Introduction

Coldfusion 2025.1 introduces a vendor-neutral framework for integrating generative AI into enterprise applications. This framework abstracts provider-specific complexities, enabling developers to build agents, chatbots, and semantic search pipelines using standard CFML.

Switching between models or providers is as simple as updating configuration, with no need to refactor application code.

Why use AI models

Treating AI models as infrastructure offers several advantages:

- Vendor independence: Switch providers by updating configuration.
- Data privacy: Support for local inference servers (e.g., Ollama) for sensitive data.
- Unified tooling: Standardized features like function calling and retrieval-augmented generation (RAG).
- Orchestration: Built-in support for model context protocol (MCP) for smooth integration.

Core architecture

The framework is layered:

- Chat model layer: Stateless, direct access to LLM for text generation, translation, or code assistance.
- AI service layer: Stateful wrapper managing conversation history (memory), personas (system messages), and tool execution.
- RAG pipeline: End-to-end solution for document ingestion and retrieval, including document loaders, text splitters, embedding models, and vector stores.

Supported models

Coldfusion supports a wide array of providers:

- Cloud-hosted: OpenAI, Anthropic
- Local/offline: Ollama, Mistral

Open AI

- gpt-3.5-turbo
- gpt-3.5-turbo-1106
- gpt-3.5-turbo-0125
- gpt-3.5-turbo-16k
- gpt-4
- gpt-4-0613
- gpt-4-turbo-preview
- gpt-4-1106-preview

- gpt-4-0125-preview
- gpt-4-turbo
- gpt-4-turbo-2024-04-09
- gpt-4o
- gpt-4o-2024-05-13
- gpt-4o-2024-08-06
- gpt-4o-2024-11-20
- gpt-4o-mini
- gpt-4o-mini-2024-07-18
- o1
- o3
- o3-2025-04-16
- o3-mini
- o3-mini-2025-01-31
- o4-mini
- o4-mini-2025-04-16
- gpt-4.1
- gpt-4.1-2025-04-14
- gpt-4.1-mini
- gpt-4.1-mini-2025-04-14
- gpt-4.1-nano
- gpt-4.1-nano-2025-04-14
- gpt-5
- gpt-5-mini
- gpt-5-nano
- gpt-5.1

Mistral AI

- open-mistral-7b
- open-mixtral-8x7b
- open-mixtral-8x22b
- mistral-small-latest
- mistral-medium-latest
- mistral-large-latest
- open-mistral-nemo
- codestral-latest

Pre-requisites

- Coldfusion version: 2025.0.07
- Network: Outbound access to provider APIs or local inference server.
- Credentials: API keys for cloud providers; local models may not require keys.
- After installing the update, either install the **ai** package using the ColdFusion package manageror install the **ai** package from the ColdFusion Administrator.

Chat models

The chat model is the fundamental interface for interacting with large language models (LLMs). It acts as a stateless engine that accepts a sequence of messages and returns a model-generated message. Unlike the AI service, the chat model does not maintain conversation history or manage tools automatically.

ChatModel

Description

This function helps in creating a configured chat LLM instance in ColdFusion.

Returns

A chat model instanceis an object to chatModel.chat(...). It returns a chat model instance from a given provider (OpenAI, Ollama, etc.), using the parameters you pass in. The returned object exposes chatModel.chat(...) which executes LLM calls and returns a response struct with message and metadata.

Syntax

Use a configuration struct to instantiate the model:

```
chatConfig = {
  PROVIDER : "openAi",
  APIKEY : "#application.apiKey#",
  MODELNAME : "gpt-4o-mini",
  TEMPERATURE : 0.7
};
chatModel = ChatModel(chatConfig);
```

Parameters

Configuration parameters

- Mandatory

- Conditional

- Optional

Param Name	Param Type	Description	Required	Open AI	Anthropic	Mistral	Ollama	Default / Example value
provider	String	Provider of the model	🔴	✅	✅	✅	✅	openai
baseUrl	String	The base URL for the API endpoint of the model.	🔄	✅	✅	✅	✅	Example: https://api.openai.com/v1/
apiKey	String	The authentication key required to access the model's API.	🔄	✅	✅	✅		Api-key
modelName	String	The specific identifier or name of the model to be used (e.g., "gpt-4", "gemini-pro").	🔴	✅	✅	✅	✅	gpt-4o
temperature	Number	Controls the randomness of the output (higher = more creative/random).	⭕	✅	✅	✅	✅	0.7
maxTokens	Number	maximum number of tokens to generate in the model's response.	⭕	✅	✅	✅	✅	2048
stop	Array of String	A list of strings that, if generated, will cause the model to stop generating further tokens.	⭕	✅	✅	✅	✅	["\nUser:", "###"]

timeout	Number	Maximum time (in milliseconds or seconds) to wait for a model response.	⭕	✅	✅	✅	✅	10
responseFormat	String	Specifies the desired format for the model's output (e.g., JSON, text).	⭕	✅			✅	"json"
httpClientBuilder	Struct	Configuration for the underlying HTTP client, including proxy and executor pool settings.	⭕	✅		✅	✅	
topP	Number	Filters token selection by cumulative probability; only tokens above a certain probability threshold are considered.	⭕	✅	✅	✅	✅	0.95
topK	Number	Filters token selection by choosing only the top K most likely next tokens.	⭕		✅	✅	✅	40
maxRetries	Number	The maximum number of times to retry a failed API request to the model.	⭕	✅	✅	✅		2
logRequests	Boolean	A boolean indicating whether to log the requests sent to the model.	⭕	✅	✅	✅	✅	True
logResponses	Boolean	A boolean indicating whether to log the responses received from the model.	⭕	✅	✅	✅	✅	True
thinkingType	String	Specifies the strategy or	⭕		✅	✅	✅	"enabled"

		mode the model uses for internal "thinking" or processing before generating a response.						
thinkingBudgetTokens	Number	The maximum number of tokens the model can use for its internal "thinking" process.	⭕		✅	✅		512
cacheSystemMessages	Boolean	A boolean indicating whether to cache system messages to optimize repeated interactions.	⭕		✅			NA
cacheTools	Boolean	A boolean indicating whether to cache tool definitions or outputs for efficiency.	⭕					NA
repeatPenalty	Number	A penalty applied to tokens that have already appeared in the text, discouraging repetition.	⭕	✅			Yes (as repeatPenalty)	0.5
seed	Number	A value that, when set, makes the model's output deterministic for a given input, useful for reproducibility.	⭕	✅			Yes	1337
numPredict	Number	The number of predictions or completions to generate.	⭕				Yes (as numPredict)	2000
presencePenalty	Number	A penalty applied to tokens based	⭕	✅				0.0

		on whether they are present in the text, discouraging the introduction of new topics.						
logitBias	Struct	Allows biasing the probability of specific tokens appearing or not appearing in the output.	⭕	✅				{1504: 100}
maxCompletionTokens	Number	The maximum total number of tokens expected in the entire output from the model.	⭕	✅				2048
metadata	Map<String, String>?	Additional data or context passed along with the request, often for logging or tracking.	⭕					NA
maxOutputTokens	Number	The maximum number of output tokens (often equivalent to maxTokens or maxCompletionTokens).	⭕			✅		2048
candidateCount	Number	The number of alternative response candidates the model should generate.	⭕					1
allowCodeExecution	Boolean	A boolean indicating whether the model is allowed to execute code as part of its reasoning or response generation.	⭕					False
includeCodeExecution	Boolean	A boolean indicating	⭕					False

		whether the generated response should include details of any code execution performed.						
safetySettings	Array of String	Configuration for content safety filters to prevent the generation of harmful content.	⭕					"HARM_CATEGORY_HATE_SPEECH"
version	String	The specific version of the model to be used.	⭕		✅	✅		"2023-06-01"
beta	Boolean	A flag indicating whether to use a beta or experimental version of the model or feature.	⭕		✅			False

.chat()

Once configured, send prompts using .chat() with the chat model instance returned from ChatModel. The .chat method accepts:

- Plain string
- Struct: systemMessage and userMessage

For example,

```
response = chatModel.chat("Explain quantum computing in one
sentence.");
writeOutput(response.message);
```

Or with structured messages:

```
chatRequest = {
  systemMessage: "You are a code converter.",
  userMessage: { message: "Convert 'console.log(x)' to ColdFusion." }
};
```

```
response = chatModel.chat(chatRequest);
```

Sending prompts to an LLM in ColdFusion

Introduction

Sending prompts is the core way to interact with a large language model (LLM) in ColdFusion. A prompt is a message or set of messages that tells the model what you want it to do, such as answering a question, generating text, translating, or performing a task. ColdFusion's AI framework provides a unified and flexible way to send prompts, whether you want simple Q&A or advanced, multi-turn conversations.

Prompt types

Structured chat request (AIService only)

For more control, use a chat request struct. This separates the system message (which sets the assistant's persona, rules, or tone) from the user message (the actual question or instruction).

Note: Structured chat requests are supported via AIService.chat(). They are not supported as an input to ChatModel.chat() in the current release. Use ChatModel.chat() only with a plain string prompt.

Plain string prompt
You pass a single string to the model, which treats it as a user message.

```
response = chatModel.chat("Summarize the key features of ColdFusion
2025.");
writeOutput(response.message);
```

Structured chat request
Once configured, send prompts using .chat() with the chat model instance returned from ChatModel.

The ChatModel.chat() method accepts only a plain string prompt. It does not support passing a structured chatRequest object (for example, systemMessage + userMessage) directly to ChatModel.

Use ChatModel when you want a lightweight, stateless call to the underlying provider with no built-in persona, memory, or tool orchestration. If you need structured prompts (system/user messages), personas, memory, or tools, use AIService.chat() instead.

```
response = chatModel.chat("Explain quantum computing in one
sentence.");
 writeOutput(response.message);
```

- The system message defines how the assistant should behave.
- The user message contains the prompt or question.

Use prompts with AI service

If you use an AI service (for memory, tools, or advanced features), you can send prompts in the same ways:

- As a plain string (uses the current system persona and memory)
- As a chat request struct (overrides the persona for that call)

For example,

```
aiService.systemMessage("You are a ColdFusion tutor. Explain
clearly.");
response = aiService.chat("What is a ColdFusion component?",
"user123");
```

Or, for a one-off persona:

```
chatRequest = {
  systemMessage: "You are a poet. Reply in two lines.",
  userMessage: { message: "Describe the ocean." }
};
response = aiService.chat(chatRequest, "user123");
```

Chat memory

Chat memory is a mechanism that allows conversational AI systems, like those built with ColdFusion's AI integration, to remember previous interactions within a conversation. Instead of treating every prompt as a standalone request, chat memory stores and retrieves conversation history so that each new chat request can be enriched with relevant previous messages.

This enables the AI to maintain context, recall user preferences, and behave coherently over multiple turns, which is essential for building realistic assistants, chatbots, or workflow agents.

How chat memory works in ColdFusion

- Layered architecture: In ColdFusion, chat memory is not part of the raw chat model itself. Instead, it's managed by the AiService layer, which wraps the stateless ChatModel and adds features like memory, personas, and tool calling.
- Configuration: You define chat memory using a ChatMemory struct, specifying parameters such as:
    - type: The memory strategy (e.g., message window, token window, or custom implementation).
    - maxMessages: How many messages to keep in memory.
    - perUser: Whether memory is global or segmented per user/session.
    - persistenceStore: Where to persist chat history (e.g., Redis, Memcache, Ehcache).

Why is chat memory important?

- Contextual Conversations: Without memory, the AI cannot "remember" earlier questions, user preferences, or intermediate workflow steps. With memory, it can answer follow-up questions, recall details, and provide a much more natural conversational experience.
- Multi-user Support: By configuring per-user memory, each user's conversation remains isolated, ensuring privacy and personalized context.

Example

Suppose a user tells the AI, "My name is Alice and I live in Paris." Later, the user asks, "What is my name and where do I live?" With chat memory enabled, the AI can recall and answer correctly. If per-user memory is set, each user's history is kept separate.

```
chatMemory = {
```

```
type: "messageWindowChatMemory",
maxMessages: 20,
perUser: true,
persistentStore : "Redis/Memcached/MyChatPersistencImpl.cfc"
};
```

Function tools

Function tools are a mechanism that allows Large Language Models (LLMs) integrated into ColdFusion to call your own business logic, specifically, ColdFusion CFC methods or MCP clients, during a conversation. Instead of only generating text, the AI can trigger real actions, fetch data, or perform computations by invoking these tools.

How do function tools work?

- Declaration: You register function tools in the AiService configuration using a TOOLS array. Each entry specifies which CFC (ColdFusion Component) and which methods are exposed to the AI.
- Schema: Each tool has a name, description, and a JSON schema describing its parameters. This schema tells the LLM what the tool does and how to use it.
- Invocation: During a chat, the LLM may decide it needs to call a tool (e.g., “getTicketStatus” for a support ticket). The integration layer sends a structured request to your CFC, gets the result, and passes it back to the model, which then continues the conversation.
- Description: For each tool method you register, always provide a clear, scenario-based DESCRIPTION. This tells the AI exactly when and why to use the tool. Without a descriptive scenario, the AI may not invoke the tool even if the user’s query matches the intent. For example,
    - "Get the status of a support ticket by ID. Use this when the user asks about the progress or state of a ticket, e.g., 'What is the status of ticket TKT-12345?'"
    - "Create a new support ticket from user input. Use this when the user wants to report an issue or request support."

Why are function tools important?

- Enterprise integration: They allow your AI assistant to interact with real business systems, fetching account balances, creating tickets, validating data, or triggering workflows.

- Enhanced capabilities: The AI can go beyond text generation, performing tasks that require up-to-date or sensitive information from your internal systems.
- Controlled access: You decide which functions are exposed, ensuring security and relevance.

Example

Suppose you expose a support tool with two methods:

- getTicketStatus(ticketId): Returns the status of a support ticket.
- createTicket(summary, priority): Creates a new support ticket.

When a user asks, “What is the status of ticket TKT-12345?”, the LLM can invoke getTicketStatus and respond with the actual status from your system.

ticket.cfm

```
<cfscript>
    // 1. Create the underlying ChatModel
    chatModelConfig = {
        PROVIDER  : "openAi",
        APIKEY    : "#application.apiKey#",
        MODELNAME : "gpt-4o-mini"
    };
    chatModel = ChatModel(chatModelConfig);

    // 2. Configure AiService with ChatMemory and Tools
    aiServiceConfig = {
        CHATMODEL  : chatModel,
        CHATMEMORY : { MAXMESSAGES : 20, PERUSER : true },
        TOOLS      : [
            {
                CFC     : "tool.SupportTool",
                METHODS : [
                    { METHOD : "getTicketStatus", DESCRIPTION : "Get the status
of a support ticket by ID" },
                    { METHOD : "createTicket",    DESCRIPTION : "Create a new
support ticket from user input" }
                ]
            }
        ]
    };
    aiService = AIService(aiServiceConfig);
```

```
    // 3. Use AiService in a conversation
    response = aiService.chat("My ticket ID is TKT-12345. What is its status?",
"user1");
    writeDump(response.message);
</cfscript>
```

SupportTool.cfc

```
component output=false {

    public string function getTicketStatus(required string ticketId) {
        // Example lookup; real implementation would query a database or API
        if (arguments.ticketId == "TKT-12345") {
            return "in progress";
        } else {
            return "closed";
        }
    }

    public struct function createTicket(
        required string summary,
        required string priority
    ) {
        var ticket = {
            id       : createUUID(),
            summary  : arguments.summary,
            priority : arguments.priority,
            status   : "new"
        };

        // Persist ticket in your system, then return a simple summary
        return ticket;
    }

}
```

Best practice

Always include a descriptive scenario in the DESCRIPTION field for each tool method. This ensures the AI knows exactly when to use the tool in response to user queries, improving reliability and user experience.

AIService

The AIService function in ColdFusion is designed to turn a stateless chat model (created with ChatModel) into a full conversational AI service. This means it adds advanced features like:

- Multi-turn conversation management: Handles ongoing dialogue, not just single prompts.
- Memory: Remembers previous messages, either globally or per user, and can persist this history using stores like Redis or Memcache.
- System personas: Allows you to set instructions or roles for the AI, such as “You are a travel assistant.”
- Function/tool calling: Lets the AI invoke ColdFusion functions (CFC methods) or external MCP clients to perform tasks, fetch data, or trigger actions.

How does it work?

- You pass a configuration struct to AIService, which must include a chatModel (the LLM instance).
- Optionally, you can add:
    - tools: A list of ColdFusion functions or MCP clients the AI can use.
    - chatMemory: Configuration for how conversation history is stored and managed.

The result is an AI service instance that provides methods like:

- aiService.chat(message): Sends a message with the current persona and memory context.
- aiService.chat(message, userId): Same as above, but memory is scoped to a specific user/session.
- aiService.chat(chatRequestStruct): Allows explicit system and user messages.
- aiService.chat(chatRequestStruct, userId): Combines rich requests with per-user memory.

You can also set a system-level instruction for all subsequent calls using aiService.systemMessage(messageString).

Example

```
<cfscript>
```

```
writeOutput("<h1>Example 2: AI Service Basics</h1>");

// Setup
chatModelConfig = {
    PROVIDER : "openAi",
    APIKEY : "api-key",
    MODELNAME : "gpt-4o-mini"
};

chatModel = ChatModel(chatModelConfig);

aiServiceConfig = {
    CHATMODEL : chatModel
};

aiService = AIService(aiServiceConfig);

// Method 1: Plain string
writeOutput("<h2>1. Plain String</h2>");
response1 = aiService.chat("What is the capital of Japan?");
writeDump(var=response1.message);

writeOutput("<hr>");

// Method 2: chatRequest struct
writeOutput("<h2>2. chatRequest Struct</h2>");
chatRequest = {
    SYSTEMMESSAGE : "You are a poet. Reply in 2 lines.",
    USERMESSAGE : { MESSAGE : "Tell me about the ocean." }
};
response2 = aiService.chat(chatRequest);
writeDump(var=response2.message);

writeOutput("<hr>");

// Method 3: System message via function
writeOutput("<h2>3. systemMessage() Function</h2>");
aiService.systemMessage("You are a scientist. Be concise.");
response3 = aiService.chat("Why is the sky blue?");
writeDump(var=response3.message);

writeOutput("<hr>");
```

```
    // Method 4: Override system message
    writeOutput("<h2>4. Override System Message</h2>");
    chatRequest2 = {
        SYSTEMMESSAGE : "You are a pirate. Use pirate speak.",
        USERMESSAGE : { MESSAGE : "What is gravity?" }
    };
    response4 = aiService.chat(chatRequest2);
    writeDump(var=response4.message);

</cfscript>
```