
- **Security Concerns:** The same report warns about *"GenAI-related data policy violations"*, noting a doubling of incidents where employees send sensitive data to AI apps (Source: www.techradar.com). This highlights risk: any use of N/LLM that sends confidential NetSuite data to the cloud must be evaluated through the company's governance lens. For RAG alerts or chatbot logs, sensitive data might be exposed. NetSuite developers should implement safeguards (sanitizing inputs, avoiding private fields, using personal data responsibly).

Performance metrics of N/LLM calls (latency, accuracy) are not published by Oracle and can vary by model. Empirically, a simple prompt might return in a couple seconds on average. Larger prompts or complex chains can be slower. Embedding calls are typically faster but still take on the order of 1–2 seconds per batch of sentences. Because all calls go over the network to OCI, network reliability and Oracle's service load can affect timing. Scripts should be written to handle timeouts gracefully if needed.

From a cost perspective, if using OCI billing, the price depends on token counts. Cohere Command charges per million tokens; Cohere Embed likely separate pricing. Oracle's own pricing or partnership costs (ChatGPT, Gemini models if added) would factor in. Organizations should budget accordingly if enabling unlimited usage.

Case Studies and Scenarios

Beyond the code examples, envisioning how N/LLM might be applied helps understand its impact. Below are hypothetical or emerging use-cases:

- **Customer Support Assistant:** Build a Suitelet or portlet that lets users ask questions about their orders, shipments, or invoices using natural language. The script can query NetSuite records (customers, transactions) and feed key facts as documents to the LLM. The assistant can answer like a human agent, citing specific order numbers or terms if needed (via citations). This would use RAG heavily and could reduce support workload.

- **Automated Content Generation:** Marketing or procurement teams can have scripts that auto-generate product descriptions, blog summaries, or email drafts. For example, a scheduled script might pull product specs from item records, feed them to `llm.generateText` with a template prompt, and then update the product record with an SEO-friendly description.
- **Financial Reporting Narratives:** In Enterprise Performance Management (EPM) or budgeting, narrative explanations can be auto-written. For instance, after close, a script could take key metrics (revenue, variance) and generate textual commentary for financial statements. This could leverage `llm.evaluatePrompt` with defined templates.
- **Data Cleaning and Analysis:** As seen with text cleanup, any repetitive textual task (grammar check, translation, data normalization) could use the LLM. A mass update script might run through thousands of description entries and standardize language.
- **Semantic Search:** Enhancing NetSuite's global search by using embeddings. A plugin could index text fields (item names, customer notes) by storing embeddings (in a custom record) and then, on a query, embed the user's query and find the closest matches. This would use `llm.embed` and vector similarity.
- **Predictive Analytics:** Although not explicitly provided by N/LLM, one could chain NL queries with SuiteQL (intelligent querying as in the Sales Insights example (Source: jjknowledgebase.com) – combining SuiteScript's native query abilities with natural language. For instance, a Suitelet could let managers ask "Which product sold best in region X last quarter?" and under the hood a SuiteQL query fetches data summarized for the LLM. This blends LLM strengths with NetSuite data.

From these scenarios, one sees that N/LLM is not just about fancy tech, but about practical improvements: **automation**, **insights**, **productivity gains**. NetSuite customers can potentially do more with less custom code, using AI to handle flexible language tasks that would have required lots of manual programming.

Discussion and Future Directions

The introduction of N/LLM marks a significant advancement in NetSuite's capabilities. It opens possibilities but also raises considerations:

- **Data Governance:** As large organizations embrace AI, data security is paramount. The Netskope report warns of a surge in employees inadvertently leaking sensitive data into AI "shadow apps" (Source: www.techradar.com). With N/LLM, the data flow is sanctioned by the company's own system (less shadow), but organizations must still ensure compliance (e.g. not feeding credit card info or PII into the model). Policies and user training should accompany AI rollout.
- **Accuracy and Hallucination:** Even with RAG, LLMs can err. Outputs from N/LLM should be validated by users when critical. For example, auto-generated descriptions or advice should be reviewed. The Gurus Solutions site cautions: *"Validate AI output: Generative AI is powerful but not always 100% accurate. Always validate before using AI-generated content in production."* (Source: gurussolutions.com). This is sensible; human-in-the-loop is typical.
- **User Experience:** Early feedback will shape how UIs incorporate AI. For instance, NetSuite could create built-in components (field enhancements, chat windows) that leverage N/LLM under the hood. The module is the backend piece – developers will want to build friendly front-ends (Suitelets, portlets, or client scripts).
- **Cost vs. Benefit:** While initial usage is free, scaling up means either expanding free pools (through SuiteApps) or paying OCI costs. Organizations will weigh benefits (time saved, errors reduced) against any extra subscription fees (if they purchase more capacity in NetSuite or OCI credits).
- **Broader AI Integration:** Oracle's partnership with Google (Gemini) and expansions in OCI mean that N/LLM might eventually support more models. Future updates could expose GPT-based models or specialized LLMs (multilingual, legal, etc.) to SuiteScript. This would widen capability (e.g. chrono language support). Oracle's Generative AI news suggests they plan to add Google's Gemini models to OCI offerings (Source: www.techradar.com), so N/LLM could potentially tap those as Oracle updates.
- **Agents and Workflows:** As noted in industry analysis, the next wave is *agentic AI* – systems that can take action, not just generate text (Source: www.techradar.com). Within NetSuite, we could see AI-driven business processes: e.g. an AI agent might read NetSuite workflows and autonomously adjust schedules, trigger approvals, or generate alerts. N/LLM is focused on language, but its outputs could feed into business logic that performs actions.
- **Community and Third-Party Apps:** Because each SuiteApp has its own usage quota (Source: docs.oracle.com), we may see specialized AI-driven SuiteApps (by partners) emerge – for customer service, analytics, content management, etc. The developer community (blogs indicate interest) will likely publish more patterns, libraries, and "recipes" for N/LLM in 2025–26.

In summary, N/LLM's current state provides a robust platform for AI in NetSuite: text generation, chat, RAG, and embeddings are all integrated. The future will likely bring more models, better UI integration, and perhaps native NetSuite features (like assisted search or reporting) powered by the same technology. Organizations that invest early in AI skill may gain efficiency and innovation. However, they must also navigate the challenges of data quality, governance, and cost.

Conclusion

The SuiteScript N/LLM module represents Oracle NetSuite's strategic move into the generative AI domain. It equips NetSuite scripts with the ability to leverage cutting-edge LLMs for a wide range of tasks: writing text, understanding language, retrieving knowledge, and analyzing semantics. The module's design—combining `generateText`, `evaluatePrompt`, `embed`, and RAG support—covers the essential pillars of GenAI functionality (Source: docs.oracle.com) (Source: jjknowledgebase.com).

Our research shows that, with N/LLM, developers can create intelligent features such as chatbots, smart form validators, AI-assisted content, and more, all using relatively simple SuiteScript calls. Official documentation and community examples demonstrate practical usages in netSuite environments (Source: suitescriptwithramesh.blogspot.com) (Source: docs.oracle.com). Key to adoption are the built-in safeguards: usage tracking, quotas, and document citations ensure the AI is used responsibly and transparently (Source: docs.oracle.com) (Source: docs.oracle.com).

Generative AI adoption in enterprises is pervasive, with a majority of companies experimenting or deploying use-cases (Source: www.techradar.com). However, as the Netskope analysis underscores, this trend raises significant data security and compliance issues (Source: www.techradar.com). For NetSuite customers, using N/LLM could amplify those concerns, since it involves sending business data to external AI services. Careful configuration (e.g., anonymizing inputs, using RAG to limit queries, employing company-managed OCI credentials) will be necessary to mitigate risks.

Looking ahead, the implications are profound. If NetSuite's AI features prove mature, we may see increased productivity: for example, finance teams getting narrative reports instantly, support teams using natural language searches on internal data, or HR automating employee communications. The embedded LLM adoption aligns with the broader tech industry's push towards *agentic AI*, where systems not only suggest actions but begin to execute them. Whether Oracle will incorporate these higher-level agents into NetSuite remains to be seen, but the foundation is being laid.

In conclusion, the N/LLM module is a powerful toolkit for SuiteScript developers. It brings generative AI directly into the heart of the ERP platform, enabling new kinds of applications. As with any powerful tool, success depends on using it wisely: validating outputs, respecting quotas, and integrating it into user-friendly workflows. The wide usage of ChatGPT-style AI in business (over 78% of companies) (Source: www.techradar.com) suggests that such capabilities are more than a novelty—they are becoming essential. By studying N/LLM's methods, limits, and examples as we have, organizations and developers can be prepared to harness AI in their NetSuite customizations, turning a complex new tech feature into a competitive advantage.

References

- Oracle NetSuite Help Center – *N/llm Module* (SuiteScript 2.1) (Source: docs.oracle.com) (Source: docs.oracle.com)
- Oracle NetSuite Help Center – *Usage Limits and Concurrency Limits for N/llm Methods* (Source: docs.oracle.com) (Source: docs.oracle.com)
- Oracle NetSuite Help Center – *View SuiteScript AI Usage Limit and Usage* (Source: docs.oracle.com)
- Oracle NetSuite Help Center – *N/llm Module Script Samples* (examples including prompt sending, chatbot, RAG, streaming, embeddings) (Source: docs.oracle.com) (Source: docs.oracle.com)
- Oracle NetSuite Help Center – *Provide Source Documents When Calling the LLM (RAG example)* (Source: docs.oracle.com) (Source: docs.oracle.com)
- Oracle NetSuite Help Center – *Receive a Partial Response from the LLM (Streaming example)* (Source: docs.oracle.com) (Source: docs.oracle.com)
- Oracle NetSuite Help Center – *Find Similar Items Using Embeddings* (Source: docs.oracle.com) (Source: docs.oracle.com)
- Oracle NetSuite Help Center – *Clean Up Content for Text Area Fields After Saving a Record* (usage example) (Source: docs.oracle.com) (Source: docs.oracle.com)
- Oracle Developer Blog – "Now You Can Talk to NetSuite: Unlock Your Data with N/LLM and Intelligent Querying!" (Source: jjknowledgebase.com) (Source: jjknowledgebase.com)
- Gurus Solutions – *NetSuite N/llm Module for SuiteScript AI* (features and best practices) (Source: gurussolutions.com) (Source: gurussolutions.com)

- Ramesh Gantala Blog – *Exploring the LLM Module in SuiteScript 2.1 with a Chatbot Example* (Source: suitescriptwithramesh.blogspot.com) (Source: suitescriptwithramesh.blogspot.com)
- TechTarget – *Oracle NetSuite follows generative AI trend in ERP* (Source: www.techtarget.com)
- TechRadar – *Agentic AI: four ways it's delivering on business expectations (Nov 2025)* (Source: www.techradar.com) (Source: www.techradar.com)
- TechRadar (Netskope report) – *Average organization now reporting over 200 GenAI-related data policy violations each month (Jan 2026)* (Source: www.techradar.com) (Source: www.techradar.com)

Tags: netsuite, suitescript 2.1, n/llm module, generative ai, embeddings, oci genai, llm integration