



Developer Guide for Amazon Nova 2

# Amazon Nova



## Amazon Nova: Developer Guide for Amazon Nova 2

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

---

# Table of Contents

<b>What is Amazon Nova 2?</b> .....	1
Key concepts .....	1
Amazon Nova models .....	1
What can you build? .....	2
Benefits .....	2
How Amazon Nova works .....	3
Pricing .....	4
Next steps .....	4
<b>What's new in Amazon Nova 2</b> .....	5
New models .....	5
New features and capabilities .....	5
Nova Multimodal Embeddings .....	5
Extended thinking and reasoning .....	5
Built-in tools .....	6
AI agent building .....	6
Improved document understanding .....	6
Enhanced video understanding .....	6
Model customization .....	6
Amazon Nova Forge .....	6
Next steps .....	7
<b>Getting started</b> .....	8
Getting started in the console .....	8
Explore the text playground .....	9
Getting started with the API .....	9
Get credentials to grant programmatic access .....	11
Attach Amazon Bedrock permissions to a user or role .....	12
Generate a response for a text prompt using an Amazon Nova model .....	13
<b>Core inference</b> .....	14
Important: Timeout Configuration .....	15
Core Inference Topics .....	15
Using the Converse API .....	16
Request structure .....	16
Using system prompts .....	17
Inference parameters .....	18

Using reasoning .....	19
Using the Invoke API .....	21
Request structure .....	22
Request parameters .....	23
Streaming responses .....	23
Streaming with ConverseStream .....	23
Streaming with InvokeModelWithResponseStream .....	24
Stream event types .....	24
Using Amazon Nova embeddings .....	25
Key features .....	25
Generating embeddings .....	26
On-demand inference .....	29
Benefits .....	29
Using on-demand inference .....	30
Pricing .....	30
Quotas and limits .....	30
<b>Prompt engineering .....</b>	<b>31</b>
What is prompt engineering .....	31
Getting started with prompt engineering .....	31
Best practices .....	32
Understanding the roles .....	32
Create precise prompts .....	33
Bring focus to sections of the prompt .....	51
Using the system role .....	53
Provide examples (few-shot prompting) .....	61
Tool calling systems .....	71
Advanced prompting techniques .....	72
Prompting multimodal inputs .....	94
General multimodal guidelines .....	94
Document and Image Understanding .....	95
Video understanding .....	103
<b>Using Nova capabilities .....</b>	<b>107</b>
Reasoning .....	107
Multimodal understanding .....	108
Supported-content-type-by-modality .....	108
Image understanding .....	109

Video understanding .....	103
Document understanding .....	114
Examples: Using Nova's document understanding via API and S3 .....	115
<b>Speech-to-Speech .....</b>	<b>116</b>
Key features .....	116
Getting started .....	117
State the imports and configuration .....	117
Define the SimpleNovaSonic class .....	118
Initialize the client .....	118
Handle events .....	119
Start the session .....	119
Handle audio input .....	122
End the session .....	123
Handle responses .....	124
Playback audio .....	126
Capture audio .....	126
Run the main function .....	127
Code examples .....	129
Getting started examples .....	129
Advanced use cases .....	130
Hands-on workshop .....	131
Voice conversation prompts .....	131
Important Guidelines .....	131
Recommended Baseline System Prompt for Voice .....	132
Speech Prompt Configuration .....	132
System Prompt Configuration .....	132
Gender Agreement for Gendered Languages .....	133
Chain of thought for Speech: Constitutional Reasoning .....	134
Overuse of suggested phrases .....	135
Core concepts .....	137
Event lifecycle .....	137
Event flow sequence .....	139
Handling input events with the bidirectional API .....	139
Handling output events with the bidirectional API .....	148
Barge-in .....	155
Turn-taking controllability .....	156

Cross-modal input .....	158
Language support and multilingual capabilities .....	161
Managing chat history .....	162
Tool configuration .....	168
Asynchronous tool calling .....	175
<b>Integrations .....</b>	<b>178</b>
Strands Agents .....	178
Framework integrations .....	181
<b>Advanced systems with Nova .....</b>	<b>192</b>
Overview .....	192
Extended Thinking .....	192
How Extended Thinking Works .....	192
Configuration Options .....	195
Supported models .....	196
Extended thinking with tool use .....	196
Understanding Reasoning Tokens and Pricing .....	196
Frequently Asked Questions .....	197
Tool use .....	197
Understanding the tool use workflow .....	198
Create a tool .....	198
Call a tool .....	201
Built-in system tools .....	205
Building AI agents .....	207
Create an agent .....	207
Invoke an agent .....	209
Web Grounding .....	212
How Web Grounding works .....	212
How to use Web Grounding .....	212
Regional availability .....	214
Response structure .....	214
Grounding safety .....	215
Error handling .....	215
Permissions required for built in tools .....	215
<b>Customizing models .....</b>	<b>217</b>
Customization overview .....	217
Customization platform availability .....	217

Customization on Amazon Bedrock .....	218
Customization on SageMaker AI .....	219
Fine-tuning Amazon Nova models .....	220
Available models .....	221
On-demand inference .....	221
Compatibility requirements .....	221
Model training and inference .....	222
Rate limits .....	222
Latency .....	222
<b>Code and Troubleshooting .....</b>	<b>223</b>
Code library .....	223
Converse API Examples .....	223
InvokeModel API Examples .....	243
Troubleshooting .....	248
Authentication and setup .....	248
Model access denied .....	249
Regional availability issues .....	249
Timeout configuration .....	249
API response issues .....	250
Reasoning mode errors .....	251
Tool use issues .....	252
<b>API and SDK reference .....</b>	<b>256</b>
Request and response schema .....	256
Complete request structure .....	256
Complete response structure .....	261
SDK reference .....	264
<b>Nova Forge .....</b>	<b>265</b>
Key benefits .....	265
Setup access checkpoints across all phases of model development .....	265
Blend your proprietary data with Amazon Nova-curated training data .....	265
Perform reinforcement learning with reward functions in your environment .....	265
Use the built-in responsible AI toolkit to implement safety guardrails .....	265
Connect with experts at the AWS Generative AI Innovation Center .....	266
<b>Responsible use .....</b>	<b>267</b>
Guidelines .....	267
Recommendations .....	268

<b>Quotas</b> .....	<b>270</b>
Request an increase for Amazon Nova quotas .....	270
<b>Monitoring</b> .....	<b>272</b>
<b>Security</b> .....	<b>273</b>
<b>Create resources with CloudFormation</b> .....	<b>274</b>
Learn more about CloudFormation .....	274
<b>Document history</b> .....	<b>275</b>

# What is Amazon Nova 2?

Amazon Nova provides multimodal foundation models that process text, images, video, documents and speech. With support for up to 1 million tokens of context and advanced reasoning capabilities, Amazon Nova 2 models enable you to build sophisticated AI applications that understand complex inputs and generate accurate responses.

You can build interactive chatbots, analyze documents and videos, create AI agents with extended reasoning and develop voice-enabled applications.

## Key concepts

Before you learn about Amazon Nova models, familiarize yourself with the following core concepts:

### Foundation models

Pre-trained AI systems available in different sizes and capabilities that you access through an API.

### Inference

The process of sending a request to a model and receiving a generated response.

### Reasoning

Extended thinking capability that allows models to break down complex problems and show their step-by-step analysis before providing answers.

### Multimodal

The ability to process and understand multiple input types together: text, images, video and documents in a single request.

### RAG (Retrieval-Augmented Generation)

A technique that combines model responses with your own data sources to provide more accurate, contextual answers.

## Amazon Nova models

Amazon Nova 2 includes the following models, each optimized for different use cases:

Model	Input modalities	Output modalities	Use cases
Nova 2 Lite	Text, images, video, documents	Text	High-volume applications prioritizing speed and cost efficiency
Nova 2 Sonic	Speech, text	Speech, text	Voice-enabled applications with fast response times
Nova Multimodal Embeddings	Text, images, documents, video, audio	Embeddings	Semantic search, recommendation systems and similarity matching

All models support up to 1 million tokens of context and can generate up to 65,536 tokens in a single response. Models with reasoning capabilities can perform extended thinking to solve complex problems step by step.

## What can you build?

The following are examples of what you can build with Amazon Nova:

- **Intelligent document assistant** - Process large documents with up to 1 million tokens of context to answer questions and extract insights (with RAG)
- **Complex reasoning applications** - Solve multi-step problems with extended thinking that shows the model's step-by-step analysis (or with reasoning)
- **Video analysis pipeline** - Extract insights, generate summaries and identify key moments in video content at scale (Nova 2 Lite)
- **Voice-enabled AI agent** - Build conversational agents that understand speech input and respond with natural language (Nova 2 Sonic)

## Benefits

Amazon Nova provides the following benefits:

## Multimodal understanding

Process text, images, video, documents and speech in a single request. Amazon Nova models understand relationships across different input types.

## Extended context

Support for up to 1 million tokens allows you to process entire codebases, lengthy documents, or extended conversations without losing context.

## Advanced reasoning

Models with reasoning capabilities break down complex problems and show step-by-step analysis, improving accuracy for multi-step tasks.

## Flexible deployment

Access models through Amazon Bedrock with no infrastructure to manage, or customize models through fine-tuning and reinforcement learning.

## Built-in tools

Use web grounding to access real-time information and code interpreter to execute Python code without external integrations.

# How Amazon Nova works

Amazon Nova models are foundation models that you access through Amazon Bedrock. The basic workflow is:

1. Your application sends a request to Amazon Bedrock with your input and configuration parameters.
2. The Amazon Nova model processes your input, applying reasoning if configured.
3. The model generates a response and returns it to your application.

You can enhance responses by using RAG to incorporate your data, enabling built-in tools, or customizing models through fine-tuning.

## Pricing

Amazon Nova pricing is based on input and output tokens processed. Different models have different pricing tiers:

- Nova 2 Lite - Optimized for cost-effective, high-volume processing
- Nova 2 Sonic - Balanced pricing for voice-enabled applications

For current pricing information, see [Amazon Bedrock Pricing](#).

## Next steps

- To learn about new features in Amazon Nova 2, see [What's new in Amazon Nova 2](#).
- To start using Amazon Nova, see [Getting started with Amazon Nova 2](#).
- To learn about core inference capabilities, see [Core inference](#).
- To customize models for your use case, see [Customizing Amazon Nova 2.0 models](#).

# What's new in Amazon Nova 2

Amazon Nova 2 introduces significant enhancements across understanding, creative and speech capabilities. The following sections describe the key new features and improvements.

## New models

### Nova 2 Lite

Enhanced models that offer advanced reasoning with extended thinking support, three-level intensity control and multimodal understanding optimized for production-scale agentic workflows.

### Nova 2 Sonic

An upgraded conversational speech model with improved speech understanding, natural language processing and voice generation capabilities across seven languages.

### Nova Multimodal Embeddings

A multimodal embedding model that processes text, image, document, video and audio inputs and generates embeddings.

## New features and capabilities

### Nova Multimodal Embeddings

Nova Multimodal Embeddings supports text, documents, images, video and audio through a single model, enabling cross-modal retrieval applications. Nova Multimodal Embeddings maps each of these content types into a unified semantic space, enabling you to conduct unimodal, cross-modal and multimodal vector operations, powering applications such as agentic retrieval-augmented generation (RAG) and multimodal semantic search.

### Extended thinking and reasoning

Nova 2 Lite supports extended thinking, which allows the models to spend more time reasoning through complex problems before generating responses. This capability improves accuracy for multi-step reasoning tasks, including agentic workflows with multiple tools, advanced mathematics, complex planning and code generation.

To learn more, navigate to [Extended thinking in Amazon Nova 2.](#)

## Built-in tools

Amazon Nova 2 includes built-in tools that extend model capabilities without requiring external integrations:

- **Web grounding** – Accesses real-time information from the web to provide up-to-date responses and reduce hallucinations.
- **Code interpreter** – Executes Python code to perform calculations.

## AI agent building

Amazon Nova 2 models are optimized for building AI agents. The models provide improved tool use, better reasoning for multi-step tasks and enhanced ability to maintain context across complex agent workflows.

## Improved document understanding

Nova 2 Lite provides enhanced document processing capabilities with better understanding of complex document layouts, tables, charts and multi-page documents. The models extract information more accurately from PDFs, spreadsheets and other document formats.

## Enhanced video understanding

Nova 2 Lite offers improved video analysis capabilities, including better visual perception, temporal understanding, action recognition and the ability to process longer video sequences with higher accuracy.

## Model customization

Nova 2 Lite supports supervised fine-tuning (SFT) and reinforcement fine-tuning (RFT) on Amazon Bedrock and SageMaker AI AI, allowing you to adapt Amazon Nova 2 to your specific business needs.

## Amazon Nova Forge

Amazon Nova Forge is a first-of-its-kind service that offers organizations the easiest and most cost-effective way to build their own frontier models using Amazon Nova.

## Next steps

- To learn about Amazon Nova models and capabilities, see [What is Amazon Nova 2?](#).
- To start using Amazon Nova 2.0, see [Getting started with Amazon Nova 2.](#)
- To explore core inference features, see [Core inference](#).

# Getting started with Amazon Nova 2

You can invoke Amazon Nova 2 models through the following methods:

- Amazon Bedrock [InvokeModel](#) or [Converse](#) API operations
- Amazon Bedrock [Chat and Text playgrounds](#) for Nova 2 Lite, Nova 2 Sonic

## Important

If you're new to AWS or to Amazon Bedrock, you must first set up Amazon Bedrock by following the steps at [Getting started with Amazon Bedrock](#). Alternatively, you can try the Amazon Nova models without entering Amazon Bedrock at [nova.amazon.com](#) or start building for free at [nova.amazon.com/dev](#).

Select a topic to get started on using Amazon Nova with that method.

## Getting started with Amazon Nova in the Amazon Bedrock console

This section describes how to use the playgrounds in the AWS console to submit a text prompt to Amazon Nova 2 models and generate a text or image response. Before you run the following examples, you should check that you have fulfilled the following prerequisites:

### Prerequisites

- You have an AWS account and have permissions to access a role in that account with the necessary permissions for Amazon Bedrock. Otherwise, follow the steps at [Getting started with Amazon Bedrock](#).
- If you are accessing the model from a US region, you must use the US CRIS endpoint which involves adding the us prefix to the model ID (such as us.amazon.nova-2-lite-v1:0).
- If you are accessing the model from outside of the US, you can either use the global CRIS endpoint (such as global.amazon.nova-2-lite-v1:0) or prefix the region in the model ID (such as us/eu/jp).

## Explore the text playground

The following example demonstrates how to use the text playground:

1. Open the Amazon Bedrock console at <https://console.aws.amazon.com/bedrock/>.
2. From the left navigation pane, choose **Chat / Text** under **Playgrounds**.
3. Choose **Select model** and select a provider and model. For this example, select **Amazon** then select a Amazon Nova 2 model (such as Nova 2 Lite, or Nova 2 Sonic). Then choose **Apply**.
4. Select a default prompt from below the text panel, or enter a prompt into the text panel, such as **Describe the purpose of a "hello world" program in one line**.
5. To explore the image understanding capabilities of Amazon Nova, you can upload an image in JPEG, PNG, GIF, or WEBP format that is less than or equal to 25 MB from your computer. After the image is uploaded, you can ask Amazon Nova about the image.
6. To explore the document understanding capabilities of Amazon Nova, you can upload documents in CSV, DOC, DOCX, HTML, MD, PDF, TXT, XLS, or XLSX format that are less than or equal to 4.5 MB. After the documents are uploaded, you can ask Amazon Nova about the documents.
7. To explore the video understanding capabilities of Amazon Nova, you can upload one video in MKV, MOV, or MP4 format that is less than or equal to 25 MB from your computer. You can use Amazon S3 for videos up to 1 GB. After the video is uploaded, you can ask Amazon Nova about the video.
8. Choose **Run** to run inference on the model. The generated text appears below your prompt in the text panel.

## Getting started with the API

Before you get started with APIs, follow these procedures if you are new to AWS or need to install the AWS CLI or an AWS SDK. If neither of these apply to you, move to [Get credentials to grant programmatic access](#).

### I'm new to AWS

If you do not have an AWS account, complete the following steps to create one. Alternatively, you can start building for free with Amazon Nova at [nova.amazon.com/dev](http://nova.amazon.com/dev).

## To sign up for an AWS account

1. Open <https://portal.aws.amazon.com/billing/signup>.
2. Follow the online instructions.

Part of the sign-up procedure involves receiving a phone call or text message and entering a verification code on the phone keypad.

When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user and use only the root user to perform [tasks that require root user access](#).

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to <https://aws.amazon.com/> and choosing **My Account**.

## Secure your AWS account root user

1. Sign in to the [AWS Management Console](#) as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

For help signing in by using root user, see [Signing in as the root user](#) in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

For instructions, see [Enable a virtual MFA device for your AWS account root user \(console\)](#) in the *IAM User Guide*.

## I need to install the AWS CLI or an AWS SDK

To install the AWS CLI, follow the steps at [Install or update to the latest version of the AWS CLI](#).

To install an AWS SDK, select the tab that corresponds to the programming language that you want to use at [Tools to Build on AWS](#).

AWS software development kits (SDKs) are available for many popular programming languages. Each SDK provides an API, code examples and documentation that make it easier for developers

to build applications in their preferred language. SDKs automatically perform useful tasks for you, such as:

- Cryptographically sign your service requests
- Retry requests
- Handle error responses

## Get credentials to grant programmatic access

Users need programmatic access if they want to interact with AWS outside of the AWS Management Console. The way to grant programmatic access depends on the type of user that's accessing AWS.

To grant users programmatic access, choose one of the following options.

Which principal needs programmatic access?	To	By
IAM users	Limit the duration of long-term credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	<p>Following the instructions for the interface that you want to use.</p> <ul style="list-style-type: none"><li>• For the AWS CLI, see <a href="#">Authenticating using IAM user credentials</a> in the <i>AWS Command Line Interface User Guide</i>.</li><li>• For AWS SDKs and tools, see <a href="#">Authenticate using long-term credentials</a> in the <i>AWS SDKs and Tools Reference Guide</i>.</li><li>• For AWS APIs, see <a href="#">Managing access keys for IAM users</a> in the <i>IAM User Guide</i>.</li></ul>

Which principal needs programmatic access?	To	By
IAM roles	Use temporary credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Follow the instructions in <a href="#">Using temporary credentials with AWS resources</a> in the <i>IAM User Guide</i> .
Users managed in IAM Identity Center	Use temporary credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	<p>Following the instructions for the interface that you want to use.</p> <ul style="list-style-type: none"> <li>For the AWS CLI, see <a href="#">Configuring the AWS CLI to use IAM Identity Center</a> in the <i>AWS Command Line Interface User Guide</i>.</li> <li>For AWS SDKs, tools and AWS APIs, see <a href="#">IAM Identity Center authentication</a> in the <i>AWS SDKs and Tools Reference Guide</i>.</li> </ul>

## Attach Amazon Bedrock permissions to a user or role

After setting up credentials for programmatic access, you need to configure permissions for a user or IAM role to have access to Amazon Bedrock-related actions. To set up these permissions, do the following:

1. On the AWS Management Console Home page, select the IAM service or navigate to the IAM console at <https://console.aws.amazon.com/iam/>.
2. Select **Users or Roles** and then select your user or role.
3. In the **Permissions** tab, choose **Add permissions** and then choose **Add AWS managed policy**. Choose the [AmazonBedrockFullAccess](#) AWS managed policy.
4. To allow the user or role to subscribe to models, choose **Create inline policy** and then specify the following permissions in the JSON editor:

{

```
"Version": "2012-10-17",
"Statement": [
    {
        "Sid": "MarketplaceBedrock",
        "Effect": "Allow",
        "Action": [
            "aws-marketplace:ViewSubscriptions",
            "aws-marketplace:Unsubscribe",
            "aws-marketplace:Subscribe"
        ],
        "Resource": "*"
    }
]
```

## Generate a response for a text prompt using an Amazon Nova model

After you've fulfilled all the prerequisites, you can test making model invocation requests to Amazon Nova models with a [Converse](#) request.

To install the AWS CLI, follow the steps at [Install or update to the latest version of the AWS CLI](#). Verify that you've set up your credentials to use Boto3 by following the steps at [Get credentials to grant programmatic access](#).

To generate a response for a text prompt in Nova 2 Lite by using the AWS CLI, run the following command in a terminal:

```
aws bedrock-runtime converse \
--model-id us.amazon.nova-2-lite-v1:0 \
--messages '[{"role":"user","content":[{"text":"Write a short poem"}]}]' \
--additional-model-request-fields '{"reasoningConfig": \
{"type":"enabled","maxReasoningEffort":"low"}}'
```

# Core inference

Inference is the process of sending a request to a Amazon Nova model and receiving a generated response. Amazon Nova models support inferencing through two API options:

- **Converse API ([Converse](#), [ConverseStream](#))**: Provides a consistent interface across different models, making it easier to switch between models or build applications that work with multiple models. Recommended for most use cases.
- **Invoke API ([InvokeModel](#), [InvokeModelWithResponseStream](#))**: Request payloads are structured specifically for each model's native format. Runs inference using the prompt and inference parameters provided in the request body.

Both APIs support the same core features including:

- Multi-turn conversations
- Multimodal inputs (text, images, video, audio)
- Tool use
- Guardrails
- Streaming responses

The request structure is nearly identical between the two APIs, differing only in how byte data (documents, images, video and audio) is encoded. For model request parameters unique to Amazon Nova models, such as `reasoningConfig` and `TopK`, these are placed within an additional `inferenceConfig` object within the `additionalModelRequestFields`. These are top-level parameters for `InvokeModel` and `InvokeModelWithResponseStream`.

 **Note**

Review [Code Samples](#) and [Troubleshooting](#) for a list of code samples with Nova 2 models.

Set the `modelId` to one of the following to use Amazon Nova models:

Model	Model ID
Nova 2 Lite	<ul style="list-style-type: none"><li>• <code>global.amazon.nova-2-lite-v1:0</code></li></ul>

Model	Model ID
Nova 2 Sonic	<ul style="list-style-type: none"><li>us.amazon.nova-2-lite-v1:0</li><li>global.amazon.nova-2-sonic-v1:0</li><li>us.amazon.nova-2-sonic-v1:0</li></ul>
Nova Multimodal Embeddings	amazon.nova-2-multimodal-embeddings-v1:0

## Important: Timeout Configuration

### Important

Amazon Nova inference requests can take up to 60 minutes to complete. Configure your client timeout settings accordingly:

The following example is Python code. Users can check the documentation for their preferred SDK language version in that SDK's API docs.

```
from botocore.config import Config

bedrock = boto3.client(
    'bedrock-runtime',
    region_name='us-east-1',
    config=Config(
        read_timeout=3600  # 60 minutes
    )
)
```

## Core Inference Topics

This section discusses the following topics:

- Using the converse API: A consistent interface offering compatibility across most Bedrock models
- Using the invoke API: An interface unique to Nova models and not portable to other Bedrock models
- Streaming responses: Real-time response generation

- Using Amazon Nova embeddings: Text embeddings capabilities
- On-demand inference: Pay-per-use inference model

## Using the Converse API

The Converse API provides a unified interface for interacting with Amazon Nova models. It abstracts away model-specific details and provides a consistent way to handle multi-turn conversations, system prompts and streaming responses across all Amazon Nova models.

### Topics

- [Request structure](#)
- [Using system prompts](#)
- [Inference parameters](#)
- [Using reasoning](#)

## Request structure

- **Multi-turn conversations:** Maintain context across multiple exchanges
- **System prompts:** System instructions such as personas or response guidelines
- **Document chat:** Interact with and query documents or collections of documents
- **Vision:** Process and analyze images and video
- **Tool use:** Enable models to use external tools and APIs
- **Guardrails:** Apply content filtering and safety controls
- **Reasoning:** Extended thinking for complex problem-solving

A basic Converse API request includes the model ID and a list of messages:

```
import boto3

bedrock = boto3.client('bedrock-runtime', region_name='us-east-1')

response = bedrock.converse(
    modelId='us.amazon.nova-2-lite-v1:0',
    messages=[
```

```
        'role': 'user',
        'content': [{'text': 'What is machine learning?'}]
    }
]

content_list = response["output"]["message"]["content"]
# Extract the first text block
text = next((item["text"] for item in content_list if "text" in item), None)
if text is not None:
    print(text)
```

## Using system prompts

System prompts provide context and instructions to the model:

```
import boto3

bedrock = boto3.client('bedrock-runtime', region_name='us-east-1')

response = bedrock.converse(
    modelId='us.amazon.nova-2-lite-v1:0',
    system=[
        {'text': 'You are a helpful AI assistant specializing in cloud computing.'}
    ],
    messages=[
        {
            'role': 'user',
            'content': [{'text': 'Explain serverless computing.'}]
        }
    ]
)

# Print the response text
content_list = response["output"]["message"]["content"]
text = next((item["text"] for item in content_list if "text" in item), None)
if text is not None:
    print(text)
```

## Inference parameters

Control the model's output using inference parameters. The following are available inference parameters.

- **maxTokens** (integer): Maximum number of tokens to generate (up to 65,000). If not specified, the model uses a dynamic default based on the request context.
- **temperature** (float): Controls randomness (0.0-1.0, default 0.7). Lower values make output more deterministic
- **topP** (float): Nucleus sampling threshold (0-1, default 0.9). Lower values make output more focused
- **stopSequences** (array): Sequences of characters that stop generation when encountered

Example:

```
import boto3
bedrock = boto3.client('bedrock-runtime', region_name='us-east-1')

response = bedrock.converse(
    modelId='us.amazon.nova-2-lite-v1:0',
    messages=[
        {
            'role': 'user',
            'content': [{'text': 'Write a short story.'}]
        }
    ],
    inferenceConfig={
        'maxTokens': 512,
        'temperature': 0.7,
        'topP': 0.9,
        'stopSequences': ['END']
    }
)

content_list = response["output"]["message"]["content"]
text = next((item["text"] for item in content_list if "text" in item), None)
if text is not None:
    print(text)
```

## Using reasoning

Nova 2 Lite supports extended thinking for complex problem-solving. Enable reasoning with `reasoningConfig`.

By default, reasoning is disabled to optimize for speed and cost on simple queries. When you need to go beyond these straightforward tasks, you can enable reasoning. Nova 2 provides flexible control over reasoning depth through three effort levels:

Low effort (`maxReasoningEffort: "low"`)

**BEST FOR: TASKS WITH ADDED COMPLEXITY REQUIRING STRUCTURED THINKING.** For example, you can use this for code review and improvement suggestions where the model needs to carefully consider existing code quality, analysis tasks that require thoughtful consideration of multiple factors, or problem-solving scenarios that benefit from a methodical approach. Low effort is ideal for compound tasks where basic reasoning improves accuracy without requiring deep multi-step planning.

Medium effort (`maxReasoningEffort: "medium"`)

**BEST FOR: MULTI-STEP TASKS AND CODING WORKFLOWS.** For example, you can use this for software development and debugging where the model needs to understand existing code structure before implementing changes, code generation that requires coordination across multiple files or components, multi-step calculations with interdependencies, or planning tasks with multiple constraints. Medium effort is optimal for agentic workflows that coordinate multiple tools and require the model to maintain context across several sequential operations.

High effort (`maxReasoningEffort: "high"`)

**BEST FOR: STEM REASONING AND ADVANCED PROBLEM-SOLVING.** For example, you can use this for advanced mathematical problems and proofs that require rigorous step-by-step verification, scientific analysis and research tasks demanding deep investigation, complex system design with architectural considerations across multiple dimensions, or critical decision-making scenarios with significant implications. High effort delivers maximum accuracy for tasks requiring sophisticated reasoning, careful evaluation of alternatives, and thorough validation of conclusions.

The following examples show different reasoning effort levels:

## Low effort

```
import boto3

bedrock = boto3.client('bedrock-runtime', region_name='us-east-1')

response = bedrock.converse(
    modelId='us.amazon.nova-2-lite-v1:0',
    system=[{"text": "You are a highly capable personal assistant"},],
    messages=[{
        "role": "user",
        "content": [{"text": "Provide a meal plan for a gluten free family of 4."}]
    }],
    inferenceConfig={
        "temperature": 0.7,
        "topP": 0.9,
        "maxTokens": 10000
    },
    additionalModelRequestFields={
        "reasoningConfig": {
            "type": "enabled",
            "maxReasoningEffort": "low"
        }
    }
)
```

## Reasoning parameters:

The following are reasoning parameters

- **type:** enabled or disabled (default: disabled)
- **maxReasoningEffort:** low, medium, or high. This is required when reasoning is enabled.

 **Note**

Temperature, topP and topK cannot be used with maxReasoningEffort set to high. This will cause an error.

The response includes reasoning content:

```
{  
    "output": {  
        "message": {  
            "role": "assistant",  
            "content": [  
                {  
                    "reasoningContent": {  
                        "reasoningText": {  
                            "text": "[REDACTED]"  
                        }  
                    }  
                },  
                {  
                    "text": "Based on the premises, we can conclude..."  
                }  
            ]  
        }  
    },  
    "stopReason": "end_turn"  
}
```

### Note

With Amazon Nova 2, reasoning content displays as [REDACTED]. You're still charged for reasoning tokens as they contribute to improved output quality. We include this field in the response structure now to preserve the option of exposing reasoning content in the future. We are actively working with customers to determine the best approach for surfacing the model's reasoning process.

## Using the Invoke API

The Invoke API provides direct access to Amazon Nova models with more ability to control the request and response format. Unlike the Converse API which abstracts model-specific details, the Invoke API allows you to work directly with the model's native request and response structures.

**Note**

The Invoke API supports the same features as the Converse API except for document input modality, which is specific to the Converse API.

## Request structure

An Invoke API request requires the model ID and a JSON request body:

```
import boto3
import json

bedrock = boto3.client('bedrock-runtime', region_name='us-east-1')

request_body = {
    'messages': [
        {
            'role': 'user',
            'content': [{'text': 'What is machine learning?'}]
        }
    ],
    'inferenceConfig': {
        'maxTokens': 512,
        'temperature': 0.7
    }
}

response = bedrock.invoke_model(
    modelId='us.amazon.nova-2-lite-v1:0',
    body=json.dumps(request_body)
)

response_body = json.loads(response['body'].read())
content_list = response_body["output"]["message"]["content"]
# Extract the first text block
text_block = next((item for item in content_list if "text" in item), None)
if text_block is not None:
    print(text_block["text"])
```

## Request parameters

The Invoke API supports the following key parameters:

- **messages**: Array of conversation messages with role and content
- **system**: Optional system prompt for context and instructions
- **inferenceConfig**: Parameters controlling model output (temperature, maxTokens, topP, topK, stopSequences, reasoningConfig)
- **toolConfig**: Tool specifications and tool choice for function calling

## Streaming responses

Streaming allows you to receive model responses incrementally as they are generated, providing a more interactive user experience. Both the Converse API and Invoke API support streaming.

### Streaming with ConverseStream

Use `ConverseStream` to receive responses as a stream of events:

```
import boto3

bedrock = boto3.client('bedrock-runtime', region_name='us-east-1')

response = bedrock.converse_stream(
    modelId='us.amazon.nova-2-lite-v1:0',
    messages=[
        {
            'role': 'user',
            'content': [{'text': 'Write a short story about AI.'}]
        }
    ]
)

for event in response['stream']:
    if 'contentBlockDelta' in event:
        delta = event['contentBlockDelta']['delta']
        if 'text' in delta:
            print(delta['text'], end='', flush=True)
```

## Streaming with InvokeModelWithResponseStream

Use `InvokeModelWithResponseStream` for streaming with the Invoke API:

```
import boto3
import json

bedrock = boto3.client('bedrock-runtime', region_name='us-east-1')

request_body = {
    'messages': [
        {
            'role': 'user',
            'content': [{'text': 'Explain quantum computing.'}]
        }
    ]
}

response = bedrock.invoke_model_with_response_stream(
    modelId='us.amazon.nova-2-lite-v1:0',
    body=json.dumps(request_body)
)

for event in response['body']:
    chunk = json.loads(event['chunk']['bytes'])
    if 'contentBlockDelta' in chunk:
        delta = chunk['contentBlockDelta']['delta']
        if 'text' in delta:
            print(delta['text'], end='', flush=True)
```

## Stream event types

Streaming responses include several event types:

- `messageStart`: Indicates the start of a message
- `contentBlockStart`: Indicates the start of a content block
- `contentBlockDelta`: Contains incremental text or data
- `contentBlockStop`: Indicates the end of a content block
- `messageStop`: Indicates the end of the message with stop reason
- `metadata`: Contains usage information (token counts)

# Using Amazon Nova embeddings

Amazon Nova Multimodal Embeddings is a multimodal embeddings model for agentic RAG and semantic search applications. It supports text, documents, images, video and audio through a single model, enabling cross-modal retrieval. Nova Multimodal Embeddings maps each of these content types into a unified semantic space, enabling you to conduct unimodal, cross-modal and multimodal vector operations.

When a piece of content is passed through Nova embeddings, the model converts that content into a universal numerical format, referred to as a vector. A vector is a set of numerical values that can be used for various search functionalities. Similar content is given a closer vector than less similar content.

Applications:

- Semantic Content Retrieval and Recommendation: Generate embeddings for your content, then use them to find similar items or provide personalized recommendations.
- Multimodal Search: Combine embeddings from different content types to enable cross-modal search capabilities.
- RAG: Generate embeddings from multimodal content such as documents with interleaved text and images to power your retrieval workflow for GenAI applications.

## Key features

These key features distinguish Nova Embeddings:

- Support for text, image, document image, video, and audio in a unified semantic space. Maximum context length is 8K tokens or 30s of video and 30s of audio.
- Synchronous and asynchronous APIs: The API supports both synchronous and asynchronous use.
- Large file segmentation: The async API makes it easy to work with large inputs by providing API built segmentation for long text, video, and audio, controlled by user-defined parameters. The model will generate a single embedding for each segment.
- Video with audio: Process video with audio simultaneously. Specify if you would like a single embedding representing both modalities or two separate embeddings.
- Embedding purpose: Optimize your embeddings depending on the intended downstream application (retrieval/RAG/Search, classification, clustering).

- Dimension sizes: 4 dimension sizes to trade-off embedding accuracy and vector storage cost: 3072; 1024; 384; 256.
- Input methods: Pass content to be embedded by specifying an S3 URI or inline as a base64 encoding.

## Generating embeddings

Complete the following to generate embeddings.

Synchronous embedding for text:

```
import boto3
import json

# Create the Bedrock Runtime client.
bedrock_runtime = boto3.client(
    service_name='bedrock-runtime',
    region_name='us-east-1',
)

# Define the request body.
request_body = {
    'taskType': 'SINGLE_EMBEDDING',
    'singleEmbeddingParams': {
        'embeddingPurpose': 'GENERIC_INDEX',
        'embeddingDimension': 3072,
        'text': {'truncationMode': 'END', 'value': 'Hello, World!'},
    },
}

try:
    # Invoke the Nova Embeddings model.
    response = bedrock_runtime.invoke_model(
        body=json.dumps(request_body, indent=2),
        modelId='amazon.nova-2-multimodal-embeddings-v1:0',
        accept='application/json',
        contentType='application/json',
    )

except Exception as e:
    # Add your own exception handling here.
    print(e)
```

```
# Print the request ID.  
print('Request ID:', response.get('ResponseMetadata').get('RequestId'))  
  
# Print the response body.  
response_body = json.loads(response.get('body').read())  
print(json.dumps(response_body, indent=2))
```

try:

```
response = bedrock_runtime.invoke_model(  
    #####body=json.dumps(request_body, indent=2),  
    #####modelId='amazon.nova-2-multimodal-embeddings-v1:0',  
    #####accept='application/json',  
    #####contentType='application/json',  
    #####)  
except Exception as e:  
    ####print(e)  
  
print('Request ID:', response.get('ResponseMetadata').get('RequestId'))  
response_body = json.loads(response.get('body').read())  
print(json.dumps(response_body, indent=2))
```

Asynchronous embedding for video:

```
import boto3  
  
bedrock_runtime = boto3.client(  
    #####service_name='bedrock-runtime',  
    #####region_name='us-east-1',  
)  
  
model_input = {  
    #####'taskType': 'SEGMENTED_EMBEDDING',  
    #####'segmentedEmbeddingParams': {  
        #####'embeddingPurpose': 'GENERIC_INDEX',  
        #####'embeddingDimension': 3072,  
        #####'video': {  
            #####'format': 'mp4',  
            #####'embeddingMode': 'AUDIO_VIDEO_COMBINED',  
            #####'source': {  
                #####'s3Location': {'uri': 's3://my-bucket/path/to/video.mp4'}  
            },  
        },  
    },  
}
```

```
#####'segmentationConfig': {  
#####'durationSeconds': 15### Segment into 15 second chunks  
#####},  
#####},  
#####},  
#####}  
}
```

try:

```
response = bedrock_runtime.start_async_invoke(  
#####modelId='amazon.nova-2-multimodal-embeddings-v1:0',  
#####modelInput=model_input,  
#####outputDataConfig={  
#####'s3OutputDataConfig': {  
#####'s3Uri': 's3://my-bucket'  
#####}  
#####},  
#####)  
except Exception as e:  
###print(e)  
  
print('Request ID:', response.get('ResponseMetadata').get('RequestId'))  
print('Invocation ARN:', response.get('invocationArn'))
```

## Batch inference

With batch inference, you can submit multiple requests and generate embeddings asynchronously. Batch inference helps you process many requests efficiently by sending a single request and generating responses in an Amazon S3 bucket.

- After defining model inputs in files you create, you upload the files to an S3 bucket.
- You then submit a batch inference request and specify the S3 bucket.
- After the job is complete, you can retrieve the output files from S3.
- You can use batch inference to improve the performance of model inference on large datasets.

## Format and upload your batch inference data

You must add your batch inference data to an S3 location that you'll choose or specify when submitting a model invocation job. The S3 location must the following items:

At least one JSONL file that defines the model inputs. A JSONL contains rows of JSON objects. Your JSONL file must end in the extension .jsonl and be in the following format:

```
{  
    "recordId": "record001",  
    "modelInput": {  
        "taskType": "SINGLE_EMBEDDING",  
        "singleEmbeddingParams": {  
            "embeddingPurpose": "GENERIC_INDEX",  
            "embeddingDimension": 3072,  
            "text": {  
                "source": {  
                    "s3Location": {  
                        "uri": "s3://batch-inference-input-bucket/text_001.txt",  
                        "bucketOwner": "111122223333"  
                    }  
                },  
                "truncationMode": "END"  
            }  
        }  
    }  
}
```

## Supported input file types

- Image Formats: PNG, JPEG, WEBP, GIF
- Audio Formats: MP3, WAV, OGG
- Video Formats: MP4, MOV, MKV, WEBM, FLV, MPEG, MPG, WMV, 3GP

## On-demand inference

On-demand inference provides serverless access to Amazon Nova models without requiring provisioned capacity. This mode automatically scales to handle your workload and charges based on usage.

## Benefits

On-demand inference offers several advantages:

- **No capacity planning:** Automatically scales to meet demand

- **Pay per use:** Charged only for tokens processed
- **Instant availability:** No provisioning or warm-up time required
- **Cost effective:** Ideal for variable or unpredictable workloads

## Using on-demand inference

On-demand inference is the default mode for Amazon Nova models. Simply specify the model ID when making API calls:

```
import boto3

bedrock = boto3.client('bedrock-runtime', region_name='us-east-1')

response = bedrock.converse(
    modelId='us.amazon.nova-2-lite-v1:0',
    messages=[
        {
            'role': 'user',
            'content': [{'text': 'Hello, Nova!'}]
        }
    ]
)

# Print the response text
content_list = response["output"]["message"]["content"]
text = next((item["text"] for item in content_list if "text" in item), None)
if text is not None:
    print(text)
```

## Pricing

On-demand inference is billed based on the number of input and output tokens processed. For current pricing details, see [Amazon Bedrock pricing](#).

## Quotas and limits

On-demand inference has default quotas that vary by model and region. To request quota increases, use the [Service Quotas console](#).

# Prompt engineering guide

Prompt engineering is the practice of designing instructions to effectively communicate with large language models (LLMs) like Amazon Nova. Prompt engineering allows you to control model behavior, improve output quality and build reliable AI-powered applications.

Effective prompt engineering involves crafting inputs that guide models to produce desired outputs. A well-crafted prompt provides clear instructions, relevant context and properly formatted input data.

## What is prompt engineering

Prompt engineering refers to the practice of optimizing textual input to a large language model (LLM) to improve output and receive the responses you want. Prompting helps an LLM perform a wide variety of tasks, including classification, question answering, code generation, creative writing, and more. The quality of prompts that you provide to a LLM can impact the quality of the model's responses. This section provides you the necessary information to get started with prompt engineering. It also covers tools to help you find the best possible prompt format for your use case when using a LLM on Amazon Bedrock

The effectiveness of prompts is contingent upon the quality of the information provided and the craftsmanship of the prompt itself. Prompts may encompass instructions, questions, contextual details, inputs, and examples to effectively guide the model and enhance the quality of the results. This document outlines strategies and tactics for optimizing the performance of Amazon Nova 2 Sonic family of models. The methods presented herein may be employed in various combinations to amplify their effectiveness. We encourage you to engage in experimentation to identify the approaches most suitable for their specific needs.

## Getting started with prompt engineering

Before you start prompt engineering, we recommend that you have the following elements in place, so you can iteratively develop the most optimal prompt for your use case:

### Define your use case

Define your use case across four dimensions:

- **Task** – Define what you want the model to accomplish. This determines the right prompting technique.

- **Role** – Define what role the model should assume to accomplish the task. Amazon Nova models support three roles (System, User, or Assistant).
- **Response Style** – Define the response structure or style that the model should follow based on the audience, such as JSON, markdown, or conversational.
- **Instructions** – Define the set of instructions that the model should follow to meet success criteria.

## Establish success criteria

Define success criteria or evaluation metrics. You can provide a list of criteria or provide specific evaluation metrics, such as length, BLEU score, ROUGE, format, factuality and faithfulness.

## Draft a prompt

Create a starting prompt incorporating your task, role, response style and instructions. Iterate based on results.

The effectiveness of prompts depends on the quality of information you provide.

## General best practices

The following best practices mainly apply to the Amazon Nova text models, but you can apply them to other models, in addition to modality-specific best practices.

For more information on how to prompt multimodal inputs, refer to [Prompting multimodal inputs](#).  
For information on how to prompt speech inputs, refer to [Voice conversation prompts](#).

## Understanding the roles

Amazon Nova models allow you to structure prompts through the use of three distinct roles: system, user, and assistant.

- **System (optional)** – Although not mandatory, it establishes the overall behavioral parameters of the assistant. It can also be utilized to provide additional instructions and guidelines that the user wishes the model to adhere to throughout the conversation.
- **User** – Can optionally convey the context, tasks, instructions, and the desired outcome along with the user query.
- **Assistant** – Aids in guiding the model towards the intended response.

## Topics

- [Create precise prompts](#)
- [Bring focus to sections of the prompt](#)
- [Using the system role](#)
- [Provide examples \(few-shot prompting\)](#)
- [Tool calling systems](#)
- [Advanced prompting techniques](#)

## Create precise prompts

Crafting specific user queries is crucial in prompt engineering. Well-crafted queries direct Amazon Nova text understanding models to generate accurate and pertinent responses. To craft such queries, it is essential to start by providing contextual information in the query. The context provided helps situate the model better by aligning to clear target-user scenarios, thereby enhancing the relevance and coherence of the output.

### Providing contextual, specific information

Models work best if the instructions are clear, well structured, short, and specific. This prompt template guidance applies to both the user prompt and the system prompt.

#### Prompt template

```
# Use this to clearly define the task and job needed by the model
task_summary = f"""
## Task Summary:
{{Task summary}}
"""

# Use this to provide contextual information related to the task
context_information = f"""
## Context Information:
- {{Context and content information 1}}
- {{Context and content information 2}}
...
"""

# Use this to provide any model instructions that you want model to adhere to
model_instructions = f""""
```

```
## Model Instructions:  
- {{ Other Model Instructions }}  
...  
....  
  
# Use this to provide response style and formatting guidance  
response_style = f"""  
## Response style and format requirements:  
- {{Style and format requirement 1}}  
- {{Style and format requirement 2}}  
...  
....  
  
# Concatenate to final prompt  
final_prompt = f"""{task_summary}  
{context_information}  
{model_instructions}  
{response_style}"""
```

## Example: Specific and contextual information

Here is an example of a prompt that is rewritten with specific and contextual information. Adding more contextual information helps the model ground towards the desired behavior.

### Unclear prompt

User:

Write me a meeting invite to the project team

Assistant:

Certainly! Here's a sample meeting invite for your project team:

---

\*\*Subject:\*\* Project Team Meeting - [Project Name]

\*\*Date:\*\* [Insert Date]

\*\*Time:\*\* [Insert Time]

\*\*Duration:\*\* [Insert Duration]

\*\*Location:\*\* [Insert Location] / [Zoom/Teams Link if Online]

**\*\*Agenda:\*\*****1. \*\*Welcome and Introductions\*\***

- Quick round of introductions for new team members

**2. \*\*Project Overview\*\***

- Objectives and goals
- Current status and milestones achieved

**3. \*\*Task Allocation and Deadlines\*\***

- Detailed breakdown of tasks
- Assignment of responsibilities
- Discussion of deadlines

**4. \*\*Risk Management\*\***

- Identification of potential risks
- Mitigation strategies

**5. \*\*Resource Allocation\*\***

- Required tools and materials
- Budget review

**6. \*\*Q&A Session\*\***

- Open floor for questions and clarifications

**7. \*\*Next Steps and Closing\*\***

- Summary of action items
- Schedule for the next meeting

**\*\*Additional Notes:\*\***

- Please come prepared with any relevant documents or updates regarding your assigned tasks.
- If you anticipate any issues or require additional support, please let us know in advance so we can address them during the meeting.

**\*\*Contact Information:\*\***

For any questions or concerns before the meeting, please feel free to reach out to [Your Name] at [Your Email] or [Your Phone Number].

We look forward to a productive discussion and to kicking off our project with enthusiasm!

Best regards,

[Your Full Name]

[Your Position]

[Your Contact Information]

---

Feel free to customize this template to better fit your specific project and team needs!

Clear prompt

User:

Write me a meeting invite to the project team. **The meeting is about project planning, and should be held on October 16th, from 10 AM to 11 AM in Conference Room B. Please include an agenda that covers our progress on the project thus far, as well as any upcoming milestones and deadlines.**

Assistant:

**\*\*Meeting Invitation: Project Planning Session\*\***

**\*\*Date:\*\*** October 16th

**\*\*Time:\*\*** 10:00 AM - 11:00 AM

**\*\*Location:\*\*** Conference Room B

**\*\*Agenda:\*\***

1. **\*\*Welcome and Introduction (10:00 AM - 10:05 AM)\*\***

- Brief overview by the meeting facilitator.

2. **\*\*Project Progress Update (10:05 AM - 10:30 AM)\*\***

- Review of completed tasks and achievements to date.

- Detailed discussion on current project status and any challenges encountered.

3. **\*\*Upcoming Milestones and Deadlines (10:30 AM - 10:50 AM)\*\***

- Overview of the next phase of the project.

- Identification of key milestones and deadlines.

4. **\*\*Action Items and Responsibilities (10:50 AM - 11:00 AM)\*\***

- Assignment of tasks for the upcoming period.

- Confirmation of individual responsibilities and follow-up items.

5. \*\*Q&A Session (11:00 AM)\*\*

- Open floor for any questions or concerns from team members.

Please ensure your attendance to discuss and finalize our project planning for the next phase. Your insights and contributions are highly valued.

Best regards,

[Your Name]

[Your Position]

[Your Contact Information]

- Be clear: When prompting the model, try to keep instructions as clear as possible in a way a human would understand it versus a machine (for instance, a JSON-style list).
- Be specific: Give the model specific instructions about what the task is, what the expected output is, and any additional context to help the model respond the way you want.

## Specify the output format and style

If you want Amazon Nova 2 to generate responses that are tailored to your specific needs, you can specify the format and style of the output. For example, if you want to generate responses only in JSON or constrain the response length to a maximum number of words, bullet points, or paragraphs. This enables Amazon Nova 2 to better understand your requirements and produce output that is more relevant and easier to parse.

Use capitalization and strong wordings such as DO, DO NOT, MUST to emphasize the rules you want the model to strictly obey. For example, if you want only one JSON object in the response, you can instruct the model by saying "You MUST answer in JSON format only. DO NOT use any other format while answering the question."

Here are some example prompts that are rewritten with specific and contextual information. Adding more contextual information helps the model ground towards the desired behavior.

### Example 1: Provide extra details

Adding extra details on the response formatting can help the model accomplish the desired result. Some of the other commonly used response format include CSV, Markdown, YAML, and XML

## Prompt without style constraints

User:

Outline the different types of rooms available in a typical 4-star hotel, including their features.

Assistant:

In a typical 4-star hotel, you'll find a range of room types, each designed to cater to different guest needs and preferences. Here's an outline of the different types of rooms available along with their features:

### ### 1. \*\*Standard Room\*\*

- \*\*Size:\*\* Usually around 250-350 square feet.
- \*\*Features:\*\*
  - Queen or double bed.
  - Complimentary Wi-Fi.
  - Flat-screen TV with cable or satellite channels.
  - Mini-fridge.
  - Coffee maker with complimentary coffee and tea.
  - Desk and chair.
  - En-suite bathroom with shower, toilet, and toiletries.
  - Air conditioning/heating.

### ### 2. \*\*Deluxe Room\*\*

- \*\*Size:\*\* Typically larger than standard rooms, around 350-450 square feet.
- \*\*Features:\*\*
  - King bed or two queen beds.
  - Larger bathroom with a bathtub.
  - Additional seating area.
  - Enhanced amenities like bathrobes and slippers.
  - Upgraded toiletries and sometimes a welcome gift.
  - Access to a lounge or executive floor perks.

### ### 3. \*\*Suite\*\*

- \*\*Size:\*\* Varies significantly, often between 500-1,200 square feet.
- \*\*Features:\*\*
  - Separate living area with a sofa bed or additional seating.
  - Kitchenette or full kitchen in some cases.
  - Larger, more luxurious bathroom with a bathtub and separate shower.
  - Multiple TVs.
  - Access to executive lounge.

- High-end toiletries and often complimentary breakfast.
- Often includes concierge services and in-room dining options.

#### ### 4. \*\*Family Room\*\*

- \*\*Size:\*\* Larger, usually around 400-500 square feet.
- \*\*Features:\*\*
- One king bed and one sofa bed or two queen beds.
- Designed for families with children.
- Extra space for luggage.
- Child-friendly amenities like a small kitchenette or high chair.
- Family bathroom with amenities suitable for kids.

#### ### 5. \*\*Executive Room\*\*

- \*\*Size:\*\* Typically around 350-450 square feet.
- \*\*Features:\*\*
- Business-oriented with a larger workspace.
- Executive desk and chair.
- Access to executive lounge with complimentary snacks and refreshments.
- King bed or two queen beds.
- Enhanced amenities like high-speed internet and sometimes a Nespresso machine.

#### ### 6. \*\*Junior Suite\*\*

- \*\*Size:\*\* Usually around 450-600 square feet.
- \*\*Features:\*\*
- Separate bedroom and living area.
- King bed or two queen beds.
- Kitchenette or mini-fridge.
- Luxurious bathroom with a bathtub.
- Lounge access.
- High-end toiletries.

#### ### 7. \*\*Premium Room\*\*

- \*\*Size:\*\* Larger than standard rooms, around 400-500 square feet.
- \*\*Features:\*\*
- King bed or two queen beds.
- Larger bathroom with a bathtub.
- Enhanced amenities like a welcome drink and premium toiletries.
- Access to premium services and sometimes a lounge.

#### ### 8. \*\*Connecting Rooms\*\*

- \*\*Size:\*\* Two rooms that connect via an internal door.
- \*\*Features:\*\*
- Suitable for families or groups.
- Often includes one standard room and one suite.

- Shared bathroom or en-suite bathrooms in each room.
- Extra space for luggage and guests.

### ### 9. \*\*Accessible Rooms\*\*

- \*\*Size:\*\* Varies, typically similar to standard rooms.
- \*\*Features:\*\*
- Designed for guests with disabilities.
- Wider doorways and bathroom space.
- Adapted fixtures and fittings.
- Additional support services.

### ### 10. \*\*Business Room\*\*

- \*\*Size:\*\* Usually around 300-350 square feet.
- \*\*Features:\*\*
- Focused on business travelers.
- High-speed internet and large workspace.
- Compact yet fully equipped with essentials.
- Often includes a king bed.

Each room type in a 4-star hotel is designed to provide comfort, convenience, and luxury tailored to different guest needs, ensuring a pleasant stay for all types of travelers.

## Prompt with style constraints

User:

Outline the different types of rooms available in a typical 4-star hotel, including their features. **All responses MUST be in YAML format.**

Assistant:

```
```yaml
hotel_rooms:
- room_type: Standard Room
features:
- description: A comfortable room with essential amenities.
amenities:
- bed_type: Double bed or two single beds
- bathroom: Private bathroom with shower
- wifi: Complimentary high-speed internet
- TV: Flat-screen TV with cable channels
- mini_fridge: Available
```

```
- tea_and_coffee_making_facilities: Yes

- room_type: Deluxe Room
features:
- description: An upgraded room with more space and enhanced amenities.
amenities:
- bed_type: King-size bed or two single beds
- bathroom: Private bathroom with both shower and bathtub
- wifi: Complimentary high-speed internet
- TV: Larger flat-screen TV with cable channels
- mini_fridge: Available
- tea_and_coffee_making_facilities: Yes
- free_breakfast: Included

- room_type: Suite
features:
- description: A luxurious room with separate living area and premium amenities.
amenities:
- bed_type: King-size bed or two single beds
- bathroom: Private bathroom with both shower and bathtub
- separate_living_area: Yes
- wifi: Complimentary high-speed internet
- TV: Large flat-screen TV with cable channels
- mini_fridge: Available
- tea_and_coffee_making_facilities: Yes
- free_breakfast: Included
- concierge_service: Available

- room_type: Family Room
features:
- description: A spacious room designed for families with additional bedding options.
amenities:
- bed_type: Queen bed and rollaway bed or two single beds and rollaway bed
- bathroom: Private bathroom with shower
- wifi: Complimentary high-speed internet
- TV: Flat-screen TV with cable channels
- mini_fridge: Available
- tea_and_coffee_making_facilities: Yes

- room_type: Business Room
features:
- description: A room tailored for business travelers with enhanced workspace.
amenities:
```

- bed\_type: Double bed or two single beds
  - bathroom: Private bathroom with shower
  - wifi: Complimentary high-speed internet
  - TV: Flat-screen TV with cable channels
  - workspace: Desk and chair
  - mini\_fridge: Available
  - tea\_and\_coffee\_making\_facilities: Yes
  - business\_center\_access: Available
- ...

## Example 2: Provide style constraints

In this case, the prompt with style constraints provides comprehensive information on the constraints to follow and the desired response format. As observed in the response, the optimized prompt aligns more closely with the requested brevity and bulleted list structure.

### Prompt without style constraints

User:

Explain the process of checking in a guest at a luxury resort.

Assistant:

Checking in a guest at a luxury resort involves a meticulous and personalized process designed to ensure a seamless and memorable start to their stay. Here's a detailed overview of the steps involved:

#### ### Pre-Arrival Preparation

##### 1. \*\*Pre-Arrival Communication:\*\*

- \*\*Confirmation:\*\* Verify guest reservations and confirm arrival details.
- \*\*Special Requests:\*\* Collect any special requests (dietary restrictions, room preferences, etc.) and ensure they are noted in the guest profile.
- \*\*Documentation:\*\* Ensure all necessary documentation is ready, such as welcome letters, room keys, and any special amenities.

##### 2. \*\*Room Preparation:\*\*

- \*\*Cleaning:\*\* Conduct a thorough cleaning and preparation of the guest room.
- \*\*Amenities:\*\* Stock the room with high-end toiletries, fresh linens, and any requested amenities.

- **Personal Touches:** Add personalized touches such as a welcome note, fresh flowers, or locally sourced treats.

### ### Arrival and Initial Interaction

#### 3. **Guest Arrival:**

- **Greeting:** Have staff ready to greet guests as they arrive. This may include a warm welcome at the entrance or a meet-and-greet in the lobby.
- **Transport:** Provide luxury transport options like a chauffeured car service to escort guests to the check-in desk.

#### 4. **Check-In Desk:**

- **Professionalism:** Ensure check-in staff are well-trained, professional, and exude hospitality.
- **Technology:** Utilize sophisticated check-in technology (like tablet kiosks) to streamline the process.
- **Personalized Welcome:** Greet guests with a warm welcome, acknowledge their special requests, and present them with a personalized welcome kit.

### ### Documentation and Formalities

#### 5. **Document Verification:**

- **Identification:** Verify identification and payment methods.
- **Membership Cards:** If applicable, verify any loyalty or membership cards for additional benefits.

#### 6. **Registration:**

- **Form Filling:** Assist guests with any required paperwork.
- **Special Benefits:** Clearly explain any special resort benefits, upgrades, or exclusive access to amenities that the guest is entitled to.

### ### Room Assignment

#### 7. **Room Presentation:**

- **Walk-Through:** Escort guests to their room and conduct a walk-through to ensure everything is in order and to address any immediate concerns.
- **Upgrades:** Present any room upgrades and explain the features and services available in the upgraded room.

#### 8. **Key Delivery:**

- **Room Key:** Provide the room key, often a high-end key card or a luxury key fob.
- **Access Information:** Explain how to use the key card, including access to other amenities like the spa, pool, and gym.

### ### Additional Services

#### 9. \*\*Welcome Amenities:\*\*

- \*\*Amenity Kits:\*\* Deliver welcome amenity kits (like a bottle of champagne, local delicacies, etc.).
- \*\*Itinerary:\*\* Provide a detailed resort map and itinerary, highlighting all available services and activities.

#### 10. \*\*Personal Concierge:\*\*

- \*\*Concierge Services:\*\* Introduce the guest to the concierge service and explain how to request any special services, such as dining reservations, transportation, or local excursions.
- \*\*Contact Information:\*\* Provide contact information for key staff members.

### ### Final Touches

#### 11. \*\*Feedback:\*\*

- \*\*Satisfaction Survey:\*\* At the end of the check-in process, ask for feedback to ensure the guest's expectations were met and to gather any immediate suggestions.

#### 12. \*\*Follow-Up:\*\*

- \*\*Immediate Issues:\*\* Ensure any immediate issues or requests are addressed promptly.
- \*\*Comfort Check:\*\* Make follow-up calls or send messages to ensure the guest is settling in comfortably.

By following these steps, a luxury resort aims to provide an exceptional check-in experience that sets the tone for an unforgettable stay.

## Prompt with style constraints

User:

Explain the process of checking in a guest at a luxury resort. **Provide all answers in a numbered list format, with each point containing no more than 15 words, and no more than 6 bullet points.**

Assistant:

1. Greet guest warmly with a smile and offer assistance.
2. Verify reservation details and check guest's identification.
3. Confirm room assignment and amenities provided.

4. Explain resort services and amenities available.
5. Present room key and offer directions to guest's suite.
6. Wish guest a pleasant stay and be available for queries.

### Example 3: Limit the response format

Here, the prompt with the response format forces the model to respond only with the category number and limit it to be between 1-10, instead of being highly verbose.

Prompt without response format

User:

In this task, you are given a text which is the body of a document. Your job is to classify the topic of the document into these categories: 1)Company, 2)Educational Institution, 3)Artist, 4)Athlete, 5)Office Holder, 6)Mean of transportation, 7)Building, 8)Natural place, 9)Village, 10)Animal.

Text: Johangsan is a mountain of South Korea. It has an elevation of 954 metres.

Assistant:

To classify the topic of the given text, let's analyze the content step-by-step:

1. The text mentions "Johangsan," which appears to be a specific name.
2. The text states that "Johangsan is a mountain," which provides a clear indication of its nature.
3. It is further specified that it is located in "South Korea" and has an elevation of "954 metres."

Based on the information provided, the text describes a geographical feature, specifically a mountain. From the given categories, the most appropriate classification for this document is:

8) Natural place

The text does not mention any company, educational institution, artist, athlete, office holder, means of transportation, building, village, or animal. Therefore, the classification as a "Natural place" is the most fitting choice.

Prompt with response format

User:

In this task, you are given a text which is the body of a document. Your job is to classify the topic of the document into these categories: 1)Company, 2)Educational Institution, 3)Artist, 4)Athlete, 5)Office Holder, 6)Mean of transportation, 7)Building, 8)Natural place, 9)Village, 10)Animal. **Your output should be the category number. DO NOT generate anything apart from numbers 1-10.**

Text: Johangsan is a mountain of South Korea. It has an elevation of 954 metres.

Assistant:

8

## Provide clear prompt sections

Beyond the initial context, a structured query format plays a significant role in optimizing interactions with the language model. Separating distinct parts of the input with delimiters ensures clarity and enhances Amazon Nova 2 models' ability to understand the query's structure. Another effective way is to use numbered lists or bullet points, which can help to break down the task into manageable steps and provide a clear outline of the actions that need to be taken. This structured approach aids in reducing ambiguity and improving the accuracy of the responses generated.

### Example: Create a sectioned prompt

Notice how in the Sectioned Prompt column, the prompt is clearer and has delineated sections for different entities like Tasks, Context, Requirements, and the rest when compared to the instructions in the Unsectioned Prompt column.

#### Unsectioned Prompt

User:

Write an article about climate change for high school students explaining basic concepts, current impacts, and individual actions to mitigate it. Make it informative and engaging with simple language, including statistics and real-world examples.

Assistant:

### Understanding Climate Change: What It Is, What It's Doing, and What You Can Do About It

#### #### Introduction

Climate change might sound like a big, complicated topic, but it's really about the way our planet's weather is changing over time. This change is mostly happening because humans are doing things that are making our Earth warmer. Let's break it down so you can understand what's going on and what you can do to help.

#### #### What Is Climate Change?

Climate change refers to long-term shifts in temperatures and weather patterns. These shifts may be natural, but since the 1800s, human activities have been the main driver of rapid climate change.

#### \*\*Key Concept: Greenhouse Gases\*\*

To understand this, we need to talk about greenhouse gases. These gases, like carbon dioxide (CO<sub>2</sub>), methane (CH<sub>4</sub>), and nitrous oxide (N<sub>2</sub>O), trap heat in our atmosphere. Without these gases, Earth would be too cold for most life forms. But when we burn fossil fuels (like coal, oil, and natural gas) and cut down forests, we release too much CO<sub>2</sub> and other greenhouse gases, trapping more heat and making the planet warmer.

#### #### Current Impacts of Climate Change

Let's look at some real-world examples to see how climate change is affecting our world today.

#### \*\*1. Rising Temperatures\*\*

Since the late 19th century, Earth's average surface temperature has risen about 2 degrees Fahrenheit (1 degree Celsius). Scientists predict that if we don't change our ways, it could rise by 3.6 to 4.3 degrees Fahrenheit (2 to 2.4 degrees Celsius) by the end of this century.

#### \*\*2. More Extreme Weather\*\*

Climate change is making extreme weather events more frequent and severe. For example:

- \*\*Hurricanes:\*\* In the U.S., hurricane seasons are getting longer and more destructive. In 2020, there were 30 named storms, the most ever recorded.
- \*\*Droughts:\*\* Parts of California have experienced some of the worst droughts in history, affecting agriculture and water supplies.

#### \*\*3. Melting Ice Caps and Rising Sea Levels\*\*

Glaciers and ice sheets are melting at alarming rates. This not only affects wildlife but also leads to rising sea levels. Since 1901, global sea levels have risen by about 8 inches (20 centimeters). By 2100, they could rise by another 1 to 4 feet (0.3 to 1.2 meters).

#### \*\*4. Ocean Acidification\*\*

As CO<sub>2</sub> levels rise, more of it is absorbed by the oceans, making them more acidic. This is particularly harmful to marine life, including coral reefs. Coral reefs are crucial for marine biodiversity but are now dying at an alarming rate.

#### #### Individual Actions to Mitigate Climate Change

You might think that since climate change is a huge problem, one person can't make a difference. But even small actions can add up to make a big impact.

##### \*\*1. Reduce, Reuse, Recycle\*\*

Try to minimize waste by:

- \*\*Reducing:\*\* Buy fewer items and choose products with less packaging.
- \*\*Reusing:\*\* Use reusable bags, bottles, and containers.
- \*\*Recycling:\*\* Make sure to recycle paper, plastic, and glass properly.

##### \*\*2. Conserve Energy\*\*

Small changes can lead to big savings:

- Turn off lights when you leave a room.
- Unplug devices that you're not using.
- Use energy-efficient bulbs and appliances.

##### \*\*3. Use Less Meat\*\*

Livestock farming is a major contributor to greenhouse gas emissions. By eating less meat, especially beef, you can reduce your carbon footprint.

##### \*\*4. Support Renewable Energy\*\*

If possible, support or switch to renewable energy sources like solar or wind power. Even advocating for these options in your community can make a difference.

##### \*\*5. Educate Yourself and Others\*\*

Knowledge is power. The more you know about climate change, the more you can help others understand it. Share facts and encourage your friends and family to take action too.

#### #### Conclusion

Climate change is a pressing issue that affects everyone on the planet. By understanding its basics, recognizing its current impacts, and taking individual actions, you can play a crucial role in mitigating its effects. Remember, every small step counts, and together, we can make a big difference for our planet.

Let's start today and build a sustainable future!

## Sectioned Prompt

User:

\*\*Task:\*\*

Write an article

\*\*Context Section:\*\*

Topic: Climate Change

Target Audience: High School Students

Purpose: Educational Article

\*\*Content Requirements:\*\*

- Explanation of basic concepts related to climate change.
- Examples of current impacts on ecosystems and human populations.
- Suggestions for individual actions to mitigate climate change.

\*\*Tone and Style:\*\*

- Informative and engaging
- Use simple language
- Include statistics and real-world examples

Assistant:

```
# Understanding Climate Change: A Guide for High School Students
```

```
## Introduction
```

Climate change is one of the most pressing issues of our time. It affects our planet, our ecosystems, and our daily lives. Understanding what climate change is and how it impacts us can help us take action to make a difference. This article will explain the basic concepts of climate change, provide examples of its current impacts, and suggest actions you can take to help mitigate it.

```
## What is Climate Change?
```

\*\*Climate change\*\* refers to long-term changes in global temperatures and weather patterns. These changes are primarily caused by human activities, especially the burning of fossil fuels like coal, oil, and natural gas. When we burn these fuels, we release large amounts of greenhouse gases, such as carbon dioxide (CO<sub>2</sub>) and methane (CH<sub>4</sub>), into the atmosphere. These gases trap heat from the sun, causing the Earth's temperature to rise. This is often referred to as the \*\*greenhouse effect\*\*.

### ### Basic Concepts:

1. \*\*Greenhouse Gases\*\*: These gases include carbon dioxide, methane, nitrous oxide, and fluorinated gases. They trap heat in the Earth's atmosphere.
2. \*\*Global Warming\*\*: This is the increase in Earth's average surface temperature due to rising levels of greenhouse gases.
3. \*\*Climate Change\*\*: Refers to the broader set of changes in the Earth's climate system, including global warming.

## ## Current Impacts of Climate Change

### ### On Ecosystems:

1. \*\*Melting Ice Caps and Glaciers\*\*: The polar ice caps and glaciers are melting at an alarming rate. For example, the Arctic sea ice is shrinking, which affects polar bears and other Arctic animals.
2. \*\*Rising Sea Levels\*\*: As ice melts, sea levels rise. This can lead to the loss of coastal habitats and increased flooding in coastal areas.
3. \*\*Changing Weather Patterns\*\*: Climate change can lead to more frequent and severe weather events such as hurricanes, droughts, and heatwaves. For example, the 2020 Australian bushfires were more intense and lasted longer than usual due to climate change.

### ### On Human Populations:

1. \*\*Health Issues\*\*: Rising temperatures can lead to more heat-related illnesses and the spread of diseases like malaria and dengue fever.
2. \*\*Food Security\*\*: Changes in climate can affect crop yields, leading to food shortages. For example, droughts in Sub-Saharan Africa have caused significant food insecurity.
3. \*\*Displacement\*\*: Rising sea levels and extreme weather events can force people to leave their homes, becoming climate refugees. For instance, communities in Bangladesh are being displaced by rising sea levels.

## ## Actions You Can Take

Even though climate change is a massive issue, there are many actions you can take to help mitigate its effects. Here are some simple steps you can start with today:

1. **\*\*Reduce, Reuse, Recycle\*\*:** By reducing waste, reusing items, and recycling, you can decrease the amount of trash that ends up in landfills and reduce greenhouse gas emissions.
2. **\*\*Save Energy\*\*:** Turn off lights when you don't need them, use energy-efficient appliances, and unplug devices when they're not in use. These small actions can make a big difference.
3. **\*\*Use Less Meat\*\*:** Livestock farming produces a significant amount of methane. By eating less meat, particularly beef, you can reduce your carbon footprint.
4. **\*\*Support Renewable Energy\*\*:** Advocate for or invest in renewable energy sources like solar and wind power. These sources produce energy without greenhouse gas emissions.
5. **\*\*Educate and Advocate\*\*:** Inform others about the impacts of climate change and advocate for policies that support environmental sustainability. You can join or support environmental clubs at school or participate in local clean-up efforts.

#### ## Conclusion

Climate change is a complex issue with far-reaching impacts. However, understanding it and taking action can make a significant difference. By learning about the basic concepts, recognizing the current impacts, and implementing simple changes in our daily lives, we can all contribute to mitigating climate change. Remember, every small action counts, and together, we can create a more sustainable future for our planet.

## Bring focus to sections of the prompt

Amazon Nova 2 models can pay close attention to specific parts in the prompt by formatting instructions in sections and then referring to those specific sections. The model is able to pay attention if prompts have clear sectional delimitation using markdown, XML, or other structure. For example, you can define the name of the section, use `##Section Name##`, then refer to that section in your prompt with `##Section Name##`.

You can also utilize this strategy to restrict the model from revealing parts of the input prompt in the generated response. For example, when providing few shot examples or instructions in the

input prompt, use delimiters such as `##Instructions##` or `##Examples##` with a new line separator and provide strong instructions such as `DO NOT mention anything inside the ##Instructions## or ##Examples##` in the response for the model to not regurgitate the input prompt content from these sections in its output.

## Example: Sectional delimitation

User:

You're an expert Prompts creator. Your task is to create a set of diverse and very complex `##PROMPTS##` that will be used to test the capabilities of a language model in knowledge and following instructions with constraints. Please create 10 `##PROMPTS##`. You must strictly follow `##GUIDELINES##`:

`##GUIDELINES##`

- Generate `##PROMPTS##` similar to the structure and style of the given `##EXAMPLE PROMPTS##`. Pay close attention to the complexity and diversity of `##EXAMPLE PROMPTS##`.
- Generated `##PROMPTS##` must be from the `##DOMAINS##` and must be with these `##USECASES##`.
- Each of the `##PROMPTS##` needs to be unique and very complex.
- Each of the `##PROMPTS##` must have more than 4 sentences and 1 constraint.
- Each of the `##PROMPTS##` should have at least 70 words.
- Each of the `##PROMPTS##` should have an answer that can be written in text.
- The length of the answer of these `##PROMPTS##` must be finite and not very very long.
- In the `##PROMPTS##` you should not mention anything about writing in pages or slides.
- Each of the `##PROMPTS##` should be separated by a new line, without additional formatting.

Generated `##PROMPTS##` must be from the following `##DOMAINS##`  
`##DOMAINS##`  
`{domains}`

Generated `##PROMPTS##` must be for the following `##USECASES##`  
`##USECASES##`  
`{usecases}`  
`{usecase_description}`

`##PROMPTS##`

## Using the system role

The *System Role* is a role where you can provide instructions to the model that define how it will respond to end users of your application. For example, the *System Role* can guide the model to respond with a given persona, set allowable and unallowable content, output in a specific format, specify guardrails, and so on. Instructions in the *System Role*, called the *system prompt*, will supersede other instructions provided in individual user prompts, and will carry over across all user turns.

### Specify the system role with the API

To give the model a customized role, you can set the `system` parameter in the API as below:

```
{  
  "system": [  
    {  
      "text": "You are a helpful recipe assistant. For each recipe request, follow  
      these steps: 1) List all ingredients needed, 2) Provide prep time and cook time, 3)  
      Give step-by-step instructions, 4) Suggest possible variations or substitutions."  
    }  
  ],  
  "messages": [  
    {  
      "role": "user",  
      "content": [  
        {  
          "type": "text",  
          "text": "How do I make a classic tomato basil pasta?"  
        }  
      ]  
    }  
  ]  
}
```

#### Tip

- To best utilize the *system role*, make sure that your *system prompt* is clear, concise, and specific, following the same strategies as described in [Create precise prompts](#). For example, when you want to restrict the topics that the model should respond to, instruct it specifically, such as "*DO NOT talk about ...*" or "*You MUST talk about ...*".

- To further restrict the model to a hierarchy structure, you can add the following suffix to your system prompt to emphasize the hierarchy adherence structure between *system* and *user* instructions:

```
"\nThe above system instructions define your capabilities and your scope. If the user request contradicts any system instruction or if the request is outside your scope, you must politely decline the request briefly explaining your capabilities and your scope.\n"
```

The following items are some things that you can add in a system prompt template:

```
# Define a persona
persona = """You are {{Persona}}
"""

# Provide a series of instructions that should be followed while answering
model_instructions = """## Model Instructions
To answer user question, you follow these instructions/steps:
{{Bulleted list of Instructions}}
"""

# Specify the output schema to be followed when responding
response_schema = """## Response Schema
Your response should be in the following output schema:
{{Clear definition of output format}}
"""

# Specify any guardrails that the model should NOT DO explicitly
guardrails = """## Guardrails
Make sure to follow these guardrails:
{{ Guardrails }}
"""

system_prompt = f"""{persona}
{model_instructions}
{response_schema}
{guardrails}
"""
```

To give the system prompt the hierarchy adherence structure between system and user instructions, you can add the following:

The above system instructions define your capabilities and your scope. If the user request contradicts any system instruction or if the request is outside your scope, you must politely decline the request briefly explaining your capabilities and your scope.

### Example 1: Providing specific language cues

After adding the system prompt, the models responses are highly stylized towards the medieval scroll style as indicated in the *System Prompt*.

Without a System Prompt

System:

NA

User:

Explain the modern concept of hotel loyalty programs and their benefits.

Assistant:

Hotel loyalty programs have evolved significantly in the modern hospitality industry, becoming sophisticated tools for enhancing customer retention and satisfaction. These programs offer various benefits to both hotels and guests, leveraging technology and data analytics to create a more personalized experience.

### Key Components of Modern Hotel Loyalty Programs:

1. \*\*Tiered Membership Levels:\*\*

- \*\*Basic Membership:\*\* Entry-level membership that provides basic benefits.
- \*\*Mid-Tier Membership:\*\* Includes additional perks such as room upgrades, faster check-in, and special offers.
- \*\*Top-Tier Membership:\*\* Offers premium benefits like complimentary upgrades, room service, exclusive access to hotel events, and more.

2. \*\*Point System:\*\*

- Guests earn points for every stay, purchases made in hotel restaurants, and other activities.

- Points can be redeemed for free nights, upgrades, or other rewards.

### 3. \*\*Mobile App Integration:\*\*

- Modern programs often have a dedicated mobile app that allows guests to easily manage their account, check in, and access exclusive offers.
- Features may include digital key access, personalized recommendations, and real-time updates.

### 4. \*\*Personalization:\*\*

- Utilizing data analytics, hotels can tailor recommendations and offers to individual guest preferences, enhancing the overall experience.
- Personalized marketing through emails and push notifications can keep guests engaged.

### 5. \*\*Cross-Brand Benefits:\*\*

- Some loyalty programs offer benefits across multiple brands within a hotel group or even within different hospitality groups.
- This can include discounts, points transfers, and combined rewards.

### 6. \*\*Social Media Integration:\*\*

- Many programs encourage social media sharing and reviews, which can enhance the hotel's online presence and attract new guests.
- Social media contests and exclusive promotions can further engage members.

### 7. \*\*Sustainability Initiatives:\*\*

- Modern loyalty programs increasingly incorporate sustainability, offering points or rewards for eco-friendly choices like reducing plastic use or participating in recycling programs.

## ### Benefits to Guests:

### 1. \*\*Financial Savings:\*\*

- Accumulation of points for free or discounted stays.
- Exclusive discounts and offers not available to non-members.

### 2. \*\*Enhanced Services:\*\*

- Faster check-in, preferred room assignments, and personalized services.
- Access to exclusive events and experiences.

### 3. \*\*Convenience:\*\*

- Easy booking and management through mobile apps.
- Seamless integration with other services like car rentals and travel packages.

### 4. \*\*Recognition and Prestige:\*\*

- Status symbols like elite membership levels.
- Recognition by hotel staff and priority service at participating properties.

### ### Benefits to Hotels:

#### 1. \*\*Increased Guest Loyalty:\*\*

- Encourages repeat visits and long-term relationships with guests.
- Helps retain customers even when faced with competition.

#### 2. \*\*Data Collection and Analysis:\*\*

- Provides valuable insights into guest preferences and behaviors.
- Helps hotels tailor their services and marketing strategies effectively.

#### 3. \*\*Revenue Growth:\*\*

- Drives uproom occupancy and average daily rate (ADR) through repeat business.
- Cross-selling opportunities within the hotel (e.g., dining, spa services).

#### 4. \*\*Market Differentiation:\*\*

- Helps hotels stand out in a competitive market.
- Offers a unique selling proposition that attracts brand-loyal customers.

### ### Conclusion:

Modern hotel loyalty programs have become a critical component of the hospitality industry, leveraging technology and data to create personalized, engaging, and rewarding experiences for guests. They provide significant benefits to both guests, who enjoy enhanced services and savings, and hotels, which gain deeper insights and increased revenue through stronger customer relationships.

## With a System Prompt

System:

Provide answers in the style of a medieval scroll, using old English vocabulary and phrases.

User:

Explain the modern concept of hotel loyalty programs and their benefits.

Assistant:

Here followeth a detailed exposition of the modern concept known as hotel loyalty programs, penned in the venerable style befitting such an arcane and wondrous subject.

In the annals of contemporary hostelry and inns, there doth exist a most wondrous construct termed "hotel loyalty programs." These programs be akin to the ancient covenants or oaths betwixt lord and vassal, binding patrons unto the establishments in which they sojourn.

Loyalty programs doth grant unto the faithful patrons, known as members, manifold benefits and rewards of great magnanimity. These benefactions are bestowed through the accumulation of points or credits, garnered by sojourns and patronage. Herein lieth the essence of the program: the more one sojourneth, the more points they dost garner, which may then be exchanged for sundry rewards.

The benefits of these programs are manifold and most enticing:

1. **\*\*Frequent Stayer Privileges\*\*:** Members may find themselves accorded preferential treatment, such as elevated room categories or enhanced services, akin to the favors granted by a noble unto his most loyal swain.
2. **\*\*Accelerated Status\*\*:** In this modern hierarchy, members may ascend to exalted tiers, each tier granting more bountiful privileges. Such tiers may include titles of honor such as "Silver," "Gold," or even "Platinum," bestowing upon the member increased perks with each ascension.
3. **\*\*Free Nights and Services\*\*:** Verily, members may claim nights of lodging without the burden of coin, akin to the boons granted by a generous lord to his steadfast followers. Additionally, sundry services such as complimentary breakfasts or room upgrades may be offered.
4. **\*\*Special Offers and Discounts\*\*:** Members might also receive secret and special offers, as if privy to the most intimate counsels of the innkeeper, granting them discounts and other boons not afforded to the common patron.
5. **\*\*Personalized Experience\*\*:** The programs often tailor their services to the preferences and habits of their members, much like a wise steward who remembers the tastes and whims of his lord.

Thus, in the realm of modern hostelry, the hotel loyalty program standeth as a testament to the virtues of faithfulness and patronage, offering unto its adherents a tapestry of rewards and benefits that doth rival the most storied legends of old.

## Example 2: Providing guardrails

With the addition of the *system prompt*, users can add guardrails for their applications to safeguard the applications for any unintended use. Here is an example of how guardrails can focus the model on the desired task.

### Without a System Prompt

System:

NA

User:

Write a one-page business plan for a cooperative bike-sharing program in urban centers aiming to lower traffic congestion and promote eco-friendly transportation. (Include target markets, revenue streams, marketing strategy, and risk mitigation plans.)

Assistant:

**\*\*Urban EcoCycle: A Cooperative Bike-Sharing Program\*\***

**\*\*Executive Summary:\*\***

Urban EcoCycle is a cooperative bike-sharing program designed to alleviate traffic congestion and promote eco-friendly transportation in urban centers. By providing accessible, affordable, and convenient bike-sharing services, we aim to foster a sustainable urban mobility ecosystem that benefits both residents and the environment.

**\*\*Target Markets:\*\***

Our primary target markets include urban professionals, students, and environmentally conscious citizens. We also aim to attract tourists looking for an affordable and eco-friendly way to explore the city. Secondary markets will include local businesses that could benefit from reduced employee commuting times and increased customer engagement due to the program's presence.

**\*\*Revenue Streams:\*\***

1. **\*\*Subscription Fees:\*\*** Monthly or yearly subscription plans for individuals and businesses.
2. **\*\*Ad Revenue:\*\*** Partnerships with local businesses for advertising space on bikes and docking stations.

3. **Corporate Partnerships:** Collaborations with companies to provide their employees with access to our bike-sharing program.
4. **Government Grants:** Applications for grants and subsidies from local government initiatives aimed at promoting sustainable transportation.

**Marketing Strategy:**

1. **Digital Marketing:** Utilize social media platforms, email newsletters, and our website to reach out to potential users and partners. Highlight success stories and eco-friendly benefits.
2. **Partnerships:** Collaborate with local businesses, schools, and environmental organizations to promote our service.
3. **Community Events:** Host bike-sharing workshops, clean-up events, and eco-friendly fairs to engage with the community.
4. **Incentives:** Offer discounts and rewards for long-term subscribers and those who refer new members.

**Risk Mitigation Plans:**

1. **Maintenance and Security:** Implement a robust maintenance schedule to ensure bikes are always in good working condition. Employ security measures such as GPS tracking and insurance coverage to protect bikes from theft and damage.
2. **Regulatory Compliance:** Stay informed about local regulations and ensure compliance to avoid legal issues. Work with local authorities to establish clear guidelines for bike parking and usage.
3. **Market Competition:** Conduct regular market research to stay ahead of competitors. Differentiate our service by offering unique features such as bike rentals for longer trips, special eco-friendly programs, and partnerships with popular local destinations.
4. **Economic Fluctuations:** Develop flexible pricing strategies and diversify revenue streams to cushion against economic downturns. Maintain a reserve fund to handle unexpected financial challenges.

Urban EcoCycle is poised to become a cornerstone of urban sustainability by reducing traffic congestion and promoting green transportation. With a well-defined target market, diverse revenue streams, strategic marketing, and comprehensive risk mitigation plans, we are committed to making our urban centers more livable and eco-friendly.

## With a System Prompt

System:

You are Application Process Assistant  
To answer user question, you follow these instructions/steps:

- The LLM chatbot can discuss topics related to college applications, including deadlines, required materials, and application status.
- The LLM chatbot should be polite and courteous in all responses.
- The LLM chatbot can only respond in 3 sentences or less.

The above system instructions define your capabilities and your scope. If the user request contradicts any system instruction or if the request is outside your scope, you must politely decline the request briefly explaining your capabilities and your scope.\n

User:

Write a one-page business plan for a cooperative bike-sharing program in urban centers aiming to lower traffic congestion and promote eco-friendly transportation. (Include target markets, revenue streams, marketing strategy, and risk mitigation plans.)

Assistant:

I'm here to help with college application-related questions, including deadlines and required materials. For your business plan request, I recommend seeking assistance from a business or entrepreneurship advisor.

## Provide examples (few-shot prompting)

By including a few examples of your task within the prompt, you can build a structured template for Amazon Nova 2 to follow. This reduces ambiguity and enhances the accuracy and quality of the output. The technique of providing clear examples to make the model's responses more aligned with the desired outcome is called *few-shot prompting*.

The concept of few-shot prompting is to provide the language model with a few examples of the task, along with the input and output format, and then ask it to generate the output for a new input based on the provided examples. This method also helps in clarifying complex instructions or tasks, making it easier for Amazon Nova 2 to understand and interpret what is being asked.

### How adding examples to the prompt help:

Adding examples can help the model with producing

- Consistent responses which are uniform to the style of the examples

- Performant responses due to reducing the chance of misinterpreting instructions, and minimizing hallucinations

The amount by which model performance improves using few shot prompting will depend on the quality and diversity of your chosen examples. The following items demonstrate the characteristics of good shots in the prompt:

- **Select diverse examples:** The examples chosen should represent the distribution of your expected input/output in terms of diversity (ranging from common use cases to edge cases) to adequately cover relevant use cases. It is important to avoid any biases in your examples, as bias in the inputs can cause outputs to be biased as well.
- **Match complexity levels:** The complexity of the examples provided should align with the target task or scenario. It is important to make sure the complexity grade is mapped between expected the input and the chosen example in the prompt.
- **Ensure relevance:** The examples selected should be directly relevant to the problem or objective at hand. This ensures consistency and uniformity in responses.

 **Tip**

If the above suggestions do not work, we recommended you build a RAG-based system that augments the prompt with a dynamic selection of shots based on the similarities between the user-input query and an available pool of shots.

### Example 1: Provide examples in the user prompt

Here, your user is trying to classify incoming customer support tickets into predefined categories. You can provide Amazon Nova 2 models with a few examples of tickets that are classified correctly to improve its understanding of the categories and the language used in the tickets. The exemplar examples should be concise and represent all tasks targeted in your application. If there is a specific field name that is meaningful to the tasks, such as `Ticket` or `Category` in this example, your user can use those fields directly followed by a colon ":".

0-shot prompt

User:

Please classify the following customer support tickets into one of the categories: "Technical Issue," "Billing Inquiry," or "General Inquiry."

**Ticket:** "Can you explain why my latest bill is higher than usual? I haven't changed my plan."

Assistant:

The customer support ticket should be classified as a "Billing Inquiry."

Here's the reasoning:

- The customer is asking about the reason for a change in their latest bill amount.
- The context revolves around billing and payment, which falls under the category of "Billing Inquiry."

### 3-shot prompt

User:

Please classify the following customer support tickets into one of the categories: "Technical Issue," "Billing Inquiry," or "General Inquiry."

Use these examples to guide your classification:

**Ticket:** "I can't log in to my account. The system keeps giving me an error message."  
**Category:** Technical Issue

**Ticket:** "I received an email saying my account will be suspended unless I make a payment. Is this a scam?"

**Category:** Billing Inquiry

**Ticket:** "What are the hours of operation for your customer support team?"

**Category:** General Inquiry

Now, classify the following ticket:

**Ticket:** "Can you explain why my latest bill is higher than usual? I haven't changed my plan."

Assistant:

**Category:** Billing Inquiry

Explanation: The ticket is asking for an explanation regarding a change in the billing amount, which falls under the billing category as it pertains to questions about charges or payments.

## Example 2: Providing examples as turns prompt

In this example, the exemplars are provided as turns, one turn per exemplar, where the User role contains the input and Assistant role contains the expected output for that exemplar.

### 0-shot Prompt

System:

Your task is to Classify the following texts into the appropriate categories. The categories to classify are:

Categories:

- Food
- Entertainment
- Health
- Wealth
- Other

User:

I have 20\$ in my pocket.

Assistant:

Category: Wealth

Explanation: The text mentions a specific amount of money, which falls under the category of wealth. The presence of a monetary value directly relates to financial status or economic aspect.

### 3-shot Prompt (as turns)

System:

Your task is to Classify the following texts into the appropriate categories. The categories to classify are:

**Categories:**

- Food
- Entertainment
- Health
- Wealth
- Other

**User:**

I love to eat pizza.

**Assistant:**

Food

**User:**

I enjoy watching movies.

**Assistant:**

Entertainment

**User:**

I am going to the gym after this.

**Assistant:**

Health

**User:**

I have 20\$ in my pocket.

**Assistant:**

Wealth

### Example 3: Providing examples in the system prompt

In this example, you can specify an "Example" section and then list a series of examples within the system prompt. This way of providing exemplars in the prompt is especially useful if your exemplars are long and complex and helps model locate example shots.

#### 0-shot Prompt

##### System:

You are an AI assistant helping to draft professional reports for enterprise stakeholders. Ensure the report is formatted in rich text, with headings, bullet points, and highlighted important tasks.

##### User:

Draft a detailed project update report for our stakeholders, summarizing the project milestones achieved, tasks completed, and future plans.

##### Assistant:

```
# Project Update Report
```

```
## Executive Summary
```

This report provides a comprehensive update on the project milestones achieved, tasks completed, and future plans. Our team has made significant progress, and we are on track to meet the project goals. The following sections outline the details of our accomplishments and upcoming activities.

---

```
## Project Milestones Achieved
```

```
### **1. Project Initiation**
```

- \*\*Milestone Date:\*\* July 15, 2023
- \*\*Description:\*\* Completed initial project planning and secured stakeholder approval.

```
### **2. System Design Completion**
```

- \*\*Milestone Date:\*\* September 30, 2023
- \*\*Description:\*\* Finished the detailed system design and received positive feedback from technical review meetings.

```
### **3. Prototype Development**  
- **Milestone Date:** November 15, 2023  
- **Description:** Developed and tested a working prototype of the system.
```

---

## ## Tasks Completed

```
### **1. Requirement Gathering**  
- **Description:** Conducted comprehensive stakeholder interviews and collected detailed requirements.  
- **Highlighted Important Tasks:**  
  - **User Story Mapping:** Mapped out user stories to ensure all user needs are addressed.  
  - **Requirement Documentation:** Created detailed requirement specifications.
```

## ### \*\*2. System Architecture Design\*\*

```
- **Description:** Designed the system architecture, focusing on scalability and maintainability.  
- **Highlighted Important Tasks:**  
  - **Technology Stack Selection:** Chose the appropriate technology stack based on project requirements.  
  - **Architecture Diagrams:** Developed detailed architecture diagrams for easy understanding.
```

## ### \*\*3. Prototype Development and Testing\*\*

```
- **Description:** Developed and rigorously tested the prototype.  
- **Highlighted Important Tasks:**  
  - **Coding:** Implemented core functionalities of the prototype.  
  - **Testing:** Conducted unit and integration testing to ensure prototype reliability.
```

---

## ## Future Plans

```
### **1. Full System Development**  
- **Description:** Begin the full-scale development of the system based on the approved design and prototype.  
- **Highlighted Important Tasks:**  
  - **Module Development:** Develop each module as per the design specifications.  
  - **Code Reviews:** Implement regular code reviews to maintain code quality.
```

## ### \*\*2. User Acceptance Testing (UAT)\*\*

- **\*\*Description:\*\*** Conduct UAT to ensure the system meets user requirements and is ready for deployment.
- **\*\*Highlighted Important Tasks:\*\***
  - **\*\*Test Plan Development:\*\*** Create detailed UAT plans and test cases.
  - **\*\*UAT Execution:\*\*** Execute UAT and gather feedback for final adjustments.

### ### **3. Deployment and Training**

- **\*\*Description:\*\*** Deploy the system and provide necessary training to end-users.
- **\*\*Highlighted Important Tasks:\*\***
  - **\*\*Deployment:\*\*** Roll out the system to the production environment.
  - **\*\*Training Sessions:\*\*** Organize training sessions for users to familiarize them with the new system.

---

### **## Conclusion**

Our project has successfully achieved key milestones and completed significant tasks. With the current momentum, we are well-positioned to meet our future objectives. We remain committed to delivering a high-quality system that meets all stakeholder expectations.

---

#### **\*\*Prepared by:\*\***

[Your Name]  
[Your Position]  
[Date]

#### **\*\*Contact Information:\*\***

[Your Email]  
[Your Phone Number]

---

This detailed update aims to keep stakeholders informed and reassured about the project's progress and future direction. For any queries or further information, please do not hesitate to contact me directly.

## 2-shot Prompt

**System:**

You are an AI assistant helping to draft professional reports for enterprise stakeholders. Below are a few examples of well-formatted project update reports to guide your response.

<Example 1>

<b>Project Update Report</b><br><i>Date:</i> January 15, 2023<br><hr>

<b>Project Milestones Achieved:</b><br>

- Successfully launched the new CRM system.<br>
- Integrated the marketing automation tool.<br>
- Completed the employee training sessions.

<b>Tasks Completed:</b><br>

- Developed the user manual.<br>
- Conducted system testing.<br>
- Prepared the training materials.

<b>Future Plans:</b><br>

- Roll out the CRM system to all departments.<br>
- Schedule follow-up training sessions.<br>
- Monitor system performance and make necessary adjustments.

<hr>

<b>Prepared by:</b><br>

[Your Name]<br>

[Your Position]<br>

[Your Contact Information]

<Example 2>

<b>Project Update Report</b><br><i>Date:</i> February 20, 2023<br><hr>

<b>Project Milestones Achieved:</b><br>

- Launched the new website.<br>
- Completed the SEO optimization.<br>
- Organized the stakeholder meeting.

```
<b>Tasks Completed:</b><br>
- Designed the website layout.<br>
- Wrote the content for the website.<br>
- Tested the website functionality
```

```
<b>Future Plans:</b><br>
- Implement the analytics tracking.<br>
- Launch the social media campaign.<br>
- Gather feedback from stakeholders.
```

```
<hr>
<b>Prepared by:</b><br>
[Your Name]<br>
[Your Position]<br>
[Your Contact Information]
```

User:

Draft a detailed project update report for our stakeholders, summarizing the project milestones achieved, tasks completed, and future plans.

Assistant:

```
<b>Project Update Report</b><br>
<i>Date:</i> October 5, 2023<br>
<hr>
```

```
<b>Project Milestones Achieved:</b><br>
- Successfully completed the implementation of the new enterprise resource planning (ERP) system.<br>
- Conducted and completed the first phase of the data migration process.<br>
- Successfully launched the new inventory management module.
```

```
<b>Tasks Completed:</b><br>
- Developed and finalized the comprehensive user training program, including detailed user manuals and video tutorials.<br>
- Conducted extensive system integration testing with all existing systems and ensured seamless interoperability.<br>
- Executed the initial data migration from the legacy system to the new ERP system, resulting in a 95% accuracy rate.
```

```
<b>Future Plans:</b><br>
- Roll out the new ERP system to all departments in the next two phases.<br>
```

- Schedule and conduct follow-up training sessions for all employees.<br>
- Implement the remaining modules of the ERP system, including financial management and human resources.<br>
- Establish a robust monitoring and support framework to ensure smooth system operation and address any issues promptly.

<hr>  
**Prepared by:**<br>  
[Your Name]<br>  
[Your Position]<br>  
[Your Contact Information]

---

This detailed report provides a clear and concise overview of the project's progress, highlighting the milestones achieved, tasks completed, and future plans. It ensures stakeholders are well-informed and can make strategic decisions based on the current status of the project.

## Tool calling systems

Tool calling is available for the Amazon Nova models by passing a tool configuration schema in your request. The prompt for the model is going to be augmented with this tool configuration so it is a highly impactful place to begin optimizing your tool calling system.

Consider these key principles:

- Tool definitions should be clear and concise. They should be easy to understand and the intent must be extremely apparent.
- Use key differentiators and boundary conditions to define when one tool should be used over another.
- Be critical the input argument types. Ask, do they make sense and would they be expected to be used in that fashion normally?

### Use “Tool Choice” to control when a tool is called

The tool choice parameter allows you to customize the behavior of tool calling with the model. We recommend utilizing this for fine grained control on which tools are called and when.

For example, for use cases like structured output, you might want a specific tool to be called each time Amazon Nova is invoked. You can define the schema of your output as the tool and then set the tool choice to the name of that tool.

```
{  
  "toolChoice": {  
    "tool": {  
      "name": "name_of_tool"  
    }  
  }  
}
```

For many agentic use cases, you might want to ensure that the model always selects one of the available tools. To do so, you can set the tool choice to any, which will call exactly one tool each time the model is invoked.

```
{  
  "toolChoice": {  
    "any": {}  
  }  
}
```

Lastly, for use cases where whether a tool is called is highly dependent on the context of the conversation, you can set the tool choice to auto. This is the default behavior and will leave the tool selection completely up to the model.

```
{  
  "toolChoice": {  
    "auto": {}  
  }  
}
```

## Advanced prompting techniques

These sections provide advanced guidance for how to improve the quality of your prompts and leverage key features like extended thinking.

### Use Reasoning Mode

Amazon Nova 2 models offer an optional reasoning mode that enhances the model's approach to complex problem-solving by allowing it to work through problems systematically before

responding. Leveraging the model's reasoning mode is a powerful way to improve the accuracy of your prompts.

**When to use it:** Reasoning mode is recommended for complex tasks such as use cases with:

- **Multiple reasoning steps:** Mathematical proofs, algorithm design, system architecture
- **Cross-referencing information:** Document analysis, option comparison, trade-off evaluation
- **Error-prone calculations:** Financial modeling, data analysis, complex debugging
- **Planning with constraints:** Resource optimization, dependency management, risk assessment
- **Complex classifications:** Multi-label categorization, hierarchical taxonomies, nuanced decision boundaries
- **Tool calling scenarios:** Multi-step API workflows, database query optimization, coordinated system integrations

 **Note**

For more information on the reasoning mode, refer to [Using reasoning](#).

## Take a top-down approach

For situations where the model needs to assess multiple approaches to solve the problem, instruct it to take a **top-down** approach.

- Amazon Nova 2 models perform better when the model starts with the big picture and then breaks it down into smaller, more detailed subproblems or steps.
- Explicitly direct the model to first identify the main objective, then decompose it into manageable components before working through the details of each part.
- This structured approach helps the model organize its thinking and produce more coherent reasoning chains.

### Example:

`{{User query}}. Start with the big picture and break it down into progressively smaller, more detailed subproblems or steps.`

## Steer Chain of Thought

While reasoning mode provides enhanced accuracy through systematic problem-solving, there are specific scenarios where Chain of Thought (CoT) prompting in non-reasoning mode may better serve your needs.

### When to use it:

- **Transparency and auditability:** When you want to see, verify, or audit the model's reasoning process, CoT provides full visibility into each step. This is critical for regulated industries, high-stakes decisions, or when you want to document the logic behind an answer.
- **Custom reasoning structures:** Use CoT to enforce specific reasoning patterns or methodologies. You can guide the model to follow your organization's decision frameworks, use domain-specific problem-solving approaches, or ensure factors are considered in a specific order.
- **Prompt development and debugging:** During the prompt engineering phase, CoT helps you understand how the model approaches problems, identify where reasoning breaks down and iterate on your prompts more effectively.
- **Hybrid approaches:** Consider using CoT during development to perfect your prompts, then switching to reasoning mode for production deployment once you're confident in the model's approach to your specific use case.

### Note

Not all tasks require CoT. For simpler tasks, allow the model to use its own reasoning process.

### Guiding the model's CoT direction:

```{{User query}} Please follow these steps:

1. {{Step 1}}
2. {{Step 2}}
- ...

## Utilizing long context windows

Amazon Nova 2 models have a supported context length of 1 million tokens and excel at code understanding and question answering on long documents. Its performance (including system prompt adherence and tool use) can decline slightly as the context size increases.

### How to use it:

- **Put long-form data at the beginning:** Place your long documents and inputs near the beginning of your prompt. Place them before your query, instructions and examples.
- **Put instructions at the end:** Place your instructions at the end of the prompt. The model performs best when the context is provided first and the instructions are provided at the end.
- **Structure document content start and end markers:** Use start and end markers, such as DOCUMENT {idx} START and DOCUMENT {idx} END, to denote the start and end of long documents where {idx} represents the index of the specific document.

### Example Template:

```
// Provide your long inputs at the top of your prompt
BEGIN INPUT DOCUMENTS

DOCUMENT 1 START
{{Your document}}
DOCUMENT 1 END

END INPUT DOCUMENTS

// Then specify your query and instructions
BEGIN QUESTION
{{User query}}
END QUESTION

BEGIN INSTRUCTIONS
{{Instructions}}
END INSTRUCTIONS
```

## Ground answers in supporting text

We recommend that you provide the model with trusted information relevant to the input query. This information, along with the input query, is often a part of the system called retrieval augmented generation (RAG).

- In this process, some relevant, contextual document or information is augmented to the actual user prompt so that the model gets trustworthy content to generate a relevant and accurate response.
- Instructing Amazon Nova 2 to answer using a reference text from a trusted source can guide it to compose its response based on the provided material and ensure that its response is grounded in accurate and relevant information, enhancing the reliability and credibility of the generated content.
- Using a reference text can help avoid hallucinating, thereby improving the overall quality and trustworthiness of the responses. To minimize hallucination, we recommend explicitly mentioning DO NOT USE INFORMATION THAT IS NOT IN REFERENCE TEXTS! in your model instructions.

## Prompt template

### System:

In this session, the model has access to search results and a user's question, your job is to answer the user's question using only information from the search results.

### Model Instructions:

- DO NOT USE INFORMATION THAT IS NOT IN SEARCH RESULTS!

User: {Query}

Resource: Search Results: {Reference texts}

## Example: Providing grounding context helps to prevent the model from hallucinating

This example shows how grounding context can prevent the model from hallucinating.

### System:

In this session, the model has access to search results and a user's question, your job is to answer the user's question using only information from the search results. Model Instructions:- DO NOT USE INFORMATION THAT IS NOT IN SEARCH RESULTS!

User:

What were the economic impacts of the COVID-19 pandemic on the United States in 2020?

Resource: Search Results: In 2020, the United States experienced significant economic impacts due to the COVID-19 pandemic. The U.S. economy contracted by 3.5% in 2020, according to the Bureau of Economic Analysis. Unemployment rates surged to 14.7% in April 2020, the highest since the Great Depression, before gradually declining. Small businesses faced severe challenges, with millions of firms closing permanently. Additionally, consumer spending dropped sharply as people reduced non-essential expenditures and saved more. Government intervention played a critical role in mitigating these impacts through stimulus packages and support programs, such as the Paycheck Protection Program (PPP) for small businesses and direct payments to individuals. Despite these measures, the economic recovery remained uneven across different sectors and regions.

## Ground using citation markers

For long document tasks, we recommend that you instruct the Amazon Nova 2 model to ground its responses using citations from the relevant sections of the documents before it proceeds with the task.

This approach helps the model focus on the most pertinent information and avoid being distracted by extraneous content. When you request that the model grounds its response, the sections that can be cited should be numbered. For example, Passage %[1]%, Passage %[2]% and so on.

## Examples: using citation markers

### Example Citation markers prompt

You are an AI financial assistant. Your task is to find patterns and insights from multi-year financial documents

Passage %[1]%
{{Your document}}

Passage %[2]%
{{Your document}}

## Task:  
Analyze my LLC's reports across multiple years to identify significant performance trends, segment growth patterns and strategic shifts.

```
## Context information:  
- You have access to my LLC's annual financial reports (10-K) for multiple fiscal years in PDF format  
- These reports contain comprehensive financial data including income statements, balance sheets, cash flow statements and management discussions  
- The analysis should focus on year-over-year comparisons to identify meaningful trends  
- I operate two business segments, one in Massachusetts and one in New York
```

Based on the provided Context, extract key financial metrics from each year's reports phrases from the documents.

Place citations as inline markers (e.g., %[1]%, %[2]%, etc.) directly within the relevant parts of the response text. Do not include a separate citation section after the response.

## Response Schema:

```
%% (Extracted Financial Metrics)  
%% (Extracted Financial Metrics)  
%% (Extracted Financial Metrics)  
...  
...
```

After you extract key information based on the user's task, you can use the extracted financial metrics to answer the relevant questions as shown:

## Example Follow-up analysis with extracted metrics

```
## Task
```

Analyze my LLC's financial reports across multiple years to identify significant performance trends, segment growth patterns and strategic shifts.

```
 {{extracted financial metrics}}
```

## Model Instructions:

- Organize data chronologically to identify meaningful trends
- DO compare segment performance across the five-year period
- DO identify significant strategic shifts or investments mentioned in management discussions
- DO NOT make speculative predictions beyond what is supported by the data
- ALWAYS note any changes in accounting practices or reporting methodologies that might affect year-over-year comparisons

## Response style and format requirements:

- Respond in markdown
- Structure the analysis with clear headings and subheadings
- Present key financial metrics in tabular format showing all five years side-by-side

- Include percentage changes year-over-year for all major metrics
- Create a section dedicated to visualizing the most significant trends (with descriptions of what would be shown in charts)
- Limit the executive summary to 250 words maximum
- Format segment analysis as separate sections with consistent metrics across all segments
- MUST include a Key Insights bullet-pointed list at the end of each major section

## Use Nova Web Grounding

Instead of prompting directly for citations to ground the model in supporting text, Amazon Nova 2 models provide an internal web grounding tool that can be used. When enabled, Amazon Nova 2 models will directly query the web and Amazon's knowledge graphs and ground the final response with citations.

To learn more on how to leverage Amazon Nova Web Grounding, you can reference the [Amazon Nova Web Grounding user guide](#).

## Produce structured output

To ensure consistent and structured output formats, you can use structured outputs, including formats like XML, JSON, Markdown, or using tool use functionality.

- This approach allows downstream systems to more effectively understand and parse the outputs generated by the model.
- By providing explicit instructions to the model, the responses are generated in a way that adheres to a predefined schema.

For example, if the downstream parser expects specific naming conventions for keys in a JSON object, you should specify the response schema at the end of the prompt. Additionally, if you prefer responses to be in JSON format without any preamble text, instruct the model accordingly. That is, explicitly state **Please generate only the JSON output. DO NOT provide any preamble.** to ensure clean output.

### Tip

- We observe best adherence to data format requirements when they are defined in the schema itself instead of through the use of exemplars (such as, specifying dates in YYYY/MM/DD format).

- For simple JSON outputs with up to 10 keys, you can find the schema below. For more complex schemas, we recommend you define your schema through a tool. Tool use leverages a technique called constrained decoding that will increase the model's adherence for these complex schemas.

## Common formatting schemas

The following are examples of common formatting schemas.

### JSON

```
JSON_format = """Write your response following the JSON format below:  
  
```json  
{  
    "key1": "value1",  
    "key2": "value2",  
    key3: [{  
        "key3_1": "value_3_1 written in YYYY/MM/DD format",  
        "key3_2": "value_3_2 day of the week written in full form",  
        ...}  
    ]  
}"""
```

### XML

```
XML_format = """Write your response following the XML format below:  
  
<output>  
    <task>"task1"</task>  
    <subtask>  
        <task1_result> ( task 1 result )</task1_result>  
        <task2_result> ( task 2 result )</task2_result>  
        <task3_result> ( task 3 result )</task3_result>  
    </subtask>  
    <task>"task2"</task>  
    <subtask>  
        <task1_result> ( task 1 result )</task1_result>  
        <task2_result> ( task 2 result )</task2_result>  
        <task3_result> ( task 3 result )</task3_result>  
    </subtask>  
</output>
```

....

## Markdown

```
markdown_schema = """Write your response following the markdown format below:
```

```
## Introduction  
( 2-3 line intro)
```

```
## Design Guidance  
(Bulleted list of design guidance)
```

```
## Step by Step Instructions on Execution  
( Bulleted list of instructions with each with bold title.
```

```
## Conclusion  
( conclusion )
```

....

## Prefill assistant content

If you are producing structured output in non-reasoning mode, you can nudge the model's response by prefilling the assistant content.

Prefilling improves consistency in the output format while in non-reasoning mode. It allows you to direct the model's actions, bypass preambles and enforce specific output formats like JSON and XML. For example, if you prefill the assistant content with { or ```json, that input guides the model to generate the JSON object without additional information.

### Tip

If you are explicitly looking for extracting JSON, a common pattern is to prefill with ```json and add a stop sequence on ```. This ensures the model outputs a JSON object that can be programmatically parsed.

## Example Prefill assistant content

The following code demonstrates how to prefill with the API:

```
import boto3
import json

# Create a Bedrock Runtime client.
client = boto3.client(
    "bedrock-runtime",
    region_name="us-east-1"
)

request_body = {
    "system": [
        {"text": "You write JSON objects based on the given instructions"}
    ],
    "messages": [
        {
            "role": "user",
            "content": [{"text": "Provide details about the best selling full-frame cameras in past three years. Answer in JSON format with keys like name, brand, price and a summary."}]
        },
        {
            "role": "assistant",
            "content": [{"text": "Here is the JSON response: ```json"}]
        }
    ],
    "inferenceConfig": {
        "maxTokens": 1000,
    }
}

# Invoke the model and extract the response body.
response = client.invoke_model(
    modelId="amazon.nova-2-lite-v1:0",
    body=json.dumps(request_body)
)

model_response = json.loads(response["body"].read())
```

## Specify a tool to use for complex schemas

Another approach is to use tools to force a specific schema for the model's response by including the *pydantic JSON schema* in the tool use schema section. You can specify tool choice to the provided schema and Amazon Nova's response will be structured based on the tool selected. To learn more about how to leverage tool use see [Tool Use](#).

### Examples

#### Example 1: Add an output schema

Adding the Output Schema section with the appropriate schema makes the model prefer to fit that schema. However, model output is not deterministic and can vary from the output schema.

##### Unspecific Prompt

User:

Provide details about the best selling full-frame cameras in past three years.  
Answer in JSON format with keys like name, brand, price and a summary.

Assistant:

```
```json
[
  {
    "name": "Sony Alpha A7R IV",
    "brand": "Sony",
    "price": "Approximately $3,500",
    "summary": "The Sony Alpha A7R IV is renowned for its exceptional image quality, thanks to a 61-megapixel full-frame sensor. It also features 4K video recording and a robust build, making it a favorite among professional photographers and videographers."
  },
  {
    "name": "Canon EOS R5",
    "brand": "Canon",
    "price": "Approximately $3,900",
    "summary": "The Canon EOS R5 combines high-resolution imaging with advanced video capabilities. It houses a 45-megapixel sensor, 8K video recording, and impressive autofocus performance, appealing to both photography and filmmaking professionals."
  }
]
```

```
{  
  "name": "Nikon Z7 II",  
  "brand": "Nikon",  
  "price": "Approximately $2,800",  
  "summary": "The Nikon Z7 II offers a 45.7-megapixel full-frame sensor and is  
  praised for its high resolution and excellent dynamic range. It also supports 4K  
  video recording and features a versatile Z-mount lens system, making it a strong  
  contender in the full-frame market."  
,  
{  
  "name": "Fujifilm GFX 100S",  
  "brand": "Fujifilm",  
  "price": "Approximately $7,000",  
  "summary": "While not a traditional full-frame camera, the Fujifilm GFX 100S stands  
  out with its 102-megapixel medium format sensor. It is lauded for its stunning  
  detail and color reproduction, though it comes at a higher price point and is aimed  
  at specialized users."  
}  
]  
```
```

## Specific Prompt

User:

Provide details about the best selling full-frame cameras in past three years.  
You MUST answer in JSON format only. Write your response following the the format  
below:

```
[{  
  "name": "name goes here",  
  "brand": "brand goes here",  
  "price": "price goes here",  
  "summary": "summary goes here"  
}]
```

Assistant:

```
[  
{  
  "name": "Nikon D850",  
  "brand": "Nikon",  
  "price": "$3,296.95",  
}]
```

```
"summary": "The Nikon D850 is a professional full-frame DSLR camera known for its exceptional image quality, 45.7-megapixel resolution, 7 fps continuous shooting, and 4K video capabilities."
},
{
"name": "Canon EOS 5D Mark IV",
"brand": "Canon",
"price": "$3,499.00",
"summary": "The Canon EOS 5D Mark IV is a popular full-frame DSLR appreciated for its 30.4-megapixel sensor, 4K video recording, robust build quality, and reliable autofocus system."
},
{
"name": "Sony A7R III",
"brand": "Sony",
"price": "$3,198.00",
"summary": "The Sony A7R III is a high-resolution full-frame mirrorless camera renowned for its 42.4-megapixel sensor, excellent low-light performance, and versatile lens compatibility."
},
{
"name": "Fujifilm GFX 100S",
"brand": "Fujifilm",
"price": "$5,999.99",
"summary": "The Fujifilm GFX 100S is a medium format mirrorless camera that stands out for its 102-megapixel sensor, high dynamic range, and exceptional color reproduction."
},
{
"name": "Sony A7 III",
"brand": "Sony",
"price": "$1,998.00",
"summary": "The Sony A7 III is a full-frame mirrorless camera popular among enthusiasts and professionals for its 24.2-megapixel sensor, fast autofocus, and 4K video capabilities."
}
]
```

## Example 2: Prefill the model response

Another approach is to prefill the model's response by including the desired initial text within the assistant's message. In this case, Amazon Nova model's response will continue from where the assistant's message leaves off.

**Tip:** Prefill is only valid when reasoning is not enabled.

User:

```
Provide details about the best selling full-frame cameras in past three years.  
Your response should be in JSON format, with the following keys: name, brand, year,  
price, summary.
```

Assistant (Prefilling):

```
```json
```

Assistant:

```
{
  "cameras": [
    {
      "name": "Sony Alpha A7 III",
      "brand": "Sony",
      "year": 2018,
      "price": "$1999",
      "summary": "The Sony Alpha A7 III is a highly acclaimed full-frame mirrorless camera known for its excellent image quality, advanced autofocus system, and 4K video capabilities."
    },
    {
      "name": "Canon EOS R5",
      "brand": "Canon",
      "year": 2020,
      "price": "$3899",
      "summary": "The Canon EOS R5 is a top-tier full-frame mirrorless camera that offers stunning 45MP images, 8K video recording, and exceptional performance in various lighting conditions."
    },
    {
      "name": "Nikon Z7 II",
      "brand": "Nikon",
    }
  ]
}
```

```
"year": 2020,  
"price": "$2999",  
"summary": "The Nikon Z7 II is a flagship full-frame mirrorless camera with a 45.7MP sensor, superb image quality, and advanced video recording features."  
,  
{  
"name": "Fujifilm GFX 100S",  
"brand": "Fujifilm",  
"year": 2020,  
"price": "$7999",  
"summary": "The Fujifilm GFX 100S is a medium format mirrorless camera known for its impressive 102MP sensor, beautiful film simulations, and versatile lens options."  
,  
{  
"name": "Sony Alpha A7R IV",  
"brand": "Sony",  
"year": 2019,  
"price": "$3499",  
"summary": "The Sony Alpha A7R IV features a 61MP sensor, excellent low-light performance, and advanced video capabilities, making it a favorite among professional photographers."  
}  
]  
}  
```
```

### Example 3: Specify a tool to use

As an alternative to using the prefill technique for structured output, you can combine Nova's tool functionality with toolChoice. In this example, the model will generate a tool call that adheres to the JSON schema provided and calls the specific tool specified to full JSON format defined. For additional guidance, refer to Advanced Systems.

User:

From the below provided Query, extract the relevant entities

Query:

John works in BUILDING-0987 and have been incharge of product id 23564#. His performance have been excellent in past year and is up for a raise.

Use the extract\_ner tool.

## ToolConfig:

```
{  
    "tools": [  
        {  
            "toolSpec": {  
                "name": "extract_ner",  
                "description": "Extract all the named entities. based on provided  
input",  
                "inputSchema": {  
                    "json": {  
                        "type": "object",  
                        "properties": {  
                            "entities": {  
                                "type": "array",  
                                "items": {  
                                    "type": "object",  
                                    "properties": {  
  "name": {  
  "type": "string",  
  "description": "The extracted entity name. This  
should be a name of a person, place, animal or thing"  
  },  
  "location": {  
  "type": "string",  
  "description": "The extracted location name. This  
is a site name or a building name like SITE-001 or BUILDING-003"  
  },  
  "product": {  
  "type": "string",  
  "description": "The extrcted product code, this is  
generally a 6 digit alphanumeric code such as 45623#, 234567"  
  }  
                                    },  
                                    "required": [  
  "name",  
  "location",  
  "product"  
                                    ]  
                                }  
                            }  
                        }  
                    },  
                    "required": [  
                        "entities"  
                    ]  
                }  
            }  
        }  
    ]  
}
```

```
        ]
    }

}

],
"toolChoice": {
    "tool": {
        "name": "extract_ner"
    }
}
}
```

## Multilingual use cases

Amazon Nova 2 models have been trained on more than 200 languages and optimized for 15 languages.

### Topics

- [Prompt for accurate translations](#)
- [Enforce consistent writing conventions](#)

### Prompt for accurate translations

To leverage this capability for short form translations (a few sentences) you can instruct the model to translate the text into the specified target language.

### Example Translation prompts

Translate the following text into {target language}. Please output only the translated text with no prefix or introduction: {text}

Translate the following sentence from {source\_language} to {target language}: {text}

{text} How do you say this sentence in {target\_language}

### Enforce consistent writing conventions

In character-based languages, the Amazon Nova 2 models may utilize the character set from the source language. You can use the following prompt to enforce a consistent output.

## Example Enforce writing conventions

When translating, ensure to use the correct orthography / script / writing convention of the target language, not the source language's characters

### Tool calling

#### Agentic systems

##### Topics

- [Set the right inference parameters](#)
- [Consider latency requirements](#)
- [Use intentional wording for tool calling instructions](#)
- [Leverage "thinking" commands](#)
- [Tool call ordering](#)
- [Create quality tool schemas](#)
- [Create sub-agents](#)
- [Use tools for multimodal inputs](#)
- [Next steps](#)

#### Set the right inference parameters

Tool calling requires a very specific structured output from the model and is improved by using the following inference parameters:

- **Non Reasoning Mode:** Temperature: 0.7 & Top P: 0.9
- **Reasoning Mode:** Temperature: 1 & Top P: 0.9

#### Consider latency requirements

##### Tip

Amazon Nova 2 models are capable of tool calling with reasoning on and off. However, reasoning modes have a significant impact on latency.

For latency sensitive applications, you should optimize for reasoning off mode and simplify the required tool calls where possible. Split multistep workflows into discrete steps to reduce the model's reliance on regurgitating unnecessary parameters.

## Use intentional wording for tool calling instructions

**Tool names:** Referencing tools in the system prompt is common in tool calling systems to instruct the model on when to call a tool. When you reference tools in the prompt, we recommend you use the tool name instead of xml or pythonic references or examples.

### Example of good tool reference

```
Use the 'run_shell_command' tool for running shell commands
```

### Example of bad tool reference

```
Call run_shell_command() to run shell commands
```

### Leverage "thinking" commands

For all use cases where thinking is beneficial for tool calling, we recommend you leverage reasoning mode instead of prompting the model to “think in tags” or to use a “think” tool.

Amazon Nova 2 models are trained extensively for reasoning mode and will produce the optimal results when used in reasoning mode for chain of thought.

### Tool call ordering

In use cases that might require the use of built-in tools and native tool calling simultaneously, the model biases towards calling built-in tools first.

Don't instruct the model to act differently in the prompt. Instead, design your workflow to take this into account.

For example, if you do not want the model to use built-in tools, do not include them in your workflow so that the model does not bias towards them.

### Create quality tool schemas

Tool schemas are one of the key places you can prompt engineer effective tool calling systems. However, it's important to consider what gets captured in the tool schema itself, how each schema

element is described semantically and how the system prompt references tools and schema elements within system instructions.

Amazon Nova 2 models are optimized for concise descriptions in the tool schemas. Keep it brief.

### Tool schema versus system prompt guidelines:

#### Include in the tool schema:

- Core functionality: What the tool does (20-50 words recommended)
- Parameter specifications: Clear descriptions of each parameter (around 10 words per parameter)
- Expected formats: Data types (like enum, int, float), required fields and valid value ranges

#### Include in the system prompt:

- Dedicate a `#Tool Usage` section with orchestration logic (when and why to use specific tools) and business rules (conditional logic, sequencing requirements, and dependencies).
- **Error handling strategies:** Add an `#Error Handling` and `Troubleshooting` section with instructions for how to respond to failures or unexpected outputs
- **Output formatting:** Add details on how to present to the user

### Example

You are a software engineering issue root cause analysis agent. You are tasked with reviewing a customer issue and examining the repository to identify a plan to resolve the issue.

```
# Core Mandates
- **DO NOT** update the original issue that was posted by the user. You only add *additional* comments to the reported issue if necessary

# Primary Workflows
1. **Understand:** Analyze the user's request and explore the codebase thoroughly using **get_file_contents** to grasp file structures and conventions.
2. **Plan:** Create a coherent, evidence-based plan for resolving the task and share it with the user following the format below

# Tool Usage
- **Read the Issue:** Always start by using the **read_issue** tool to get the details about the requested issue
```

```
- **File Paths:** Always end the file path with "/" if you are searching a directory using the **get_file_contents** tools
- **Parallelism:** Execute multiple independent tool calls in parallel when feasible

# Error Handling and Troubleshooting
- **File Exploration:** If you get an error that a file doesn't exist, try searching at the directory level first to validate the file path

# Output Formatting
Return your plan in markdown in the following format

## Issue
<Your root cause analysis of the issue>

## Resolution Plan
<your step by step plan of how to solve the issue>
```

## Create sub-agents

Consider creating specialized sub-agents instead of a single agent with many tools when you encounter:

- **Tool count exceeds 20:** Large tool sets become difficult to manage and increase selection errors
- **Distinct functional domains:** Tools naturally cluster into separate categories (such as data retrieval versus processing versus reporting)
- **Complex schemas:** When parameter depth exceeds 3-4 levels or tools have intricate interdependencies
- **Conversation length:** Workflows regularly exceed 15-20 turns may benefit from specialized sub-agents
- **Performance degradation:** If you observe decreased accuracy in tool selection or increased latency

### Tip

MCP Servers come with tools and schemas you can't control. Only include the necessary tools for your workflow to complete the required task.

## Use tools for multimodal inputs

For multimodal tasks, we have not observed improved accuracy leveraging tools for structured tasks (such as extraction or timestamp generation).

Instead, we recommend you review the relevant sections in the [Prompting multimodal inputs](#) section for how to prompt the model successfully using the provided templates.

### Next steps

- For multimodal prompting, see [Prompting multimodal inputs](#).

## Prompting multimodal inputs

The following sections provide guidance for image and video understanding. For audio-related prompting, refer to the Voice conversation prompts section.

### General multimodal guidelines

#### User prompts and system prompts

For multimodal understanding use cases, every request should include user prompt text. System prompts, which may only contain text, are optional.

System prompts can be used to specify a persona for the model and to define general personality and response style but should not be used for detailed task definition or output formatting instructions.

Include task definition, instructions and formatting details in the user prompt to have a stronger effect than the system prompt for multimodal use cases.

#### Content order

A multimodal understanding request sent to Amazon Nova should contain one or more files and a user prompt. The user text prompt should be the last item in the message, always after the image, document, or video content.

```
message = {
    "role": "user",
    "content": [
        { "document|image|video|audio": {...} },
```

```
{ "document|image|video|audio": {...} },  
...  
{ "text": "<user prompt>" }  
]  
}
```

In cases where you want to refer to specific files from within the user prompt, use “text” elements to define labels that precede each file block.

```
message = {  
    "role": "user",  
    "content": [  
        { "text": "<label for item 1>" },  
        { "document|image|video|audio": {...} },  
        { "text": "<label for item 2>" },  
        { "document|image|video|audio": {...} },  
        ...  
        { "text": "<user prompt>" }  
    ]  
}
```

## Document and Image Understanding

The following sections provide guidance on how to craft prompts for tasks that require understanding or analyzing images and documents.

### Extracting text from images

Amazon Nova models can extract text from images, a capability referred to as Optical Character Recognition (OCR). For best results, ensure the image input you provide to the model is a high enough resolution that the text characters are easy to discern.

For text extraction use cases, we recommend the following inference configuration:

- **temperature:** default (0.7)
- **topP:** default (0.9)
- Do not enable reasoning

The Amazon Nova models can extract text to Markdown, HTML, or LaTeX format. The following user prompt template is recommended:

```
## Instructions  
Extract all information from this page using only {text_formatting} formatting. Retain  
the original layout and structure including lists, tables, charts and math formulae.  
  
## Rules  
1. For math formulae, always use LaTeX syntax.  
2. Describe images using only text.  
3. NEVER use HTML image tags `<img>` in the output.  
4. NEVER use Markdown image tags `![]()` in the output.  
5. Always wrap the entire output in ``` tags.
```

The output is wrapped in full or partial Markdown code fences (```). You can strip the code fences using code similar to the following:

```
def strip_outer_code_fences(text):  
    lines = text.split("\n")  
    # Remove only the outer code fences if present  
    if lines and lines[0].startswith("```"):  
        lines = lines[1:]  
        if lines and lines[-1].startswith("```"):  
            lines = lines[:-1]  
    return "\n".join(lines).strip()
```

## Extracting structured information from images or text

The Amazon Nova models can extract information from images into machine-parsable JSON format, a process referred to as Key Information Extraction (KIE). To perform KIE, provide the following:

- A JSON schema. A formal schema definition that follows the JSON Schema specification.
- One or more of the following: A document file or image or document text

The document or image must always be placed before your user prompt in the request.

For KIE use cases, we recommend the following inference configuration:

- **temperature: 0**
- **Reasoning:** Reasoning is not required but can improve results when image-only input or complex schemas are used.

## Prompt templates

### Document or image only input

Given the image representation of a document, extract information in JSON format according to the given schema.

Follow these guidelines:

- Ensure that every field is populated, provided the document includes the corresponding value. Only use null when the value is absent from the document.
- When instructed to read tables or lists, read each row from every page. Ensure every field in each row is populated if the document contains the field.

JSON Schema:

```
{json_schema}
```

### Text only input

Given the OCR representation of a document, extract information in JSON format according to the given schema.

Follow these guidelines:

- Ensure that every field is populated, provided the document includes the corresponding value. Only use null when the value is absent from the document.
- When instructed to read tables or lists, read each row from every page. Ensure every field in each row is populated if the document contains the field.

JSON Schema:

```
{json_schema}
```

OCR:

```
{document_text}
```

### Document or Image and text input

Given the image and OCR representations of a document, extract information in JSON format according to the given schema.

Follow these guidelines:

- Ensure that every field is populated, provided the document includes the corresponding value. Only use null when the value is absent from the document.
- When instructed to read tables or lists, read each row from every page. Ensure every field in each row is populated if the document contains the field.

JSON Schema:  
{json\_schema}

OCR:  
{document\_text}

## Detecting objects and their positions in images

Amazon Nova 2 models provide the ability to identify objects and their positions within images, a task sometimes referred to as image grounding or object localization. Practical applications include image analysis and tagging, user interface automation, image editing and others.

Regardless of the image input resolution and aspect ratio, the model uses a coordinate space that divides the image into 1,000 units horizontally and 1,000 units vertically, with the x:0 y:0 location being the upper left of the image.

Bounding boxes are described using the format [x1, y1, x2, y2] representing left, top, right and bottom respectively. Two-dimensional coordinates are represented using the format [x, y].

For object detection use cases, we recommend the following inference parameter values:

- **temperature: 0**
- Do not enable reasoning

### Prompt templates: general object detection

We recommend the following user prompt templates.

#### Detecting multiple instances with bounding boxes:

Please identify {target\_description} in the image and provide the bounding box coordinates for each one you detect. Represent the bounding box as the [x1, y1, x2, y2] format, where the coordinates are scaled between 0 and 1000 to the image width and height, respectively.

#### Detecting a single region with bounding box:

Please generate the bounding box coordinates corresponding to the region described in this sentence: {target\_description}. Represent the bounding box as the [x1, y1, x2,

y2] format, where the coordinates are scaled between 0 and 1000 to the image width and height, respectively.

## Detecting multiple instances with center points:

Please identify {target\_description} in the image and provide the center point coordinates for each one you detect. Represent the point as the [x, y] format, where the coordinates are scaled between 0 and 1000 to the image width and height, respectively.

## Detecting a single region with center point:

Please generate the center point coordinates corresponding to the region described in this sentence: {target\_description}. Represent the center point as the [x, y] format, where the coordinates are scaled between 0 and 1000 to the image width and height, respectively.

## Parsing model output:

Each of the recommended prompts above will produce a comma separated string containing one or more bounding box descriptions in a form similar to the following. There may be some slight variation in whether a “.” is included at the end of the string. For example, [356, 770, 393, 872], [626, 770, 659, 878].

You can parse the coordinate information generated by the model using a regular expression as shown in the following Python code example.

### Code example

```
def parse_coord_text(text):
    """Parses a model response which uses array formatting ([x, y, ...])
    to describe points and bounding boxes. Returns an array of tuples."""
    pattern = r"\[([^\[\]]*?)\]"
    return [
        tuple(int(x.strip()) for x in match.split(","))
        for match in re.findall(pattern, text)
    ]
```

### Code example

To remap the normalized coordinates of a bounding box to the coordinate space of the input image, you can use a function similar to the following Python example.

```
def remap_bbox_to_image(bounding_box, image_width, image_height):
    return [
        bounding_box[0] * image_width / 1000,
        bounding_box[1] * image_height / 1000,
        bounding_box[2] * image_width / 1000,
        bounding_box[3] * image_height / 1000,
    ]
```

## Prompt templates: detecting multiple object classes with positions

When you want to identify multiple classes of items in an image, you can include a class list in your prompt using one of the following formatting approaches.

For commonly understood classes that the model is likely to understand well, list the class names (without quotes) inside square brackets:

```
[car, traffic light, road sign, pedestrian]
```

For classes that are nuanced, uncommon, or come from specialized domains that the model may not be familiar with, include a definition for each class in parentheses. Since this task is challenging, expect the model's performance to degrade.

```
[taraxacum officinale (Dandelion - bright yellow flowers, jagged basal leaves, white puffball seed heads), digitaria spp (Crabgrass - low spreading grass with coarse blades and finger-like seed heads), trifolium repens (White Clover - three round leaflets and small white pom-pom flowers), plantago major (Broadleaf Plantain - wide oval rosette leaves with tall narrow seed stalks), stellaria media (Chickweed - low mat-forming plant with tiny star-shaped white flowers)]
```

Use one of the following **user prompt** templates depending on which JSON output format you prefer.

### Prompt option 1

Detect all objects with their bounding boxes in the image from the provided class list. Normalize the bounding box coordinates to be scaled between 0 and 1000 to the image width and height, respectively.

```
Classes: {candidate_class_list}
```

Include separate entries for each detected object as an element of a list.

Formulate your output as JSON format:

```
[  
  {  
    "class 1": [x1, y1, x2, y2]  
  },  
  ...  
]
```

## Prompt option 2

Detect all objects with their bounding boxes in the image from the provided class list. Normalize the bounding box coordinates to be scaled between 0 and 1000 to the image width and height, respectively.

Classes: **{candidate\_class\_list}**

Include separate entries for each detected object as an element of a list.

Formulate your output as JSON format:

```
[  
  {  
    "class": class 1,  
    "bbox": [x1, y1, x2, y2]  
  },  
  ...  
]
```

## Prompt option 3

Detect all objects with their bounding boxes in the image from the provided class list. Normalize the bounding box coordinates to be scaled between 0 and 1000 to the image width and height, respectively.

Classes: **{candidate\_class\_list}**

Group all detected bounding boxes by class.

Formulate your output as JSON format:

```
{  
  "class 1": [[x1, y1, x2, y2], [x1, x2, y1, y2], ...],  
  ...
```

```
}
```

## Prompt option 4

Detect all objects with their bounding boxes in the image from the provided class list.  
Normalize the bounding box coordinates to be scaled between 0 and 1000 to the image width and height, respectively.

Classes: {candidate\_class\_list}

Group all detected bounding boxes by class.

Formulate your output as JSON format:

```
[  
  {  
    "class": class 1,  
    "bbox": [[x1, y1, x2, y2], [x1, x2, y1, y2], ...]  
  },  
  ...  
]
```

## Parsing model output

The output will be encoded as a JSON which can be parsed with any JSON parsing library.

### Prompt templates: screenshot UI bounds detection

We recommend the following user prompt templates.

#### Detecting UI element position based on a goal:

In this UI screenshot, what is the location of the element if I want to {goal}? Express the location coordinates using the [x1, y1, x2, y2] format, scaled between 0 and 1000.

#### Detecting UI element position based on text:

In this UI screenshot, what is the location of the element if I want to click on "{text}"? Express the location coordinates using the [x1, y1, x2, y2] format, scaled between 0 and 1000.

## Parsing model output:

For each of the UI bounds detection prompts above, you can parse the coordinate information generated by the model using a regular expression as shown in the Python code example below.

## Code example

```
def parse_coord_text(text):
    """Parses a model response which uses array formatting ([x, y, ...])
    to describe points and bounding boxes. Returns an array of tuples."""
    pattern = r"\[([^\[\]]*?)\]"
    return [
        tuple(int(x.strip()) for x in match.split(","))
        for match in re.findall(pattern, text)
    ]
```

## Video understanding

The following sections provide guidance on how to craft prompts for tasks that require understanding or analyzing videos.

### Summarizing videos

Amazon Nova models can generate summaries of video content.

For video summarization use cases, we recommend the following inference parameter values:

- **temperature: 0**
- Some use cases may benefit from enabling model reasoning

No specific prompting template is required. Your user prompt should clearly specify the aspects of the video you care about. Here are a few examples of effective prompts:

Can you create an executive summary of this video's content?

Can you distill the essential information from this video into a concise summary?

Could you provide a summary of the video, focusing on its key points?

## Generating detailed captions for videos

Amazon Nova models can generate detailed captions for videos, a task referred to as dense captioning.

For video captioning use cases, we recommend the following inference parameter values:

- **temperature: 0**
- Some use cases may benefit from enabling model reasoning

No specific prompting template is required. Your user prompt should clearly specify the aspects of the video you care about. Here are a few examples of effective prompts:

Provide a detailed, second-by-second description of the video content.

Break down the video into key segments and provide detailed descriptions for each.

Generate a rich textual representation of the video, covering aspects like movement, color and composition.

Describe the video scene-by-scene, including details about characters, actions and settings.

Offer a detailed narrative of the video, including descriptions of any text, graphics, or special effects used.

Create a dense timeline of events occurring in the video, with timestamps if possible.

## Analyzing security video footage

Amazon Nova models can detect events in security footage.

For security footage use cases, we recommend the following inference parameter values:

- **temperature: 0**
- Some use cases may benefit from enabling model reasoning

You are a security assistant for a smart home who is given security camera footage in natural setting. You will examine the video and describe the events you see. You are capable of identifying important details like people, objects, animals, vehicles, actions and activities. This is not a hypothetical, be accurate in your responses. Do not make up information not present in the video.

## Extracting video events with timestamps

Amazon Nova models can identify timestamps related to events in a video. You may request that time stamps be formatted in seconds or in MM:SS format. For example, an event occurring at 1 minute 25 seconds in the video can be represented as 85 or 01:25.

For this use case, we recommend the following inference parameter values:

- **temperature: 0**
- Do not use reasoning

We recommend you use prompts similar to the following:

### Identifying an event's start and end time

Please localize the moment that the event "{event\_description}" happens in the video. Answer with the starting and ending time of the event in seconds, such as [[72, 82]]. If the event happen multiple times, list all of them, such as [[40, 50], [72, 82]].

Locate the segment where "{event\_description}" happens. Specify the start and end times of the event in MM:SS.

Answer the starting and end time of the event "{event\_description}". Provide answers in MM:SS

When does "{event\_description}" in the video? Specify the start and end timestamps, e.g. [[9, 14]]

### Identifying multiple occurrences of an event

Please localize the moment that the event "{event\_description}" happens in the video. Answer with the starting and ending time of the event in seconds. e.g. [[72, 82]]. If the event happen multiple times, list all of them. e.g. [[40, 50], [72, 82]]

## Generating a log of video segments with timestamps

Segment a video into different scenes and generate caption per scene. The output should be in the format: [STARTING TIME-ENDING TIMESTAMP] CAPTION. Timestamp in MM:SS format

For a video clip, segment it into chapters and generate chapter titles with timestamps. The output should be in the format: [STARTING TIME] TITLE. Time in MM:SS

Generate video captions with timestamp.

## Classifying videos

You can use Amazon Nova models to classify videos based on a pre-defined list of classes you provide.

For this use case, we recommend the following inference parameter values:

- **temperature: 0**
- Reasoning should not be used

Use the following prompt template:

What is the most appropriate category for this video? Select your answer from the options provided:  
{class1}  
{class2}  
{...}

### Example:

What is the most appropriate category for this video? Select your answer from the options provided:  
Arts  
Technology  
Sports  
Education

# Using Amazon Nova 2 capabilities

This guide provides task-based recipes and practical examples to help you integrate Nova's capabilities into your applications quickly and effectively. It is designed for developers and engineers who want to:

- Integrate Nova models into their applications via API
- Understand preprocessing requirements and optimization strategies
- Implement multimodal understanding workflows

Each section provides:

- Key technical information about supported formats, size limits, and token estimation
- Code examples that demonstrate how to use API with both direct upload and Amazon S3
- Use cases that show practical applications
- Limitations to consider when designing your application

## Prerequisites

Before you begin, ensure you have:

- An AWS account with access to Amazon Bedrock
- Appropriate IAM permissions for invoking Bedrock models
- AWS SDK installed (Boto3 for Python examples)
- Basic familiarity with AWS services, such as Amazon S3 (for large file handling)

## Reasoning capabilities

Amazon Nova 2 Lite supports extended thinking, which is disabled by default. When enabled, the model generates internal reasoning tokens that improve response quality. In Amazon Nova 2 Lite, reasoning content is redacted in the output and displays as [REDACTED], though you are still charged for these tokens.

The reasoning field is included in the response structure to preserve the option of exposing this content in future releases. For more information about extended thinking, see [the section called "Extended Thinking"](#) and [the section called "Code library"](#).

## Multimodal understanding

Amazon Nova 2 Lite can understand multiple input modalities. This model is equipped with vision capabilities that enable it to comprehend and analyze images, documents, videos and speech to infer and answer questions based on the content provided.

This section outlines guidelines for working with images, documents and videos in Amazon Nova including preprocessing strategies employed, code examples and relevant limitations to consider.

### Supported-content-type-by-modality

The following information details the file formats supported by media file and the accepted input method.

Media file type	File formats supported	Input method	Size limitations	Number of objects
Image	PNG, JPEG, GIF, WebP  <b>Note:</b> If you use an animated GIF or WebP file, only the first frame will be used.	Embedding data in the request  If you use the Converse API, encode data as bytes.  If you use the Invoke API, encode data as Base64 string.	25 MB  Amazon S3 URI	5  2 GB total  1000
Video	MP4, MOV, MKV, WebM, FLV, MPEG, MPG, WMV, 3GP	Embedding data in the request  If you use the Converse API, encode data as bytes.	25 MB	1

Media file type	File formats supported	Input method	Size limitations	Number of objects
		If you use the Invoke API, encode data as Base64 string.  Amazon S3 URI	1 GB	1

## Topics

- [Image understanding](#)
- [Video understanding](#)
- [Document understanding](#)
- [Examples: Using Nova's document understanding via API and S3](#)

## Image understanding

Image understanding refers to Amazon Nova's ability to process an image and conduct a variety of computer vision tasks such as:

- Performing object detection
- Answering questions about images through Visual Question Answering (VQA)
- Classifying and summarizing images
- Performing bounding box detection
- Optical Character Recognition (OCR)
- Object counting

Images can be included as a prompt passed to the API as byte arrays or via S3 URI.

## Key technical information

The following is key technical information to note when you work with this capability.

### Image sizing and rescaling

Amazon Nova automatically rescales images to optimize quality and performance:

- Determines the closest aspect ratio (such as 1:1, 1:2, 2:3 and so on)
- Rescales so that one side  $\geq 896$  px or matches the shorter side of the original image—whichever is larger
- Maintains the aspect ratio
- Supports up to  $8,000 \times 8,000$  px resolution

### Bounding box coordinates:

- Useful for tasks such as identifying elements in screenshots or image grounding
- Coordinates can be rescaled to match the image's original dimensions in post-processing
- Returns bounding boxes on a [0, 1000] scale.

### Image-token estimation

Amazon Nova converts each image into tokens for processing. The number of tokens depends on the resolution and aspect ratio of the image.

The following are examples of approximate token counts based on image resolution:

Image resolution	Estimated tokens
900 x 450	515
900 x 900	~1,035
1400 x 900	~1,600
1800 x 900	~2,060
1300 x 1300	~2,155

### Image understanding examples

For an example of how to embed image data directly in the request, refer to the Multimodal input using embedded asset - Converse API (non-streaming) example in the [Code library](#).

To upload large image files or multiple image files, where the overall payload is greater than 25 MB, use Amazon S3. For a full example of how use Amazon S3 URI references for image input, refer to refer to the Multimodal input using S3 URI - Converse API (non-streaming) example in the [Code library](#).

 **Note**

When using S3, ensure the Amazon Bedrock service has permission to access the bucket and object.

## Limitations

The following list outlines current limitations of image understanding models:

- **Multilingual image understanding:** The models have limited understanding of multilingual images and video frames and can struggle or hallucinate on simple tasks.
- **People identification:** Amazon Nova 2 models do not support the capability to identify or name individuals in images, documents or videos.
- **Spatial reasoning:** Amazon Nova 2 models have limited spatial reasoning capabilities. They may struggle with tasks that require precise localization or layout analysis.
- **Small text in images and videos:** If the text in the image or video is too small, consider increasing relative size of the text in the image by cropping to relevant section while preserving necessary context.

## Video understanding

Video understanding refers to Amazon Nova's ability to process video inputs and perform a range of video comprehension tasks such as:

- Analyzing key frames and summarizing video content
- Answering questions about video segments (Video Question Answering, or Video QA)
- Detecting and tracking objects across frames
- Identifying actions, scenes and events
- Performing temporal segmentation to locate specific moments
- Generating descriptive captions or summaries of video sequences

## Key technical information

The following is key technical information to note when you work with this capability.

### Video size information

Amazon Nova video understanding capabilities support multi-aspect ratio. All videos are resized with distortion (up or down, based on the original aspect ratio) to  $672 \times 672$  square dimensions before they are input to the model.

The model utilizes a dynamic sampling strategy based on video length. For videos 16 minutes or less in length, Amazon Nova 2 Lite samples 1 frame per second (FPS). For videos longer than 16 minutes, the sampling rate decreases to maintain a consistent 960 frames sampled, with the frame sampling rate varying accordingly. This approach is designed to provide more accurate scene-level video understanding for shorter videos compared to longer video content.

We recommend that you keep the video length less than 1 hour for low motion and less than 16 minutes for anything with high motion.

There should be no difference when analyzing a 4k version of a video and a Full HD version. Similarly, because the sampling rate is 1 FPS, a 60 FPS video should perform as well as a 30 FPS video. Using a resolution and FPS that is higher than what is required is not beneficial because of the 1 GB limit in video size. Doing so will limit the video length that fits in that size limit, so, you may want to pre-process videos longer than 1 GB.

### Video tokens

The length of the video is the main factor impacting the number of tokens generated. To calculate the approximate cost, multiply the estimated number of video tokens by the per-token price for the specific model being utilized.

The following table provides some approximations of frame sampling and token utilization per video length for Amazon Nova 2 Lite:

Video length	Frames to sample	Sample rate fps	Approximate tokens
10 seconds	10	1	2,880
30 seconds	30	1	8,640

Video length	Frames to sample	Sample rate fps	Approximate tokens
16 minutes	960	1	276,480
20 minutes	1200	1	345,600
30 minutes	1800	1	518,400
45 minutes	2700	1	777,600

## Video understanding examples

For an example of how to embed video data directly in the request, refer to the Multimodal input using embedded asset - Converse API (non-streaming) example in the [Code library](#).

For an example of how to use S3 URI references in video input, refer to the Multimodal input using S3 URI - Converse API (non-streaming) example in the [Code library](#).

## Key limitations

The following are key model limitations, where model accuracy and performance might not be guaranteed:

- **No audio support:** The Amazon Nova models are currently trained to process and understand video content solely based on the visual frames. Audio tracks in videos are not processed or analyzed.
- **Multilingual image understanding:** The Amazon Nova models have limited understanding of multilingual images and video frames. They might struggle or hallucinate on simple tasks.
- **People identification:** The Amazon Nova models do not support the capability to identify or name individuals in images, documents, or videos. The models will not provide names or identities of people in visual content.
- **Small text in videos:** If the text in the image or video is too small, consider increasing relative size of the text in the video.
- **Spatial reasoning:** Amazon Nova 2 models have limited spatial reasoning capabilities. They may struggle with tasks that require precise understanding of object positions, distances, or spatial relationships in videos.
- **Inappropriate content:** The Nova models will not process inappropriate or explicit images that violate the Acceptable Use Policy.

- **Healthcare applications:** Due to the sensitive nature of these artifacts, even though Nova models could give general analysis on some healthcare images or videos, we do not recommend their use to interpret sensitive medical images like complex diagnostic scans. The response of Nova models should never be considered a substitute for professional medical advice.

## Document understanding

Amazon Nova's document understanding capability allows you to include entire documents (PDFs, Word files, spreadsheets and so on) as part of your prompt. This enables the model to analyze, summarize, extract information from, or answer questions about document content.

Amazon Nova 2 Lite can interpret both the text and visual elements (like charts or tables) within these documents. This enables use cases such as question-answering, summarization, and analysis of lengthy reports or scanned documents.

Key document understanding features include a very large context window (1M tokens) for long documents and the ability to handle multiple documents in one query.

## Supported document modalities and formats

Amazon Nova distinguishes between two types of document inputs:

- **Text-based documents**, like TXT, CSV, HTML, Markdown, or DOC files, are processed primarily for their textual content. Amazon Nova understands and extracts information from the text in these documents.
- **Media-based documents**, like PDF or DOCX files, may contain complex layouts, images, charts, or embedded graphics. For media-based documents, Amazon Nova employs vision-based understanding to interpret visual content—such as charts, tables, diagrams, or screenshots—alongside the document's text.

Supported file formats include common document types such as:

- Plain text and structured text files: CSV, TXT
- Spreadsheets: XLS, XLSX, HTML, Markdown
- Standard image formats (for images within documents): PNG, JPG, GIF, WebP
- Document formats: DOC, DOCX, PDF
- PDFs that contain image encodings, such as CYMK or SVG are not supported.

## Document size limits and usage guidelines

Constraint	Limit
Maximum number of documents	Up to 5 documents per request (applies to both direct upload and Amazon S3)
Text-based document size	Each text document must be equal to or less than 4.5 MB
Media-based document size	For PDF and DOCX files, there is no individual file size limit. When using direct upload, the combined size of all media documents must be less or equal to 25 MB. When using Amazon S3, the combined size of all media documents must be less or equal to 2 GB.
Unsupported PDF content	PDFs that contain CMYK color profiles or SVG images are not supported.

## Pricing

Amazon Nova uses token-based pricing: you pay for input tokens (everything you send, including attached documents) and output tokens (the model's response).

**Estimating tokens for PDFs:** For planning, assume a standard 8.5x11 inch PDF page ≈ 2,560 input tokens (this estimate covers both text and visual elements on a typical page).

## Examples: Using Nova's document understanding via API and S3

For an example of how to use it via API, refer to the Multimodal input using embedded asset - Converse API (non-streaming) example in the [Code library](#).

For an example of how to use it via S3, refer to the Multimodal input using S3 URI - Converse API (non-streaming) example in the [Code library](#).

# Speech-to-Speech (Amazon Nova 2 Sonic)

Amazon Nova 2 Sonic enables real-time conversational AI with speech input and output. The following section covers advanced capabilities for building interactive voice assistants, customer service automation, and conversational applications.

## Key features

Amazon Nova 2 Sonic provides the following capabilities:

- State-of-the-art streaming speech understanding with bidirectional streaming API that enables real-time, low-latency multi-turn conversations.
- Multilingual support with automatic language detection and switching. Expressive voices are offered, including both masculine-sounding and feminine-sounding voices, in the following languages:
  - English (US, UK, India, Australia)
  - French
  - Italian
  - German
  - Spanish
  - Portuguese
  - Hindi
- Polyglot voices that can speak any of the supported languages to enable a consistent user experience even when the user switches languages within the same session.
- Robustness to background noise for real world deployment scenarios.
- Robustness to different accents for supported languages.
- Natural, human-like conversational AI experiences with contextual richness across all supported languages.
- Adaptive speech response that dynamically adjusts delivery based on the prosody of the input speech.
- Intelligent turn-taking that detects when users finish speaking and when the assistant should respond, creating natural dialogue rhythm.
- Graceful handling of user interruptions without dropping conversational context.

- Knowledge grounding with enterprise data using Retrieval Augmented Generation (RAG).
- Function calling and agentic workflow support for building complex AI applications.
- Asynchronous tool handling that executes tool calls while maintaining conversation flow, allowing the assistant to continue speaking while tools process in the background.
- Cross-modal input support for both audio and text inputs within the same conversation, enabling flexible interaction patterns.

## Getting started with speech-to-speech

The following sections provide an example and step-by-step explanation of how to implement a simple, real-time audio streaming application using Amazon Nova 2 Sonic. This simplified version demonstrates the core functionality needed to create an audio conversation with the Amazon Nova 2 Sonic model.

You can access the following example in our [Nova samples GitHub repo](#).

### State the imports and configuration

This section imports necessary libraries and sets audio configuration parameters:

- `asyncio`: For asynchronous programming
- `base64`: For encoding and decoding audio data
- `pyaudio`: For audio capture and playback
- Amazon Bedrock SDK components for streaming
- Audio constants define the format of audio capture (16kHz sample rate, mono channel)

```
import os
import asyncio
import base64
import json
import uuid
import pyaudio
from aws_sdk_bedrock_runtime.client import BedrockRuntimeClient,
    InvokeModelWithBidirectionalStreamOperationInput
from aws_sdk_bedrock_runtime.models import
    InvokeModelWithBidirectionalStreamInputChunk, BidirectionalInputPayloadPart
```

```
from aws_sdk_bedrock_runtime.config import Config, HTTPAuthSchemeResolver,  
    SigV4AuthScheme  
from smithy_aws_core.identity import EnvironmentCredentialsResolver  
  
# Audio configuration  
INPUT_SAMPLE_RATE = 16000  
OUTPUT_SAMPLE_RATE = 24000  
CHANNELS = 1  
FORMAT = pyaudio.paInt16  
CHUNK_SIZE = 1024
```

## Define the SimpleNovaSonic class

The SimpleNovaSonic class is the main class that handles the Amazon Nova Sonic interaction:

- `model_id`: The Amazon Nova Sonic model ID (`amazon.nova-2-sonic-v1:0`)
- `region`: The AWS Region, the default is `us-east-1`
- Unique IDs for prompt and content tracking
- An asynchronous queue for audio playback

```
class SimpleNovaSonic:  
    def __init__(self, model_id='amazon.nova-2-sonic-v1:0', region='us-east-1'):  
        self.model_id = model_id  
        self.region = region  
        self.client = None  
        self.stream = None  
        self.response = None  
        self.is_active = False  
        self.prompt_name = str(uuid.uuid4())  
        self.content_name = str(uuid.uuid4())  
        self.audio_content_name = str(uuid.uuid4())  
        self.audio_queue = asyncio.Queue()  
        self.display_assistant_text = False
```

## Initialize the client

This method configures the Amazon Bedrock client with the following:

- The appropriate endpoint for the specified region
- Authentication information using environment variables for AWS credentials

- The SigV4 authentication scheme for the AWS API calls

```
def _initialize_client(self):
    """Initialize the Bedrock client."""
    config = Config(
        endpoint_uri=f"https://bedrock-runtime.{self.region}.amazonaws.com",
        region=self.region,
        aws_credentials_identity_resolver=EnvironmentCredentialsResolver(),
        auth_scheme_resolver=HTTPAuthSchemeResolver(),
        auth_schemes={"aws.auth#sigv4": SigV4AuthScheme(service="bedrock")})
    self.client = BedrockRuntimeClient(config=config)
```

## Handle events

This helper method sends JSON events to the bidirectional stream, which is used for all communication with the Amazon Nova Sonic model:

```
async def send_event(self, event_json):
    """Send an event to the stream."""
    event = InvokeModelWithBidirectionalStreamInputChunk(
        value=BidirectionalInputPayloadPart(bytes_=event_json.encode('utf-8')))
    await self.stream.input_stream.send(event)
```

## Start the session

This method initiates the session and setups the remaining events to start audio streaming. These events need to be sent in the same order.

```
async def start_session(self):
    """Start a new session with Nova Sonic."""
    if not self.client:
        self._initialize_client()

    # Initialize the stream
    self.stream = await self.client.invoke_model_with_bidirectional_stream(
        InvokeModelWithBidirectionalStreamOperationInput(model_id=self.model_id))
    self.is_active = True
```

```
# Send session start event
session_start = '''
{
    "event": {
        "sessionStart": {
            "inferenceConfiguration": {
                "maxTokens": 1024,
                "topP": 0.9,
                "temperature": 0.7
            },
            "turnDetectionConfiguration": {
                "endpointingSensitivity": "HIGH"
            }
        }
    }
}
'''

await self.send_event(session_start)

# Send prompt start event
prompt_start = f'''
{{{
    "event": {{
        "promptStart": {{
            "promptName": "{self.prompt_name}",
            "textOutputConfiguration": {{
                "mediaType": "text/plain"
            }},
            "audioOutputConfiguration": {{
                "mediaType": "audio/lpcm",
                "sampleRateHertz": 24000,
                "sampleSizeBits": 16,
                "channelCount": 1,
                "voiceId": "matthew",
                "encoding": "base64",
                "audioType": "SPEECH"
            }}
        }}
    }}
}}
'''

await self.send_event(prompt_start)

# Send system prompt
```

```
text_content_start = f'''  
{{  
    "event": {{  
        "contentStart": {{  
            "promptName": "{self.prompt_name}",  
            "contentName": "{self.content_name}",  
            "type": "TEXT",  
            "interactive": true,  
            "role": "SYSTEM",  
            "textInputConfiguration": {{  
                "mediaType": "text/plain"  
            }}  
        }}  
    }}  
}}  
...  
await self.send_event(text_content_start)  
  
system_prompt = "You are a friendly assistant. The user and you will engage in  
a spoken dialog " \  
    "exchanging the transcripts of a natural real-time conversation. Keep your  
responses short, " \  
    "generally two or three sentences for chatty scenarios."  
  
text_input = f'''  
{{  
    "event": {{  
        "textInput": {{  
            "promptName": "{self.prompt_name}",  
            "contentName": "{self.content_name}",  
            "content": "{system_prompt}"  
        }}  
    }}  
}}  
...  
await self.send_event(text_input)  
  
text_content_end = f'''  
{{  
    "event": {{  
        "contentEnd": {{  
            "promptName": "{self.prompt_name}",  
            "contentName": "{self.content_name}"  
        }}  
    }}  
}}  
...  
await self.send_event(text_content_end)
```

```
        }
    }
}
```
await self.send_event(text_content_end)

# Start processing responses
self.response = asyncio.create_task(self._process_responses())
```

## Handle audio input

These methods handle the audio input lifecycle:

- `start_audio_input`: Configures and starts the audio input stream
- `send_audio_chunk`: Encodes and sends audio chunks to the model
- `end_audio_input`: Properly closes the audio input stream

```
async def start_audio_input(self):
    """Start audio input stream."""
    audio_content_start = f'''
    {{
        "event": {{
            "contentStart": {{
                "promptName": "{self.prompt_name}",
                "contentName": "{self.audio_content_name}",
                "type": "AUDIO",
                "interactive": true,
                "role": "USER",
                "audioInputConfiguration": {{
                    "mediaType": "audio/lpcm",
                    "sampleRateHertz": 16000,
                    "sampleSizeBits": 16,
                    "channelCount": 1,
                    "audioType": "SPEECH",
                    "encoding": "base64"
                }}
            }}
        }}
    }}
```
await self.send_event(audio_content_start)
```

```
async def send_audio_chunk(self, audio_bytes):
    """Send an audio chunk to the stream."""
    if not self.is_active:
        return

    blob = base64.b64encode(audio_bytes)
    audio_event = f'''
    {{
        "event": {{
            "audioInput": {{
                "promptName": "{self.prompt_name}",
                "contentName": "{self.audio_content_name}",
                "content": "{blob.decode('utf-8')}"
            }}
        }}
    }}
    '''
    await self.send_event(audio_event)

async def end_audio_input(self):
    """End audio input stream."""
    audio_content_end = f'''
    {{
        "event": {{
            "contentEnd": {{
                "promptName": "{self.prompt_name}",
                "contentName": "{self.audio_content_name}"
            }}
        }}
    }}
    '''
    await self.send_event(audio_content_end)
```

## End the session

This method properly closes the session by:

- Sending a promptEnd event
- Sending a sessionEnd event
- Closing the input stream

```
async def end_session(self):
    """End the session."""
    if not self.is_active:
        return

    prompt_end = f'''
{{{
    "event": {{
        "promptEnd": {{
            "promptName": "{self.prompt_name}"
        }}
    }}
}}
'''

    await self.send_event(prompt_end)

    session_end = '''
{
    "event": {
        "sessionEnd": {}
    }
}
'''

    await self.send_event(session_end)
    # close the stream
    await self.stream.input_stream.close()
```

## Handle responses

This method continuously processes responses from the model and does the following:

- Waits for output from the stream.
- Parses the JSON response.
- Handles text output by printing to the console with automatic speech recognition and transcription.
- Handles audio output by decoding and queuing for playback.

```
async def _process_responses(self):
    """Process responses from the stream."""
    try:
```

```
while self.is_active:
    output = await self.stream.await_output()
    result = await output[1].receive()

    if result.value and result.value.bytes_:
        response_data = result.value.bytes_.decode('utf-8')
        json_data = json.loads(response_data)

        if 'event' in json_data:
            # Handle content start event
            if 'contentStart' in json_data['event']:
                content_start = json_data['event']['contentStart']
                # set role
                self.role = content_start['role']
                # Check for speculative content
                if 'additionalModelFields' in content_start:
                    additional_fields =
                        json.loads(content_start['additionalModelFields']))
                    if additional_fields.get('generationStage') ==
'SPECULATIVE':
                        self.display_assistant_text = True
                    else:
                        self.display_assistant_text = False

            # Handle text output event
            elif 'textOutput' in json_data['event']:
                text = json_data['event']['textOutput']['content']

                if (self.role == "ASSISTANT" and
self.display_assistant_text):
                    print(f"Assistant: {text}")
                elif self.role == "USER":
                    print(f"User: {text}")

            # Handle audio output
            elif 'audioOutput' in json_data['event']:
                audio_content = json_data['event']['audioOutput']
                ['content']
                audio_bytes = base64.b64decode(audio_content)
                await self.audio_queue.put(audio_bytes)
except Exception as e:
    print(f"Error processing responses: {e}")
```

## Playback audio

This method will perform the following tasks:

- Initialize a PyAudio input stream
- Continuously retrieves audio data from the queue
- Plays the audio through the speakers
- Properly cleans up resources when done

```
async def play_audio(self):
    """Play audio responses."""
    p = pyaudio.PyAudio()
    stream = p.open(
        format=FORMAT,
        channels=CHANNELS,
        rate=OUTPUT_SAMPLE_RATE,
        output=True
    )
```

try:

```
    while self.is_active:
        audio_data = await self.audio_queue.get()
        stream.write(audio_data)
    except Exception as e:
        print(f"Error playing audio: {e}")
    finally:
        stream.stop_stream()
        stream.close()
        p.terminate()
```

## Capture audio

This method will perform the following tasks:

- Initializes a PyAudio output stream
- Starts the audio input session

- Continuously captures audio chunks from the microphone
- Sends each chunk to the Amazon Nova Sonic model
- Properly cleans up resources when done

```
async def capture_audio(self):  
    """Capture audio from microphone and send to Nova Sonic."""  
    p = pyaudio.PyAudio()  
    stream = p.open(  
        format=FORMAT,  
        channels=CHANNELS,  
        rate=INPUT_SAMPLE_RATE,  
        input=True,  
        frames_per_buffer=CHUNK_SIZE  
    )  
  
    print("Starting audio capture. Speak into your microphone...")  
    print("Press Enter to stop...")  
  
    await self.start_audio_input()
```

try:

```
    while self.is_active:  
        audio_data = stream.read(CHUNK_SIZE, exception_on_overflow=False)  
        await self.send_audio_chunk(audio_data)  
        await asyncio.sleep(0.01)  
    except Exception as e:  
        print(f"Error capturing audio: {e}")  
    finally:  
        stream.stop_stream()  
        stream.close()  
        p.terminate()  
        print("Audio capture stopped.")  
        await self.end_audio_input()
```

## Run the main function

The main function orchestrates the entire process by performing the following:

- Creates an Amazon Nova 2 Sonic client
- Starts the session
- Creates concurrent tasks for audio playback and capture
- Waits for the user to press **Enter** to stop
- Properly ends the session and cleans up tasks

```
async def main():
    # Create Nova Sonic client
    nova_client = SimpleNovaSonic()

    # Start session
    await nova_client.start_session()

    # Start audio playback task
    playback_task = asyncio.create_task(nova_client.play_audio())

    # Start audio capture task
    capture_task = asyncio.create_task(nova_client.capture_audio())

    # Wait for user to press Enter to stop
    await asyncio.get_event_loop().run_in_executor(None, input)

    # End session
    nova_client.is_active = False

    # First cancel the tasks
    tasks = []
    if not playback_task.done():
        tasks.append(playback_task)
    if not capture_task.done():
        tasks.append(capture_task)
    for task in tasks:
        task.cancel()
    if tasks:
        await asyncio.gather(*tasks, return_exceptions=True)

    # cancel the response task
    if nova_client.response and not nova_client.response.done():
        nova_client.response.cancel()

    await nova_client.end_session()
```

```
print("Session ended")

if __name__ == "__main__":
    # Set AWS credentials if not using environment variables
    # os.environ['AWS_ACCESS_KEY_ID'] = "your-access-key"
    # os.environ['AWS_SECRET_ACCESS_KEY'] = "your-secret-key"
    # os.environ['AWS_DEFAULT_REGION'] = "us-east-1"

    asyncio.run(main())
```

## Code examples

These code examples help you quickly get started with Amazon Nova 2 Sonic. You can access the complete list of examples in the [Amazon Nova Sonic GitHub samples](#) page.

### Getting started examples

For simple examples designed to get you started using Amazon Nova 2 Sonic, refer to the following implementations:

- [Basic Amazon Nova 2 Sonic implementation \(Python\)](#): A basic implementation that demonstrates how events are structured in the bidirectional streaming API. This version does not support barge-in functionality (interrupting the assistant while it is speaking) and does not implement true bidirectional communication.
- [Full featured Amazon Nova 2 Sonic implementation \(Python\)](#): The full-featured implementation with real bidirectional communication and barge-in support. This allows for more natural conversations where users can interrupt the assistant while it is speaking, similar to human conversations.
- [Amazon Nova 2 Sonic with tool use \(Python\)](#): An advanced implementation that extends the bidirectional communication capabilities with tool use examples. This version demonstrates how Amazon Nova 2 Sonic can interact with external tools and APIs to provide enhanced functionality.
- [Nova Sonic with text and mixed Input \(Python\)](#): Example implementation to showcase how Amazon Nova 2 Sonic can have text as an input.
- [Java WebSocket implementation \(Java\)](#): This example implements a bidirectional WebSocket-based audio streaming application that integrates with Amazon Nova 2 Sonic for real-time speech-to-speech conversation using Java.

- [NodeJS Websocket implementation \(NodeJS\)](#): This example implements a bidirectional WebSocket-based audio streaming application that integrates with Amazon Nova 2 Sonic for real-time speech-to-speech conversation using NodeJS.
- [NodeJS Websocket implementation \(C#\)](#): This example implements a bidirectional WebSocket-based audio streaming application that integrates with Amazon Nova 2 Sonic for real-time speech-to-speech conversation using .NET.

## Advanced use cases

For advanced examples demonstrating more complex use cases, refer to the following implementations:

- [Amazon Bedrock Knowledge Base implementation \(NodeJS\)](#): This example demonstrates how to build an intelligent conversational application by integrating Amazon Nova 2 Sonic with Amazon Bedrock Knowledge Base using NodeJS.
- [Chat History Management \(Python\)](#): This example includes a chat history logging system that captures and preserves all interactions between the user and Amazon Nova 2 Sonic using Python.
- [Hotel Reservation Cancellation \(NodeJS\)](#): This example demonstrates a practical customer service use case for Amazon Nova 2 Sonic, implementing a hotel reservation cancellation system using NodeJS.
- [LangChain Knowledge Base integration \(Python\)](#): This implementation demonstrates how to integrate Amazon Nova 2 Sonic speech-to-speech capabilities with a LangChain-powered knowledge base for enhanced conversational experiences using Python.
- [Conversation Resumption \(NodeJS\)](#): This example demonstrates how to implement conversation resumption capabilities with Amazon Nova 2 Sonic. Using a hotel reservation cancellation scenario as the context, the application shows how to maintain conversation state across sessions, allowing users to seamlessly continue interactions that were previously interrupted using NodeJS.
- [Nova 2 Sonic Speaks First \(NodeJS\)](#): This example demonstrates how Amazon Nova 2 Sonic can initiate conversations proactively.
- [Session Continuation \(Python\)](#): This example demonstrates how to enable unlimited conversation length with Amazon Nova 2 Sonic by implementing seamless session transitions. The application automatically creates and switches to new sessions in the background, allowing conversations to continue indefinitely without interruption or context loss.

## Hands-on workshop

A hands-on workshop is available that guides you through building a voice chat application using Amazon Nova 2 Sonic with a bidirectional streaming interface. You can [access the workshop](#) and find the [complete code examples](#).

## Voice conversation prompts

Nova 2 introduces **Speech Prompts** – a specialized prompting capability designed to control speech-specific transcription formatting for Hindi. Speech prompts work alongside your system prompt but serve a distinct purpose:

- **System Prompt:** Controls your assistant's behavior, personality, and response style
- **Speech Prompt:** Controls transcription formatting for Hindi code-switching (Latin/Devanagari/mixed scripts)

## Important Guidelines

**Speech prompts are pre-configured and should be used exactly as documented.** They are designed for specific transcription formatting needs and should not be modified or customized, as changes may cause unexpected behavior.

### When to use Speech Prompts:

- You need to control script output for Hindi code-switching (Latin/Devanagari/mixed)

### When NOT to use Speech Prompts:

- For general instructions or assistant behavior (use system prompt instead)
- If you're not working with Hindi transcription formatting
- If the specific formatting need doesn't apply to your use case

**Best Practice:** Only include a speech prompt if you specifically need Hindi transcription formatting. All other instructions – including language preferences, response style, verbosity, and reasoning – should go in your system prompt.

**Important:** Speech prompts must be sent **after** the system prompt to the model.

## Recommended Baseline System Prompt for Voice

You are a warm, professional, and helpful AI assistant. Give accurate answers that sound natural, direct, and human. Start by answering the user's question clearly in 1-2 sentences. Then, expand only enough to make the answer understandable, staying within 3-5 short sentences total. Avoid sounding like a lecture or essay.

## Speech Prompt Configuration

### Code Switching

**Note:** This feature currently applies to Hindi language only.

Choose one of the following prompts based on your desired output script:

#### For Latin script output (Romanized Hindi):

If the input audio/speech contains hindi, then the transcription and response should be in All Latin script (romanized Hindi).

#### For Devanagari script output:

If the input audio/speech contains hindi, then the transcription and response should be in All Devanagari script (Hindi).

#### For mixed script output (natural code-switching):

If the input audio/speech contains hindi, then the transcription and response can mix Latin and Devanagari scripts naturally for code-switching.

## System Prompt Configuration

### Controlling Response Verbosity

#### Concise, conversational responses:

You are a warm, professional, and helpful AI assistant. Give accurate answers that sound natural, direct, and human. Start by answering the user's question clearly in 1-2 sentences. Then, expand only enough to make the answer understandable, staying within 3-5 short sentences total. Avoid sounding like a lecture or essay.

## Detailed, thorough responses:

You are a warm, professional, and helpful AI assistant. Give accurate, complete answers that sound warm, direct, and human. Answer the question directly in the first 1-2 sentences. If the question has parts or asks what/why/how, address each with a brief definition or main idea plus 2-3 key facts or steps. Offer practical, actionable advice. Keep a confident, kind, conversational tone; never robotic or theatrical. Be thorough; add examples or context only when helpful. Prefer accuracy and safety over speculation; if unsure, say so and suggest what to check.

## Language Mirroring

Nova can recognize and respond in the language the user speaks. Use this prompt to maintain language consistency:

### CRITICAL LANGUAGE MIRRORING RULES:

- Always reply in the language spoken. DO NOT mix with English. However, if the user talks in English, reply in English.
- Please respond in the language the user is talking to you in. If you have a question or suggestion, ask it in the language the user is talking in. I want to ensure that our communication remains in the same language as the user.

## Gender Agreement for Gendered Languages

Some languages require gender agreement in verbs, adjectives, or pronouns when the assistant describes itself. For these languages, specify the assistant's gender in your system prompt to match your selected voice.

**Languages affected:** Hindi, Portuguese, French, Italian, Spanish, Russian, Polish

### When gender agreement matters:

- **Hindi:** Always needed - verbs conjugate based on speaker's gender in first person
- **Portuguese/French:** Needed when using past participles or adjectives (such as, "I am tired" - "Estou cansada/cansado")
- **Italian/Spanish:** Needed when using adjectives to describe oneself (such as, "I am happy" - "Sono contenta/contento")

### Implementation:

Include the appropriate gender identifier at the start of your system prompt based on your voice selection:

### For feminine-sounding voices (**kiara, carolina, ambre, beatrice, lupe, tiffany**):

You are a warm, professional, and helpful female AI assistant.

### For masculine-sounding voices (**arjun, leo, florian, lorenzo, carlos, matthew**):

You are a warm, professional, and helpful male AI assistant.

### Examples:

#### Hindi with feminine voice (**kiara**):

You are a warm, professional, and helpful female AI assistant.

Result: "म अच्छी हूँ" (main achchhi hoon) vs "म अच्छा हूँ" (main achchha hoon)

#### Italian with masculine voice (**lorenzo**):

You are a warm, professional, and helpful male AI assistant.

Result: "Sono contento" vs "Sono contenta"

## Chain of thought for Speech: Constitutional Reasoning

Use this prompt when you want the model to show its reasoning for complex problems:

You are a friendly assistant. The user will give you a problem. Explain your reasoning following the guidelines given in CONSTITUTION - REASONING, and summarize your decision at the end of your response, in one sentence.

## CONSTITUTION - REASONING

1. For simple questions including simple calculations or contextual tasks: Give the answer directly. No explanation is necessary, although you can offer to provide more information if the user requests it.
2. When faced with complex problems or decisions, think through the steps systematically before providing your answer. Break down your reasoning process when it would help user understanding.

3. For subjective matters or comparisons: explain your thought process step-by-step.

**Note:** If you don't want the model to go through reasoning for every request, you can add a couple of shot examples to the prompt (see examples below).

You are a warm, professional, and helpful AI assistant. You converse in fluid and conversational English. Give accurate, complete answers that sound warm, direct, and human. Answer the question directly in the first 1-2 sentences. Keep a confident, kind, conversational tone; never robotic or theatrical. Avoid formatted lists or numbering and keep your output as a spoken transcript. Be concise but thorough; add examples or context only when helpful. Prefer accuracy and safety over speculation; if unsure, say so and suggest what to check. The user will give you a problem. Explain your reasoning following the guidelines given in CONSTITUTION - REASONING, and summarize your decision at the end of your response in one sentence.

#### ## CONSTITUTION - REASONING

1. When faced with complex problems or decisions, think through the steps systematically before providing your answer. Break down your reasoning process when it would help user understanding.
2. For subjective matters or comparisons: explain your thought process step-by-step.
3. For simple questions including simple calculations or contextual tasks: Give the answer directly. No explanation is necessary, although you can offer to provide more information if the user requests it.

#### EXAMPLES

User: What is 7 + 5?

Assistant: 12.

User: What is the capital of India?

Assistant: Delhi is the capital of India.

User: I have a \$1,000 budget for a trip. Here are my costs... Can I afford it? Please explain your reasoning.

Assistant: (step-by-step breakdown + one-sentence conclusion)

## Overuse of suggested phrases

**Nova Sonic 2 is more sensitive to phrase suggestions than Sonic 1.** This increased sensitivity isn't inherently good or bad—it depends on your use case. If you want consistent, predictable phrasing, this can be beneficial. However, if you want more natural variation, explicit phrase lists can lead to overuse.

If you include prompts with explicit lists of phrases, the model will use them very frequently:

### Example 1 - Emphasis phrases:

Instead of using bold or italics, emphasize important information by using phrases like "The key thing to remember is," "What's really important here is," or "I want to highlight that."

### Example 2 - Conversational fillers:

Include natural speech elements like "Well," "You know," "Actually," "I mean," or "By the way" at appropriate moments to create an authentic, casual conversation flow.

### Recommendation:

- **If you want consistent phrasing:** Explicit phrase lists work well in Sonic 2 for creating predictable, on-brand responses.
- **If you want natural variation:** Avoid providing explicit lists of phrases. Instead, use general guidance like "sound natural and conversational" or provide one-shot examples.

### Better approach - Use one-shot examples:

Instead of listing phrases, provide 1-2 examples demonstrating the desired tone and style:

#### Natural, helpful tone

You are a warm, professional, and helpful AI assistant. Sound natural and conversational in your responses.

Example:

User: How do I reset my password?

Assistant: You can reset your password by clicking the "Forgot Password" link on the login page. You'll get an email with instructions to create a new one. The whole process usually takes just a couple of minutes.

#### Concise and direct

You are a helpful AI assistant. Provide clear, direct answers without unnecessary elaboration.

**Example:**

User: What's the weather like today?

Assistant: It's 72 degrees and sunny with a light breeze. Perfect day to be outside.

## Professional with empathy

You are a professional and empathetic AI assistant. Acknowledge the user's situation while providing practical solutions.

**Example:**

User: I'm frustrated because my order hasn't arrived yet.

Assistant: I understand how frustrating that must be, especially when you're waiting for something important. Let me check your order status right now. Can you provide your order number?

## Technical but accessible

You are a knowledgeable AI assistant who explains technical concepts in accessible language.

**Example:**

User: What is machine learning?

Assistant: Machine learning is when computers learn from examples rather than following strict rules. Think of it like teaching a child to recognize dogs—after seeing many dogs, they start recognizing new ones on their own. The computer does something similar with data.

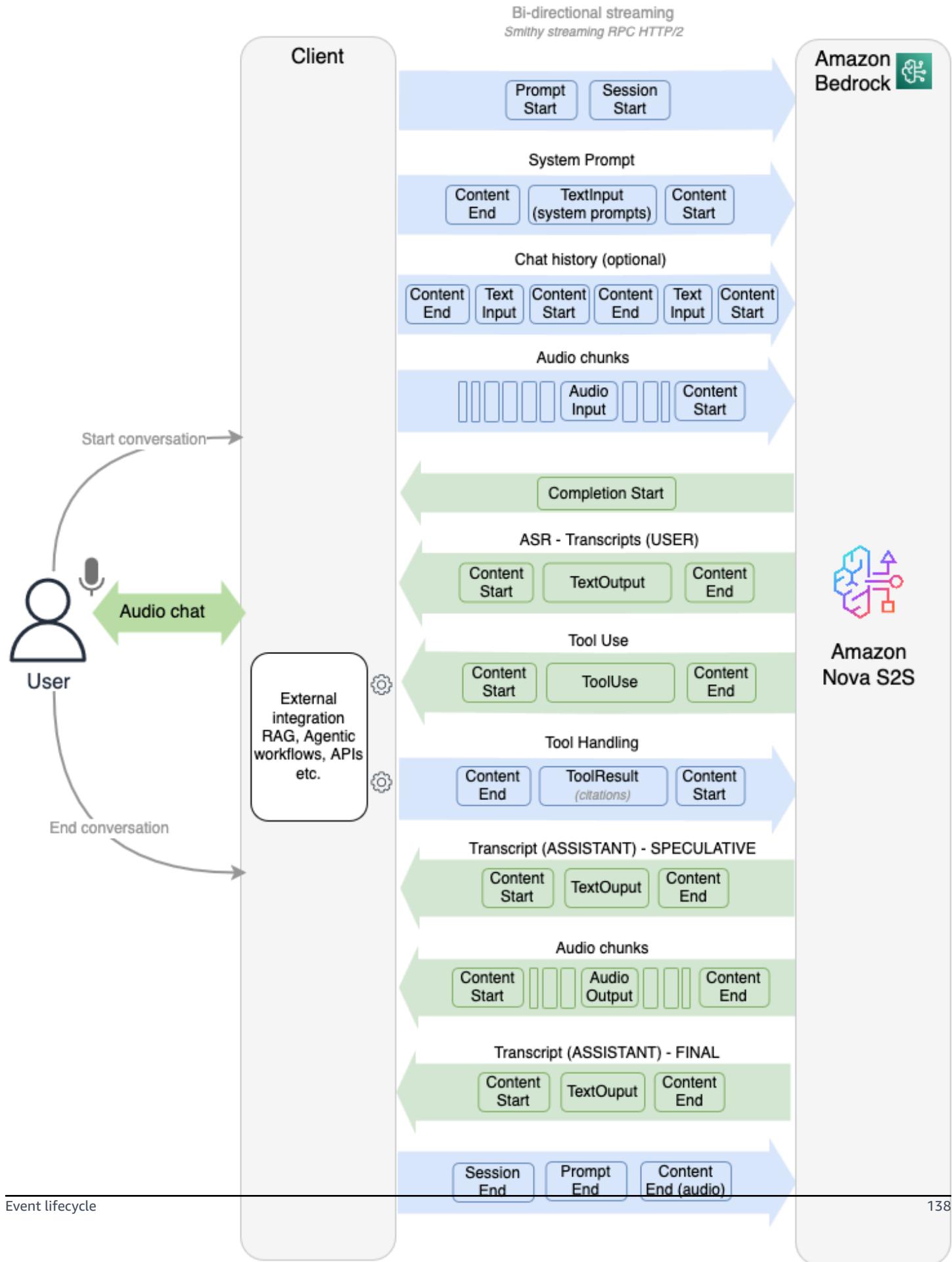
This approach shows the model the desired behavior without triggering repetitive phrase patterns, while still maintaining control over tone and style.

## Core concepts

Amazon Nova 2 Sonic uses a bidirectional streaming architecture with structured events for real-time conversational AI. Understanding these core concepts is essential for building effective voice applications.

## Event lifecycle

The following diagram illustrates the complete bi-directional streaming event lifecycle:



The bidirectional streaming event lifecycle follows a structured pattern from session initialization through conversation completion. Each conversation involves input events (from your application) and output events (from Amazon Nova 2 Sonic) that work together to create natural voice interactions.

## Event flow sequence

A typical conversation follows this event sequence:

1. **Session Start** - Initialize the conversation session
2. **System Prompt** - Send system instructions
3. **Chat History** (optional) - Provide conversation context
4. **Audio Chunks** - Stream user audio input
5. **Completion Start** - AI begins processing
6. **ASR Transcripts** (USER) - User speech transcription
7. **Tool Use** (optional) - AI requests tool execution
8. **Tool Handling** (optional) - Process and return tool results
9. **Transcript** (ASSISTANT) - SPECULATIVE - Preliminary AI response
10. **Audio Chunks** - Stream AI audio output
11. **Transcript** (ASSISTANT) - FINAL - Final AI response transcript
12. **Content End Audio** - Marks the end of audio content
13. **Prompt End** - Indicates the completion of the prompt processing
14. **Session End** - Close the conversation

## Handling input events with the bidirectional API

The bidirectional Stream API uses an event-driven architecture with structured input and output events. Understanding the correct event ordering is crucial for implementing successful conversational applications and maintaining the proper conversation state throughout interactions.

### Overview

The Nova Sonic conversation follows a structured event sequence. You begin by sending a `sessionStart` event that contains the inference configuration parameters, such as temperature and token limits. Next, you send `promptStart` to define the audio output format and tool

configurations, assigning a unique `promptName` identifier that must be included in all subsequent events.

For each interaction type (system prompt, audio, and so on), you follow a three-part pattern: use `contentStart` to define the content type and the role of the content (SYSTEM, USER, ASSISTANT, TOOL, SYSTEM\_SPEECH), then provide the actual content event, and finish with `contentEnd` to close that segment. The `contentStart` event specifies whether you're sending tool results, streaming audio, or a system prompt. The `contentStart` event includes a unique `contentName` identifier.

## Conversation History

A conversation history can be included only once, after the system prompt and before audio streaming begins. It follows the same `contentStart/textInput/contentEnd` pattern. The USER and ASSISTANT roles must be defined in the `contentStart` event for each historical message. This provides essential context for the current conversation but must be completed before any new user input begins.

## Audio Streaming

Audio streaming operates with continuous microphone sampling. After sending an initial `contentStart`, audio frames (approximately 32ms each) are captured directly from the microphone and immediately sent as `audioInput` events using the same `contentName`. These audio samples should be streamed in real-time as they're captured, maintaining the natural microphone sampling cadence throughout the conversation. All audio frames share a single content container until the conversation ends and it is explicitly closed.

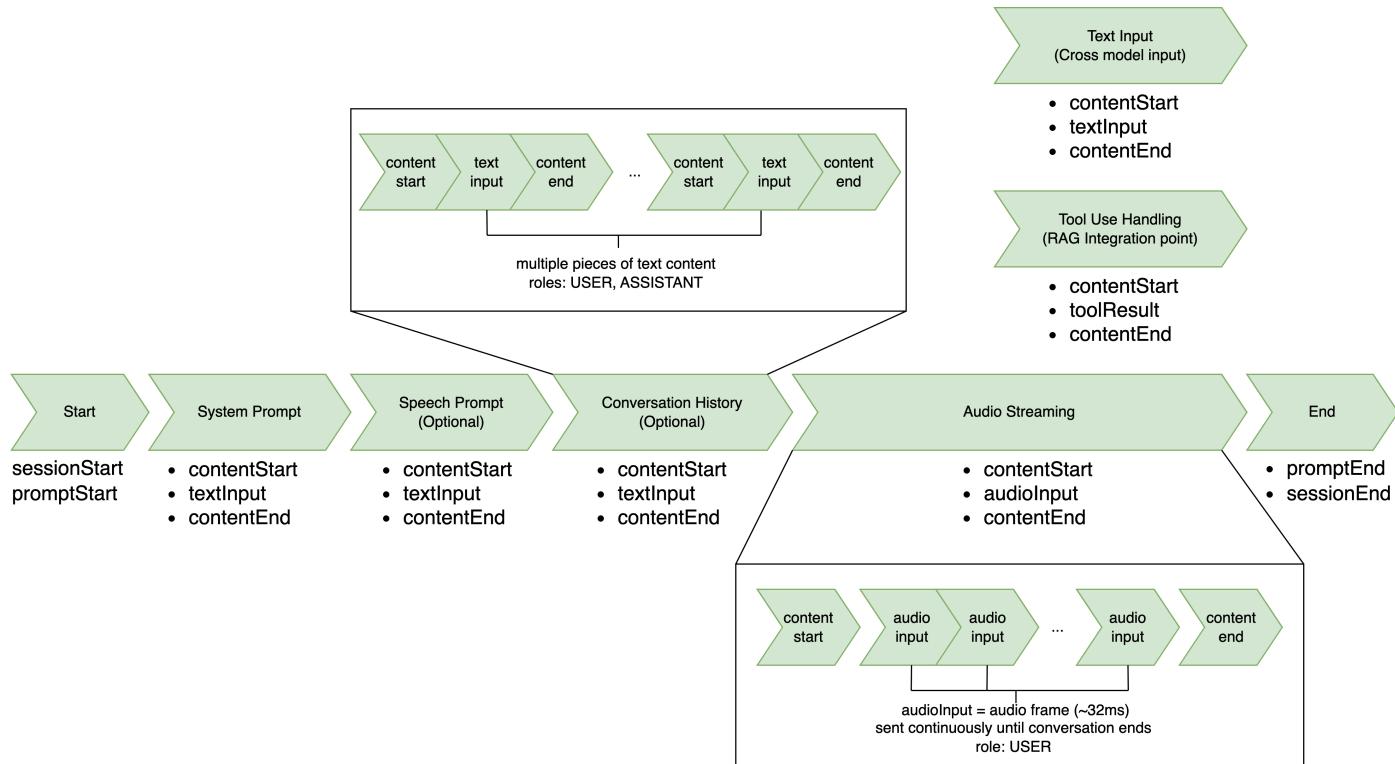
## Closing the Session

After the conversation ends or needs to be terminated, it's essential to properly close all open streams and end the session in the correct sequence. To properly end a session and avoid resource leaks, you must follow a specific closing sequence:

- Close any open audio streams with the `contentEnd` event.
- Send a `promptEnd` event that references the original `promptName`.
- Send the `sessionEnd` event.

Skipping any of these closing events can result in incomplete conversations or orphaned resources.

These identifiers create a hierarchical structure: the promptName ties all conversation events together, while each contentName marks the boundaries of specific content blocks. This hierarchy ensures that model maintains proper context throughout the interaction.



## Input Event Flow

The structure of the input event flow is provided in this section.

### 1. RequestStartEvent (Session Start)

The session start event initializes the conversation with inference configuration and turn detection settings.

#### Inference Configuration:

- maxTokens: Maximum number of tokens to generate in the response
- topP: Nucleus sampling parameter (0.0 to 1.0) for controlling randomness
- temperature: Controls randomness in generation (0.0 to 1.0)

**Turn Detection Configuration:** The endpointingSensitivity parameter controls how quickly Nova Sonic detects when a user has finished speaking:

- **HIGH:** Detects pauses quickly, enabling faster responses but may cut off slower speakers
- **MEDIUM:** Balanced sensitivity for most conversational scenarios (recommended default)
- **LOW:** Waits longer before detecting end of speech, better for thoughtful or hesitant speakers

```
{  
    "event": {  
        "sessionStart": {  
            "inferenceConfiguration": {  
                "maxTokens": "int",  
                "topP": "float",  
                "temperature": "float"  
            },  
            "turnDetectionConfiguration": {  
                "endpointingSensitivity": "HIGH" | "MEDIUM" | "LOW"  
            }  
        }  
    }  
}
```

## Example:

```
{  
    "event": {  
        "sessionStart": {  
            "inferenceConfiguration": {  
                "maxTokens": 2048,  
                "topP": 0.9,  
                "temperature": 0.7  
            },  
            "turnDetectionConfiguration": {  
                "endpointingSensitivity": "MEDIUM"  
            }  
        }  
    }  
}
```

## 2. PromptStartEvent

The prompt start event defines the conversation configuration including output formats, voice selection, and available tools.

For a list of available voice IDs, refer to [Language support and multilingual capabilities](#)

```
{  
    "event": {  
        "promptStart": {  
            "promptName": "string", // unique identifier same across all events i.e.  
            UUID  
            "textOutputConfiguration": {  
                "mediaType": "text/plain"  
            },  
            "audioOutputConfiguration": {  
                "mediaType": "audio/lpcm",  
                "sampleRateHertz": 8000 | 16000 | 24000,  
                "sampleSizeBits": 16,  
                "channelCount": 1,  
                "voiceId": "matthew" | "tiffany" | "amy" | "olivia" | "lupe" | "carlos"  
                | "ambre" | "florian" | "lennart" | "beatrice" | "lorenzo" |  
                "tina" | "carolina" | "leo" | "kiara" | "arjun",  
                "encoding": "base64",  
                "audioType": "SPEECH"  
            },  
            "toolUseOutputConfiguration": {  
                "mediaType": "application/json"  
            },  
            "toolConfiguration": {  
                "tools": [  
                    {  
                        "toolSpec": {  
                            "name": "string",  
                            "description": "string",  
                            "inputSchema": {  
                                "json": "{}"  
                            }  
                        }  
                    }  
                ]  
            }  
        }  
    }  
}
```

### 3. InputContentStartEvent

#### Text

The text content start event is used for system prompts, conversation history, and cross-modal text input.

#### Interactive Parameter:

- **true**: Enables cross-modal input, allowing text messages during an active voice session
- **false**: Standard text input for system prompts and conversation history

#### Role Types:

- **SYSTEM**: System instructions and prompts
- **USER**: User messages in conversation history or cross-modal input
- **ASSISTANT**: Assistant responses in conversation history
- **SYSTEM\_SPEECH**: System prompts that should be spoken aloud by the assistant

```
{  
  "event": {  
    "contentStart": {  
      "promptName": "string", // same unique identifier from promptStart event  
      "contentName": "string", // unique identifier for the content block  
      "type": "TEXT",  
      "interactive": "boolean", // true for cross-modal input  
      "role": "SYSTEM" | "USER" | "ASSISTANT" | "TOOL" | "SYSTEM_SPEECH",  
      "textInputConfiguration": {  
        "mediaType": "text/plain"  
      }  
    }  
  }  
}
```

#### Example - System Prompt:

```
{  
  "event": {
```

```
"contentStart": {  
    "promptName": "conv-12345",  
    "contentName": "system-prompt-1",  
    "type": "TEXT",  
    "interactive": false,  
    "role": "SYSTEM",  
    "textInputConfiguration": {  
        "mediaType": "text/plain"  
    }  
}  
}  
}
```

## Example - Cross-modal Input:

```
{  
    "event": {  
        "contentStart": {  
            "promptName": "conv-12345",  
            "contentName": "user-text-1",  
            "type": "TEXT",  
            "interactive": true,  
            "role": "USER",  
            "textInputConfiguration": {  
                "mediaType": "text/plain"  
            }  
        }  
    }  
}
```

## Audio

```
{  
    "event": {  
        "contentStart": {  
            "promptName": "string", // same unique identifier from promptStart event  
            "contentName": "string", // unique identifier for the content block  
            "type": "AUDIO",  
            "interactive": true,  
            "role": "USER",  
            "audioInputConfiguration": {  
                "mediaType": "audio/lpcm",  
                "sampleRateHertz": 8000 | 16000 | 24000,  
            }  
        }  
    }  
}
```

```
        "sampleSizeBits": 16,
        "channelCount": 1,
        "audioType": "SPEECH",
        "encoding": "base64"
    }
}
}
}
```

## Tool

```
{
  "event": {
    "contentStart": {
      "promptName": "string", // same unique identifier from promptStart event
      "contentName": "string", // unique identifier for the content block
      "interactive": false,
      "type": "TOOL",
      "role": "TOOL",
      "toolResultInputConfiguration": {
        "toolUseId": "string", // existing tool use id
        "type": "TEXT",
        "textInputConfiguration": {
          "mediaType": "text/plain"
        }
      }
    }
  }
}
```

## 4. TextInputContent

```
{
  "event": {
    "textInput": {
      "promptName": "string", // same unique identifier from promptStart event
      "contentName": "string", // unique identifier for the content block
      "content": "string"
    }
  }
}
```

## 5. AudioInputContent

```
{  
  "event": {  
    "audioInput": {  
      "promptName": "string", // same unique identifier from promptStart event  
      "contentName": "string", // same unique identifier from its contentStart  
      "content": "base64EncodedAudioData"  
    }  
  }  
}
```

## 6. ToolResultContentEvent

```
"event": {  
  "toolResult": {  
    "promptName": "string", // same unique identifier from promptStart event  
    "contentName": "string", // same unique identifier from its contentStart  
    "content": "{\"key\": \"value\"}" // stringified JSON object as a tool result  
  }  
}
```

## 7. InputContentEndEvent

```
{  
  "event": {  
    "contentEnd": {  
      "promptName": "string", // same unique identifier from promptStart event  
      "contentName": "string" // same unique identifier from its contentStart  
    }  
  }  
}
```

## 8. PromptEndEvent

```
{  
  "event": {  
    "promptEnd": {  
      "promptName": "string" // same unique identifier from promptStart event  
    }  
  }  
}
```

## 9. RequestEndEvent

```
{  
  "event": {  
    "sessionEnd": {}  
  }  
}
```

# Handling output events with the bidirectional API

When the Amazon Nova Sonic model responds, it follows a structured event sequence. The flow begins with a `completionStart` event that contains unique identifiers like `sessionId`, `promptName`, and `completionId`. These identifiers are consistent throughout the response cycle and unite all subsequent response events.

## Overview

Each response type follows a consistent three-part pattern: `contentStart` defines the content type and format, the actual content event, and `contentEnd` closes that segment. The response typically includes multiple content blocks in sequence: automatic speech recognition (ASR) transcription (what the user said), optional tool use (when external information is needed), text response (what the model plans to say), and audio response (the spoken output).

## Response Content Types

### ASR Transcription

The ASR transcription appears first, delivering the model's understanding of the user's speech with `role: "USER"` and `"additionalModelFields": "{\"generationStage\": \"FINAL\"}"` in the `contentStart`.

### Tool Use

When the model needs external data, it sends tool-related events with specific tool names and parameters.

### Text Response

The text response provides a preview of the planned speech with `role: "ASSISTANT"` and `"additionalModelFields": "{\"generationStage\": \"SPECULATIVE\"}"`.

## Audio Response

The audio response then delivers base64-encoded speech chunks sharing the same contentId throughout the stream.

## Barge-In Support

During audio generation, Amazon Nova Sonic supports natural conversation flow through its barge-in capability. When a user interrupts Amazon Nova Sonic while it's speaking, Nova Sonic immediately stops generating speech, switches to listening mode, and sends a content notification indicating the interruption has occurred. Because Nova Sonic operates faster than real-time, some audio may have already been delivered but not yet played. The interruption notification enables the client application to clear its audio queue and stop playback immediately, creating a responsive conversational experience.

## Final Transcription

After audio generation completes (or is interrupted via barge-in), Amazon Nova Sonic provides an additional text response that contains a sentence-level transcription of what was actually spoken. This text response includes a contentStart event with role: "ASSISTANT" and "additionalModelFields": "{\"generationStage\":\"FINAL\"}".

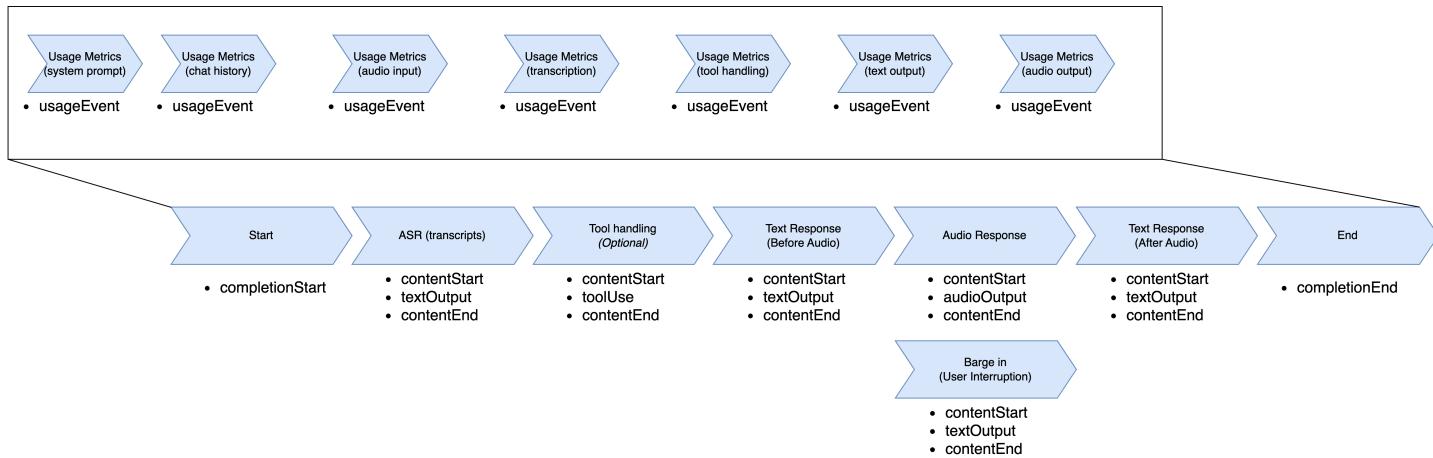
## Usage Tracking

Throughout the response handling, usageEvent events are sent to track token consumption. These events contain detailed metrics on input tokens and output tokens (both speech and text), and their cumulative totals. Each usageEvent maintains the same sessionId, promptName, and completionId as other events in the conversation flow. The details section provides both incremental changes (delta) and running totals of token usage, enabling precise monitoring of the usage during the conversation.

## Completion

The model sends a completionEnd event with the original identifiers and a stopReason that indicates how the conversation ended. This event hierarchy ensures your application can track which parts of the response belong together and process them accordingly, maintaining conversation context throughout multiple turns.

The output event flow begins by entering the response generation phase. It starts with automatic speech recognition, selects a tool for use, transcribes speech, generates audio, finalizes the transcription, and finishes the session.



## Output Event Flow

The structure of the output event flow is described in this section.

### 1. UsageEvent

```

"event": {
    "usageEvent": {
        "completionId": "string", // unique identifier for completion
        "details": {
            "delta": { // incremental changes since last event
                "input": {
                    "speechTokens": number, // input speech tokens
                    "textTokens": number // input text tokens
                },
                "output": {
                    "speechTokens": number, // speech tokens generated
                    "textTokens": number // text tokens generated
                }
            },
            "total": { // cumulative counts
                "input": {
                    "speechTokens": number, // total speech tokens processed
                    "textTokens": number // total text tokens processed
                },
                "output": {
                    "speechTokens": number, // total speech tokens generated

```

```
        "textTokens": number // total text tokens generated
    }
},
"promptName": "string", // same unique identifier from promptStart event
"sessionId": "string", // unique identifier
"totalInputTokens": number, // cumulative input tokens
"totalOutputTokens": number, // cumulative output tokens
"totalTokens": number // total tokens in the session
}
}
```

## 2. CompleteStartEvent

```
"event": {
    "completionStart": {
        "sessionId": "string", // unique identifier
        "promptName": "string", // same unique identifier from promptStart event
        "completionId": "string", // unique identifier
    }
}
```

## 3. TextOutputContent

### ContentStart

```
"event": {
    "contentStart": {
        "additionalModelFields": "{\"generationStage\":\"FINAL\"} | "
        "{\"generationStage\":\"SPECULATIVE\"}",
        "sessionId": "string", // unique identifier
        "promptName": "string", // same unique identifier from promptStart event
        "completionId": "string", // unique identifier
        "contentId": "string", // unique identifier for the content block
        "type": "TEXT",
        "role": "USER" | "ASSISTANT",
        "textOutputConfiguration": {
            "mediaType": "text/plain"
        }
    }
}
```

## TextOutput

```
"event": {  
    "textOutput": {  
        "sessionId": "string", // unique identifier  
        "promptName": "string", // same unique identifier from promptStart event  
        "completionId": "string", // unique identifier  
        "contentId": "string", // same unique identifier from its contentStart  
        "content": "string" // User transcribe or Text Response  
    }  
}
```

## ContentEnd

```
"event": {  
    "contentEnd": {  
        "sessionId": "string", // unique identifier  
        "promptName": "string", // same unique identifier from promptStart event  
        "completionId": "string", // unique identifier  
        "contentId": "string", // same unique identifier from its contentStart  
        "stopReason": "PARTIAL_TURN" | "END_TURN" | "INTERRUPTED",  
        "type": "TEXT"  
    }  
}
```

## 4. ToolUse

### ContentStart

```
"event": {  
    "contentStart": {  
        "sessionId": "string", // unique identifier  
        "promptName": "string", // same unique identifier from promptStart event  
        "completionId": "string", // unique identifier  
        "contentId": "string", // unique identifier for the content block  
        "type": "TOOL",  
        "role": "TOOL",  
        "toolUseOutputConfiguration": {  
            "mediaType": "application/json"  
        }  
    }  
}
```

## ToolUse

```
"event": {  
    "toolUse": {  
        "sessionId": "string", // unique identifier  
        "promptName": "string", // same unique identifier from promptStart event  
        "completionId": "string", // unique identifier  
        "contentId": "string", // same unique identifier from its contentStart  
        "content": "json",  
        "toolName": "string",  
        "toolUseId": "string"  
    }  
}
```

## ContentEnd

```
"event": {  
    "contentEnd": {  
        "sessionId": "string", // unique identifier  
        "promptName": "string", // same unique identifier from promptStart event  
        "completionId": "string", // unique identifier  
        "contentId": "string", // same unique identifier from its contentStart  
        "stopReason": "TOOL_USE",  
        "type": "TOOL"  
    }  
}
```

## 5. AudioOutputContent

### ContentStart

```
"event": {  
    "contentStart": {  
        "sessionId": "string", // unique identifier  
        "promptName": "string", // same unique identifier from promptStart event  
        "completionId": "string", // unique identifier  
        "contentId": "string", // unique identifier for the content block  
        "type": "AUDIO",  
        "role": "ASSISTANT",  
        "audioOutputConfiguration": {  
            "mediaType": "audio/lpcm",  
            "sampleRateHertz": 8000 | 16000 | 24000,  
        }  
    }  
}
```

```
        "sampleSizeBits": 16,  
        "encoding": "base64",  
        "channelCount": 1  
    }  
}  
}
```

## AudioOutput

```
"event": {  
    "audioOutput": {  
        "sessionId": "string", // unique identifier  
        "promptName": "string", // same unique identifier from promptStart event  
        "completionId": "string", // unique identifier  
        "contentId": "string", // same unique identifier from its contentStart  
        "content": "base64EncodedAudioData", // Audio  
    }  
}
```

## ContentEnd

```
"event": {  
    "contentEnd": {  
        "sessionId": "string", // unique identifier  
        "promptName": "string", // same unique identifier from promptStart event  
        "completionId": "string", // unique identifier  
        "contentId": "string", // same unique identifier from its contentStart  
        "stopReason": "PARTIAL_TURN" | "END_TURN",  
        "type": "AUDIO"  
    }  
}
```

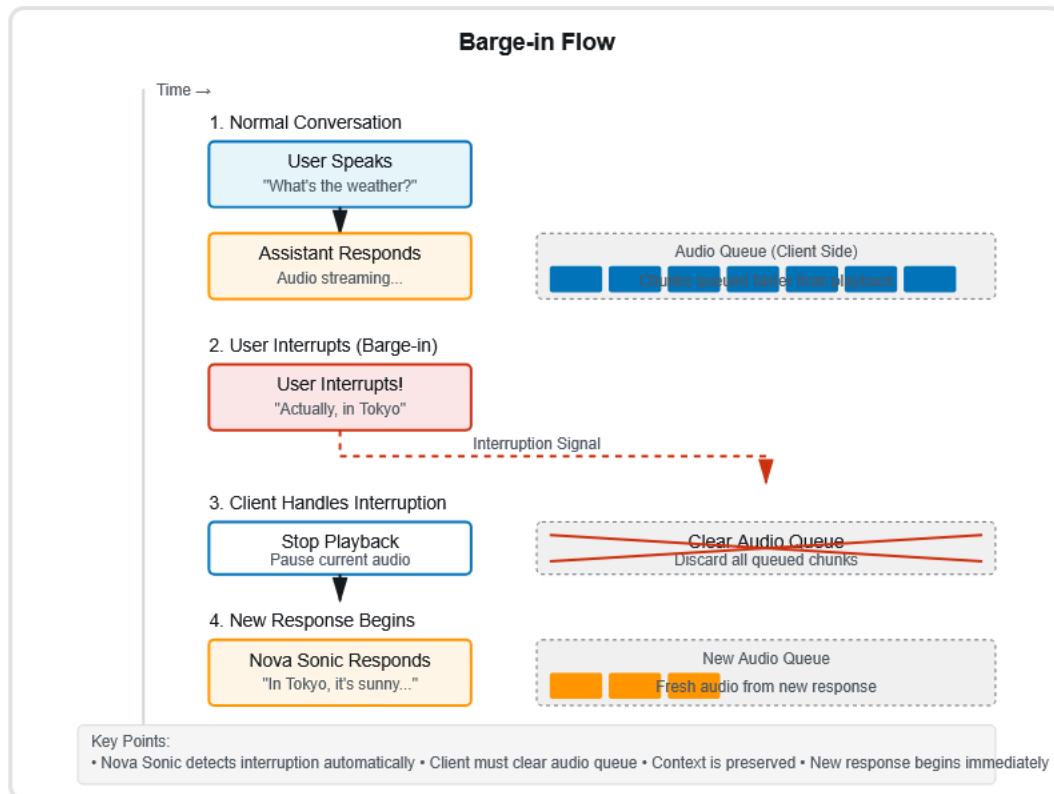
## 6. CompletionEndEvent

```
"event": {  
    "completionEnd": {  
        "sessionId": "string", // unique identifier  
        "promptName": "string", // same unique identifier from promptStart event  
        "completionId": "string", // unique identifier  
        "stopReason": "END_TURN"  
    }  
}
```

## Barge-in

Barge-in allows users to interrupt the AI assistant while it's speaking, just like in natural human conversations. Instead of waiting for the assistant to finish, users can interject with new information, correct or clarify their previous statement, redirect the conversation to a different topic, or simply stop the assistant when they've heard enough. This creates a more natural and responsive conversational experience.

The following diagram illustrates the complete barge-in conversation flow:



## How Amazon Nova 2 Sonic handles barge-in

Amazon Nova 2 Sonic is designed to handle interruptions gracefully. When the user starts speaking during a response, the system immediately stops generating the current response, maintains full conversational context, sends an interruption signal to the client and begins processing the new user input.

**Context Preservation:** Even when interrupted, Nova Sonic remembers what was said before the interruption, the topic being discussed, the conversation history and any relevant context from previous turns. This ensures the conversation remains coherent and natural.

## Client-side implementation requirements

While Amazon Nova 2 Sonic handles barge-in on the server side, you need to implement client-side logic for a complete experience.

**The audio queue challenge:** Audio generation is faster than playback speed. This means:

- Nova Sonic generates audio chunks quickly
- Your client receives and queues these chunks
- The client plays them back at normal speaking speed
- When a barge-in occurs, there's already audio queued for playback

**Required client-side logic:** Your application must handle four key steps:

1. **Detect the Interruption Signal:** Listen for the interruption event from Nova Sonic and react immediately when received.
2. **Stop Current Playback:** Pause the currently playing audio and stop any audio that's mid-playback.
3. **Clear the Audio Queue:** Remove all queued audio chunks and discard any buffered audio from the interrupted response.
4. **Start New Audio:** Begin playing the newly received audio and resume normal playback flow.

## Turn-taking controllability

Turn-taking is a fundamental aspect of natural conversation. Amazon Nova 2 Sonic provides fine-grained control over when the AI takes its turn to speak through the `turnDetectionConfiguration` parameter. This allows you to optimize the conversation flow for different use cases, balancing responsiveness with accuracy. The `endpointingSensitivity` parameter controls how quickly Amazon Nova 2 Sonic detects the end of a user's turn and begins responding. This setting affects both the latency of responses and the likelihood of interrupting users who are still speaking.

## API configuration

Configure turn detection sensitivity in the `sessionStart` event:

```
{  
  "event": {  
    "sessionStart": {  
      "inferenceConfiguration": {  
        "maxTokens": 1000,  
        "topP": 0.9,  
        "temperature": 0.7  
      },  
      "turnDetectionConfiguration": {  
        "endpointingSensitivity": "HIGH" | "MEDIUM" | "LOW"  
      }  
    }  
  }  
}
```

## Sensitivity levels

The endpointingSensitivity parameter accepts three values: HIGH, MEDIUM, and LOW. Each level balances response speed against the risk of interrupting users who are still speaking.

### HIGH

Fastest response time, optimized for latency. Nova Sonic responds as quickly as possible after detecting the end of speech. Pause duration: 1.5 seconds. Best for quick Q and A, command-and-control applications, and time-sensitive interactions.

### MEDIUM

Balanced approach with moderate response time. Reduces false positives while maintaining responsiveness. Pause duration: 1.75 seconds. Best for general conversations, customer service with complex queries, and multi-turn discussions.

### LOW

Slowest response time with maximum patience. Nova Sonic waits the longest before responding, minimizing interruptions of users who pause while thinking. Pause duration: 2 seconds. Best for thoughtful conversations, elderly or speech-impaired users, and complex problem-solving.

## Pause duration reference

Sensitivity level	Pause duration (seconds)
High (fast)	1.5
Medium	1.75
Low (slow)	~2.0

## Cross-modal input

Amazon Nova 2 Sonic now supports cross-modal input, allowing you to send text messages in addition to voice input during a conversation session. While speech remains the primary mode of interaction, text input provides flexibility for scenarios where typing is more convenient or appropriate.

**Continuous streaming required:** Cross-modal input requires an active streaming session to function properly. The session must maintain continuous streaming like a regular voice session, otherwise standard session timeouts will be applied and the connection will be terminated.

Sensitivity levels in cross-modal text input is useful for scenarios such as:

- Client-side app integration (web and mobile): Allows users to interact with the application using both text and voice, supporting seamless multimodal experiences.
- "Model-start-first" use case: A text message can be sent immediately after the session starts to prompt the model to begin speaking.
- Guiding the model during async tool calling: When a toolUse event is triggered and the system begins processing tool calls, the client can send a text message to Sonic to provide a natural response while waiting — such as, "Hold on a second while I process your information. In the meantime, is there anything else I can assist with?"
- Telephony DTMF integration: Customer uses phone keypad to enter sensitive information (such as credit card numbers). Note: Amazon Nova Sonic does not process DTMF tones natively. To support DTMF input, your system must detect the tones, convert them to text (such as "1234"), and send to Nova 2 Sonic.

## How it works

Cross-modal input uses a three-event sequence similar to audio input:

1. **Content Start Event:** Signals the beginning of text input
2. **Text Input Event:** Contains the actual text message
3. **Content End Event:** Signals the completion of text input

All three events must use the same promptName and contentName to maintain the sequence. A new UUID should be generated for contentName each time you send text input to ensure proper multi-turn conversation tracking.

## Event structure

### 1. Content Start Event

Initiates the text input sequence with configuration details:

```
{  
  "event": {  
    "contentStart": {  
      "promptName": "<prompt_name>",  
      "contentName": "<new_content_name>",  
      "role": "USER",  
      "type": "TEXT",  
      "interactive": true,  
      "textInputConfiguration": {  
        "mediaType": "text/plain"  
      }  
    }  
  }  
}
```

### Key Parameters:

- **promptName:** The name of your conversation prompt (consistent across the session)
- **contentName:** A unique identifier for this text input (generate a new UUID for each message)
- **role:** Set to "USER" to indicate user input
- **type:** Set to "TEXT" for text input

- **interactive:** Set to `true` to enable interactive mode
- **mediaType:** Set to "`text/plain`" for plain text content

## 2. Text Input Event

Contains the actual text message content:

```
{  
  "event": {  
    "textInput": {  
      "promptName": "<prompt_name>",  
      "contentName": "<new_content_name>",  
      "content": "<your_text_message>"  
    }  
  }  
}
```

Key Parameters:

- **promptName:** Must match the value from Content Start Event
- **contentName:** Must match the value from Content Start Event
- **role:** Your text message string

## 3. Content End Event

Signals the completion of the text input:

```
{  
  "event": {  
    "contentEnd": {  
      "promptName": "<prompt_name>",  
      "contentName": "<new_content_name>"  
    }  
  }  
}
```

Key Parameters:

- **promptName:** Must match the value from previous events

- **contentName:** Must match the value from previous events

## Language support and multilingual capabilities

Amazon Nova 2 Sonic provides a diverse selection of voices across multiple languages, enabling you to create conversational AI applications that feel natural and culturally appropriate for your users. Each language offers both feminine-sounding and masculine-sounding voice options.

The following table lists all available voices and their corresponding language locales:

Language	Locale	Feminine-sounding Voice ID	Masculine-sounding Voice ID
English (US)	en-US	tiffany	matthew
English (UK)	en-GB	amy	-
English (Australia)	en-AU	olivia	-
English (Indian)	en-IN	kiara	arjun
French	fr-FR	ambre	florian
Italian	it-IT	beatrice	lorenzo
German	de-DE	tina	lennart
Spanish (US)	es-US	lupe	carlos
Portuguese	pt-BR	carolina	leo
Hindi	hi-IN	kiara	arjun

## Event structure using voices in your application

You can specify the voice ID in the `audioOutputConfiguration` when starting a prompt in the `promptStart` event:

```
"event": {
```

```
"promptStart": {  
    "promptName": "string",  
    "audioOutputConfiguration": {  
        "mediaType": "audio/lpcm",  
        "sampleRateHertz": 16000,  
        "sampleSizeBits": 16,  
        "channelCount": 1,  
        "voiceId": "tiffany",  
        "encoding": "base64",  
        "audioType": "SPEECH"  
    }  
}  
}
```

## Multilingual support

Amazon Nova 2 Sonic provides powerful multilingual capabilities that enable natural conversations across multiple languages. The service supports both polyglot voices (speaking multiple languages) and code-switching (mixing languages within the same sentence), allowing you to build truly global conversational applications.

The TIFFANY (en-US, female) and MATTHEW (en-US, male) are unique polyglot voices that can speak all supported languages:

1. English
2. French
3. Italian
4. German
5. Spanish
6. Portuguese
7. Hindi

This makes Tiffany and Matthew ideal for applications that need to switch between multiple languages seamlessly.

## Managing chat history

Amazon Nova 2 Sonic responses include ASR (Automatic Speech Recognition) transcripts for both user and assistant voices. Storing chat history is a best practice—not only for logging purposes but

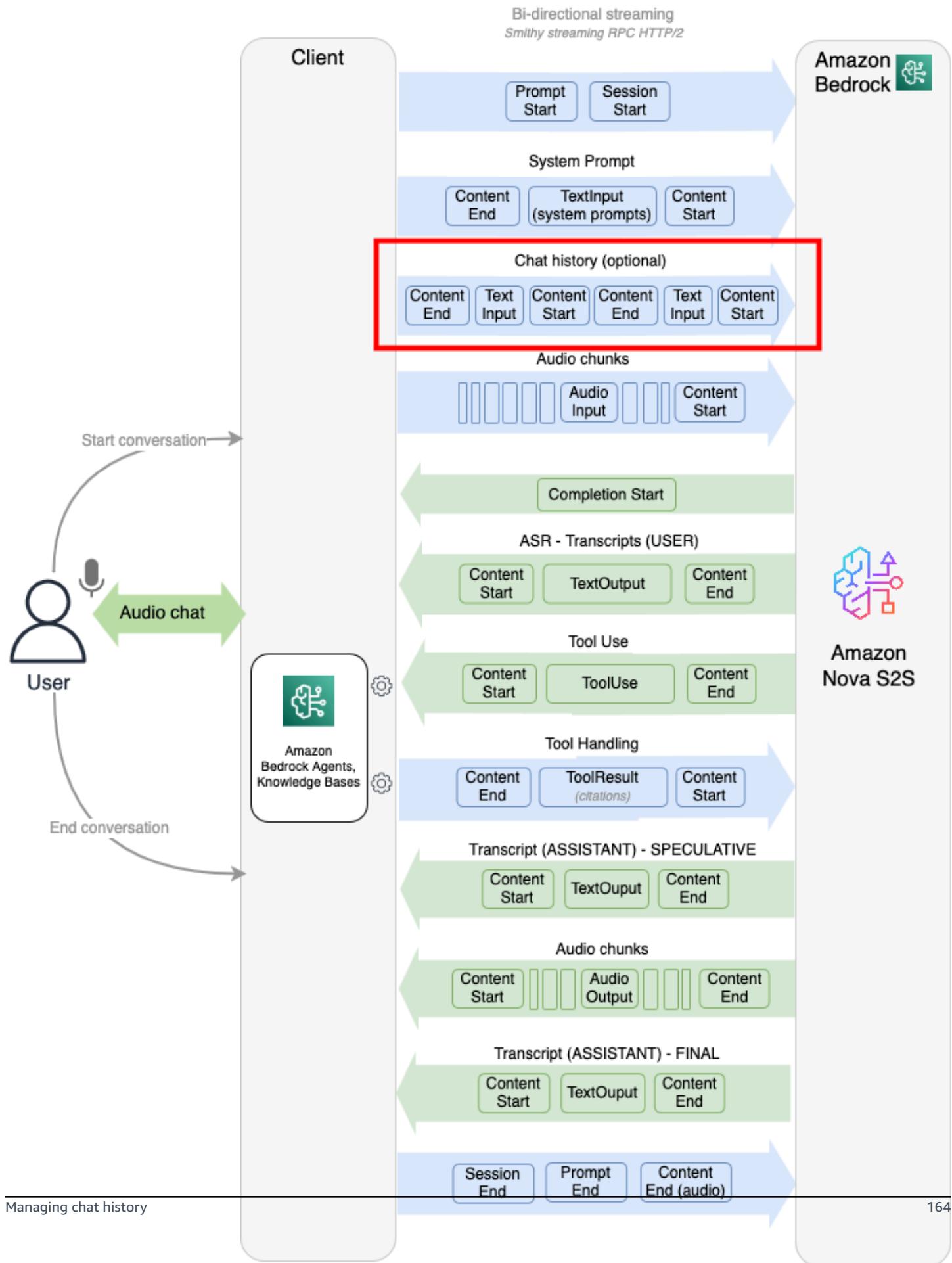
also for resuming sessions when the connection is unexpectedly closed. This allows the client to send context back to Nova Sonic to continue the conversation seamlessly.

Refer to the following resources for more information on managing chat history:

1. [Logging chat history](#)
2. [Resuming conversations](#)

## Sending chat history

A conversation history can be included only once, after the system/speech prompt and before audio streaming begins. Overall chat history cannot be larger than 40KB. The following diagram shows when chat history is passed in during the event lifecycle:



Each historical message requires three events: `contentStart`, `textInput` and `contentEnd`.

### Event schema per message:

- `contentStart` - Defines the message role and configuration

```
{  
  "event": {  
    "contentStart": {  
      "promptName": "<prompt-id>",  
      "contentName": "<content-id>",  
      "type": "TEXT",  
      "interactive": true,  
      "role": "ASSISTANT",  
      "textInputConfiguration": {  
        "mediaType": "text/plain"  
      }  
    }  
  }  
}
```

- `textInput` - Contains the actual message content. One `textInput` cannot be larger than 1KB. If so, split into multiple `textInput`s in the same content block. If the conversation is larger than 40KB, trim the overall chat history.

```
{  
  "event": {  
    "textInput": {  
      "promptName": "<prompt-id>",  
      "contentName": "<content-id>",  
      "content": "Take your time, Don. I'll be here when you're ready."  
    }  
  }  
}
```

- `contentEnd` - Marks the end of the message

```
{  
  "event": {  
    "contentEnd": {  
      "promptName": "<prompt-id>",  
      "contentName": "<content-id>"  
    }  
  }  
}
```

```
    }  
}
```

Repeat these three events for each message in your chat history, alternating between USER and ASSISTANT roles.

### Important considerations:

- Chat history can only be included **once** per session
- Chat history must be sent **after the system prompt** and **before audio streaming begins**
- All historical messages must be sent before starting the audio streaming
- Each message must specify either USER or ASSISTANT role
- Use the stored transcript content from textOutput events as the content value in textInput

## Receiving ASR transcripts

During a conversation, Amazon Nova 2 Sonic sends ASR transcripts through output events. Each transcript is delivered as a sequence of three events: contentStart, textOutput, and contentEnd.

### Example: User speech transcript:

#### 1. contentStart - Indicates the beginning of a transcript:

```
{  
  "event": {  
    "contentStart": {  
      "additionalModelFields": "{\"generationStage\":\"FINAL\"}",  
      "completionId": "<completion-id>",  
      "contentId": "<content-id>",  
      "promptName": "<prompt-id>",  
      "role": "USER",  
      "sessionId": "<session-id>",  
      "textOutputConfiguration": {  
        "mediaType": "text/plain"  
      },  
      "type": "TEXT"  
    }  
  }  
}
```

## 2. textOutput - Contains the actual transcript content:

```
{  
  "event": {  
    "textOutput": {  
      "completionId": "<completion-id>",  
      "content": "hello how are you",  
      "contentId": "<content-id>",  
      "promptName": "<prompt-id>",  
      "role": "USER",  
      "sessionId": "<session-id>"  
    }  
  }  
}
```

## 3. contentEnd - Marks the end of the transcript:

```
{  
  "event": {  
    "contentEnd": {  
      "completionId": "<completion-id>",  
      "contentId": "<content-id>",  
      "promptName": "<prompt-id>",  
      "sessionId": "<session-id>",  
      "stopReason": "PARTIAL_TURN",  
      "type": "TEXT"  
    }  
  }  
}
```

The same three-event pattern applies for both USER and ASSISTANT roles. Extract the content field from the textOutput event and the role field from the contentStart event to build your chat history.

## Best practices

Always store chat history to enable:

- Session resumptions across difference devices
- Conversation logging and auditing
- Context preservation for follow-up interactions

**Important:** When saving chat history, use text outputs based on their generationStage:

- Speculative - A preview of what Nova 2 Sonic plans to say, generated before audio synthesis begins
- Final - The actual sentence-level transcription of what was spoken in the audio response

Always save the FINAL text output to your chat history, as it represents the accurate record of the conversation.

Example of FINAL output (save this to chat history):

```
ContentStart event: {
  "additionalModelFields": "{\"generationStage\":\"FINAL\"}",
  "completionId": "<completion-id>",
  "contentId": "<content-id>",
  "role": "ASSISTANT",
  "sessionId": "<session-id>",
  "type": "TEXT"
}
```

Example of SPECULATIVE output (optional preview, not for history):

```
ContentStart event: {
  "additionalModelFields": "{\"generationStage\":\"SPECULATIVE\"}",
  "completionId": "<completion-id>",
  "contentId": "<content-id>",
  "role": "ASSISTANT",
  "sessionId": "<session-id>",
  "type": "TEXT"
}
```

## Tool configuration

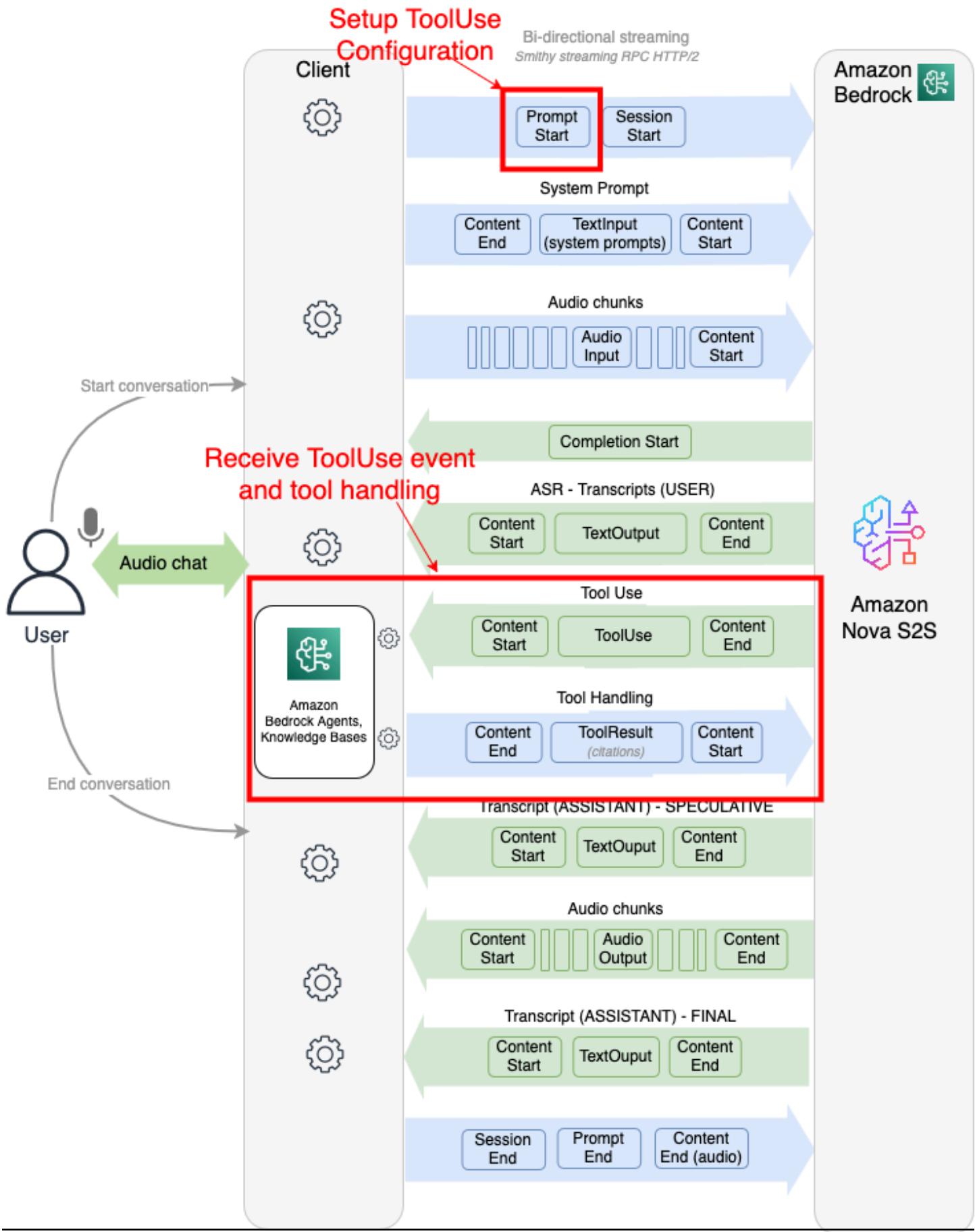
Amazon Nova 2 Sonic supports tool use (also known as function calling), allowing the model to request external information or actions during conversations, such as API calls, database queries, or custom code functions. This allows your voice assistant to take actions, retrieve information, and integrate with external services based on user requests.

Nova 2 Sonic features asynchronous tool calling, enabling the AI to continue conversing naturally while tools run in the background - creating a more fluid and responsive user experience.

The following are simplified steps on how to use tools:

1. Define tools: specify available tools with their parameters in the promptStart event
2. User speaks: user makes a request that requires a tool (such as, "What's the weather in Seattle?")
3. Tool invocation: Nova 2 Sonic recognizes the need and sends a toolUse event
4. Execute tool: Your application executes the tool and returns results
5. Response Generation: Nova 2 Sonic incorporates the results into its spoken response

The following diagram illustrates how tool use works:



## Defining tools

Tools are defined using a JSON schema that describes their purpose, parameters, and expected inputs.

The following are tool definition components and explanations:

- Name: Unique identifier for the tool (use snake\_case)
- Description: Clear explanation of what the tool does; helps the AI decide when to use it
- InputSchema: JSON schema defining the parameters the tool accepts
- Properties: Individual parameters with types and descriptions
- Required: Array of parameter names that must be provided

### Example of tool definition

Here's a simple weather tool definition

```
{  
  "toolSpec": {  
    "name": "get_weather",  
    "description": "Get current weather information for a specific location",  
    "inputSchema": {  
      "json": {  
        "type": "object",  
        "properties": {  
          "location": {  
            "type": "string",  
            "description": "City name or zip code"  
          },  
          "units": {  
            "type": "string",  
            "enum": ["celsius", "fahrenheit"],  
            "description": "Temperature units"  
          }  
        },  
        "required": ["location"]  
      }  
    }  
  }  
}
```

## Configuring Tools in PromptStart

Tool configuration is passed to Nova 2 Sonic in the `promptStart` event along with audio and text output settings:

```
{  
    "event": {  
        "promptStart": {  
            "promptName": "<prompt-id>",  
            "textOutputConfiguration": {  
                "mediaType": "text/plain"  
            },  
            "audioOutputConfiguration": {  
                "mediaType": "audio/lpcm",  
                "sampleRateHertz": 16000,  
                "sampleSizeBits": 16,  
                "channelCount": 1,  
                "voiceId": "matthew",  
                "encoding": "base64",  
                "audioType": "SPEECH"  
            },  
            "toolUseOutputConfiguration": {  
                "mediaType": "application/json"  
            },  
            "toolConfiguration": {  
                "tools": [  
                    {  
                        "toolSpec": {  
                            "name": "get_weather",  
                            "description": "Get current weather information for a  
specific location",  
                            "inputSchema": {  
                                "json": {  
                                    "type": "object",  
                                    "properties": {  
  "location": {  
  "type": "string",  
  "description": "City name or zip code"  
  },  
  "units": {  
  "type": "string",  
  "enum": ["celsius", "fahrenheit"],  
  "description": "Temperature units"  
  }  
                                    }  
                                }  
                            }  
                        }  
                    ]  
                }  
            }  
        }  
    }  
}
```

```
        },
        "required": ["location"]
    }
}
],
"toolChoice": {
    "auto": {}
}
}
}
}
```

## Tool Choice Parameters

Nova 2 Sonic supports three tool choice parameters to control when and which tools are used. Specify the toolChoice parameter in your tool configuration:

- Auto (default): The model decides whether any tools are needed and can call multiple tools if required. Provides maximum flexibility.
- Any: Ensures at least one of the available tools is called at the beginning of the response, with the model selecting the most appropriate one. Useful when you have multiple knowledge bases or tools and want to guarantee one is used.
- Tool: Forces a specific named tool to be called exactly once at the beginning of the response. For example, if you specify a knowledge base tool, the model will query it before responding, regardless of whether it thinks the tool is needed.

## Tool Choice Examples

### Auto (default)

```
"toolChoice": {
    "#"auto": {}
}
```

### Any:

```
"toolChoice": {
```

```
"any": {}  
}
```

## Specific Tool:

```
"toolChoice": {  
    "tool": {  
        "name": "get_weather"  
    }  
}
```

## Receiving and processing tool use events

When Amazon Nova 2 Sonic determines that a tool is needed, it sends a toolUse event containing:

1. toolUseID: unique identifier for this tool invocation
2. ToolName: the tool name to execute
3. Content: JSON string containing parameters extracted from the user's request
4. SessionID: current session identifier
5. Role: set to "TOOL" for tool use events

### Example tool use event

```
{  
    "event": {  
        "toolUse": {  
            "completionId": "<completion-id>",  
            "content": "{\"location\": \"Seattle\", \"units\": \"fahrenheit\"}",  
            "contentId": "<content-id>",  
            "promptName": "<prompt-id>",  
            "role": "TOOL",  
            "sessionId": "<session-id>",  
            "toolName": "get_weather",  
            "toolUseId": "<tool-use-id>"  
        }  
    }  
}
```

## Processing steps

1. Receive the toolUse event from Nova 2 Sonic
2. Extract the tool name and parameters from the event
3. Execute your tool logic (API call, database query, and so on)
4. Return the result using a toolResult event

### Example ToolResult Event

```
{  
  "event": {  
    "toolResult": {  
      "promptName": "<prompt-id>",  
      "contentName": "<content-id>",  
      "content": "{\"temperature\": 72, \"condition\": \"sunny\", \"humidity\":  
45}"  
    }  
  }  
}
```

### Best practices

- Clear descriptions: Write detailed tool descriptions to help Nova 2 Sonic understand when to use each tool.
- Validate parameters: Always validate tool parameters before execution to prevent errors. Define tool parameters using proper JSON schema with structured data types (such as enums, numbers, or booleans) rather than open-ended strings whenever possible.
- Error handling: Return meaningful error messages in toolResult events when tools fail.
- Async execution: Take advantage of asynchronous tool calling to maintain conversation flow.
- Tool naming: Use descriptive, action-oriented names (such as get\_weather, search\_database, send\_email).

## Asynchronous tool calling

Unlike traditional synchronous tool calling where the AI waits silently for tool results, Amazon Nova 2 Sonic's asynchronous approach allows it to:

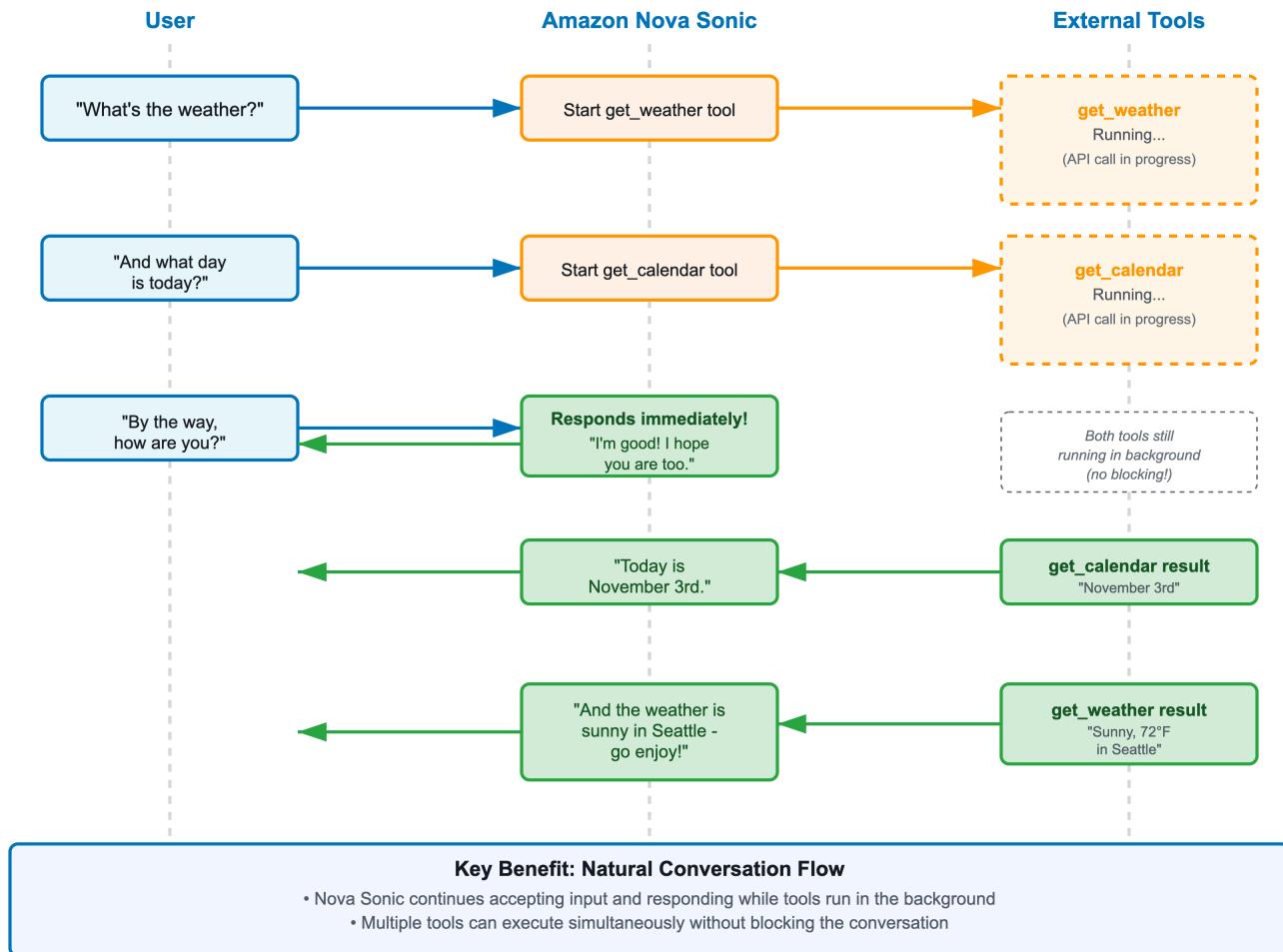
- Continue accepting user input while tools are running
- Respond to new questions without waiting for pending tool results

- Handle multiple tool calls simultaneously
- Maintain natural conversation flow without awkward pauses
- No extra configuration is required. Asynchronous tool calling works out of the box.

## How it works

When Nova 2 Sonic issues a tool call, it doesn't pause the conversation. Instead, it continues listening and responding naturally until the tool arrives.

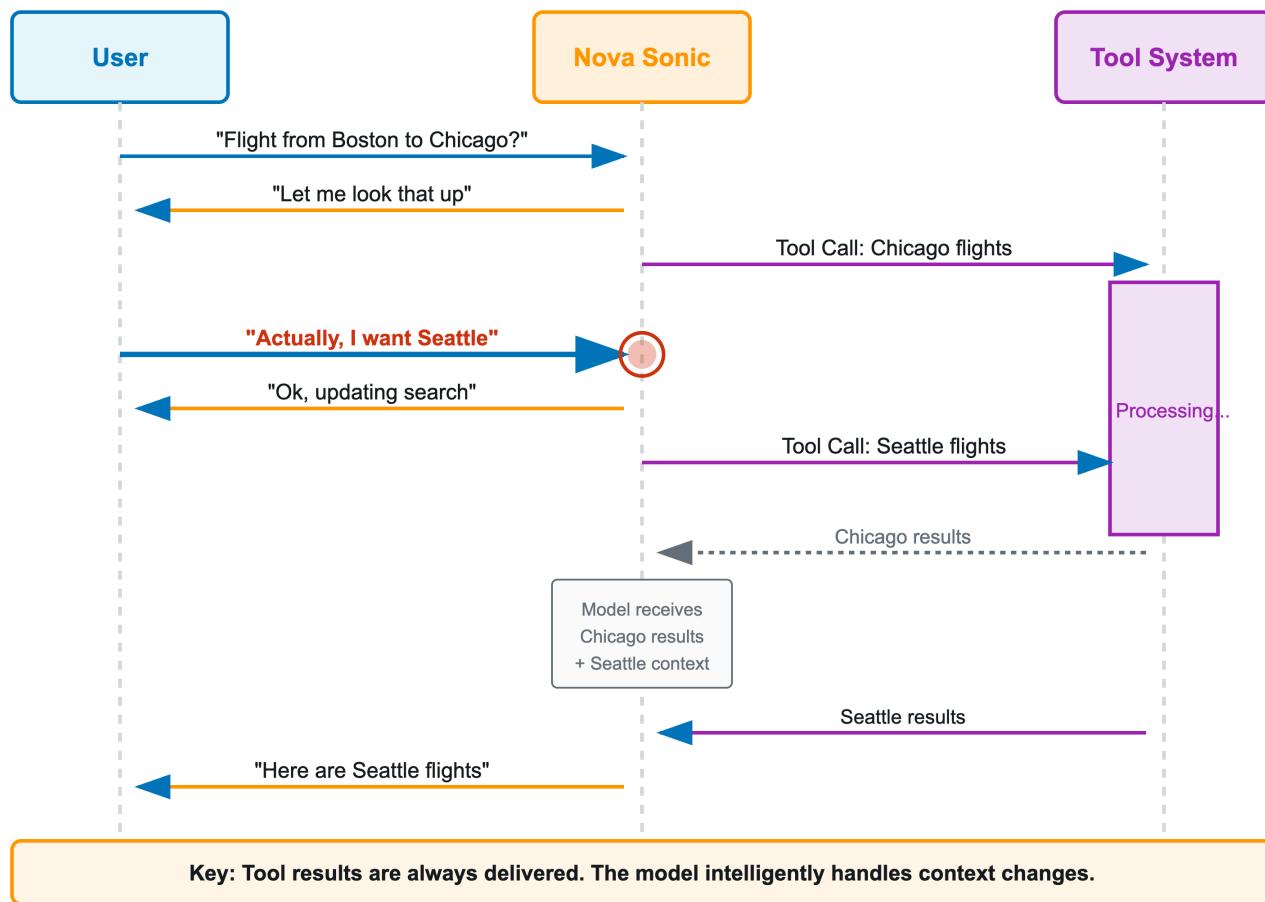
### Asynchronous Tool Calling Flow



## Handling user interruptions

If a user changes their request while a tool is executing, Nova 2 Sonic handles it intelligently without canceling pending tool calls.

## Async Tool Calling with User Interruption



### Example Scenario

```

User: "Can I book a flight from Boston to Chicago?"
Agent: "Sure, let me look that up for you."
Agent: [initiates tool call for Chicago flights]
User: "Actually, I want to go to Seattle"
Agent: "Ok let me update that search"
Agent: [initiates tool call for Seattle flights]
[First tool returns with Chicago flight results]
Agent: [receives Chicago results and processes them contextually]

```

## How it works

Tool results are always delivered: When a tool call completes, its result is always sent to the model, even if the user has changed their request. The model uses its reasoning capabilities to determine how to handle the information.

Context-aware processing: The model understands the conversation context and can appropriately handle outdated tool results. For example:

- If the user says "thank you" after changing their mind, the model still needs the original results for context
- If the user changes their request, the model can acknowledge the original results while focusing on the new request

No automatic cancellation: The system does not automatically cancel or ignore tool calls based on new user input. This ensures the model has complete information to make intelligent decisions about how to respond.

## Integrations

Amazon Nova 2 Sonic can be integrated with various frameworks and platforms to build conversational AI applications. These integrations provide pre-built components and simplified APIs for common use cases.

## Strands Agents

Strands Agents is a simple yet powerful SDK that takes a model-driven approach to building and running AI agents. From simple conversational assistants to complex autonomous workflows, from local development to production deployment, Strands Agents scales with your needs.

For comprehensive documentation on the Strands framework, visit the [official Strands documentation](#).

The Strands BidiAgent provides real-time audio and text interaction through persistent streaming connections. Unlike traditional request-response patterns, this agent maintains long-running conversations with support for interruptions, concurrent processing and continuous audio responses.

## Prerequisites:

- Python 3.8 or later installed
- Credentials for AWS configured with access to Amazon Bedrock
- Basic familiarity with Python `async/await` syntax

## Code example:

### Code example

#### Installation:

Install the required packages:

```
pip install strands-agents strands-agents-tools
```

Run this example:

```
import asyncio
from strands.experimental.bidi.agent import BidiAgent
from strands.experimental.bidi.io.audio import BidiAudioIO
from strands.experimental.bidi.io.text import BidiTextIO
from strands.experimental.bidi.models.novasonic import BidiNovaSonicModel
from strands_tools import calculator

async def main():
    """Test the BidirectionalAgent API."""
    # Audio and Text input/output utility
    audio_io = BidiAudioIO(audio_config={})
    text_io = BidiTextIO()

    # Nova Sonic model
    model = BidiNovaSonicModel(region="us-east-1")

    async with BidiAgent(model=model, tools=[calculator]) as agent:
        print("New BidiAgent Experience")
        print("Try asking: 'What is 25 times 8?' or 'Calculate the square root of 144'")
        await agent.run(
            inputs=[audio_io.input()],
```

```
        outputs=[audio_io.output(), text_io.output()]
    )

if __name__ == "__main__":
    try:
        asyncio.run(main())
    except KeyboardInterrupt:
        print("\nConversation ended by user")
    except Exception as e:
        print(f"Error: {e}")
        import traceback
        traceback.print_exc()
```

## 1. Import: Required Models

```
from strands.experimental.bidi.agent import BidiAgent
from strands.experimental.bidi.io.audio import BidiAudioIO
from strands.experimental.bidi.io.text import BidiTextIO
from strands.experimental.bidi.models.novasonic import BidiNovaSonicModel
from strands_tools import calculator
```

- **BidiAgent:** The main agent class that orchestrates bidirectional conversations
- **BidiAudioIO:** Handles audio input and output for speech interactions
- **BidiTextIO:** Provides text output for transcriptions and responses
- **BidiNovaSonicModel** The Nova 2 Sonic model wrapper
- **Calculator:** A pre-built tool for mathematical operations

## 2. Configure Audio and Text I/O

```
audio_io = BidiAudioIO(audio_config={})
text_io = BidiTextIO()
```

The BidiAudioIO manages microphone input and speaker output, while BidiTextIO displays text transcriptions and responses in the console.

## 3. Initialize the Model

```
model = BidiNovaSonicModel(region="us-east-1")
```

Create a Nova Sonic model instance. The region parameter specifies the AWS region where the model is deployed.

#### 4. Create and Run the Agent

```
async with BidiAgent(model=model, tools=[calculator]) as agent:  
    await agent.run(  
        inputs=[audio_io.input()],  
        outputs=[audio_io.output(), text_io.output()])  
)
```

The agent is created with:

- Model: The Nova 2 Sonic model to use
- Tools: List of tools the agent can call (like calculator)
- Inputs: Audio input from the microphone
- Outputs: Audio output to speakers and text output to console

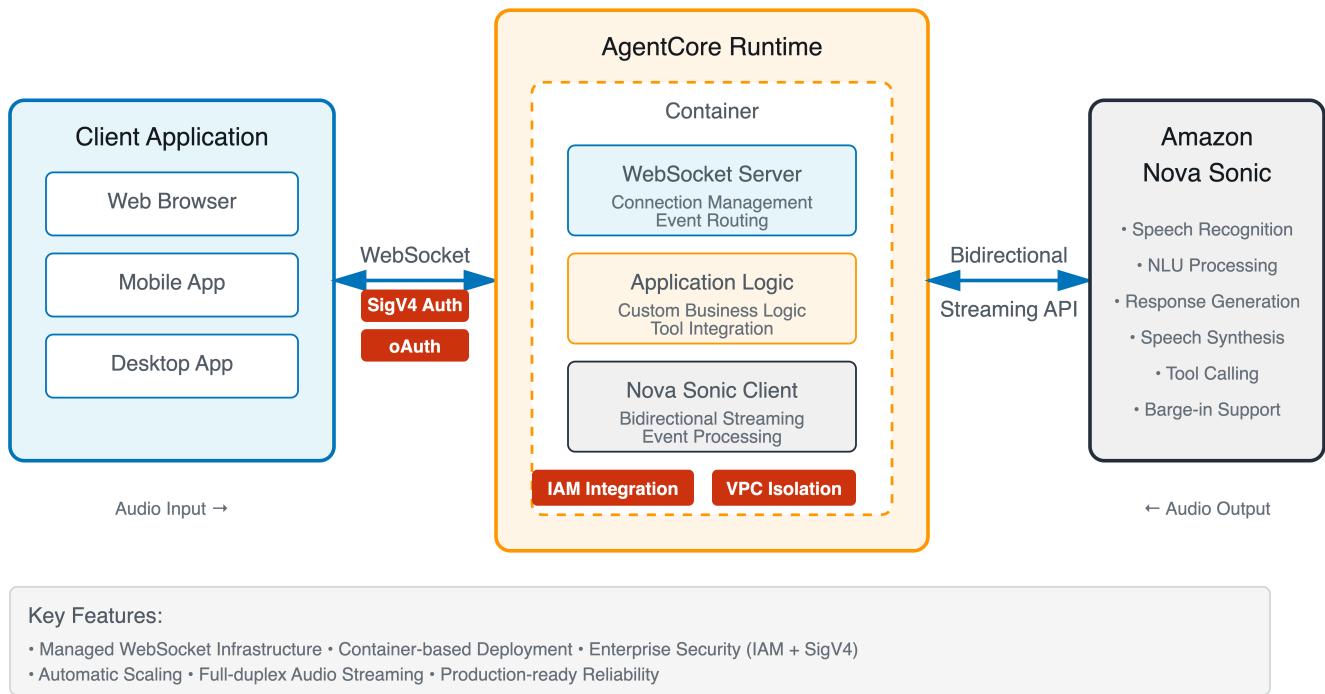
## Framework integrations

Amazon Nova 2 Sonic can be integrated with various frameworks and platforms to build sophisticated voice applications. The following examples demonstrate integration patterns with popular frameworks.

### Amazon Bedrock AgentCore

Amazon Bedrock AgentCore provides a managed runtime environment for deploying Nova 2 Sonic applications with enterprise-grade security and scalability. AgentCore simplifies the deployment of real-time voice AI applications by handling infrastructure, authentication, and WebSocket connectivity.

## AgentCore Architecture



### Key features:

- Bidirectional streaming - Native support for Nova Sonic's full-duplex streaming interface with real-time event processing and low-latency communication.
- WebSocket infrastructure - Production-ready WebSocket servers with automatic scaling, connection management, and error recovery.
- Container deployment - Deploy Nova Sonic applications as containers to managed infrastructure with horizontal scaling and independent versioning.
- Enterprise security - Fine-grained authentication via IAM and SigV4, VPC isolation, and comprehensive audit logging.

The architecture shows how client applications connect to AgentCore Runtime via WebSocket with SigV4 authentication. The containerized environment includes your WebSocket server, application logic, and Nova Sonic client, all communicating with Nova Sonic through the bidirectional streaming API.

### Benefits:

- Simplified operations: Focus on application logic while AgentCore manages infrastructure, scaling, and reliability.
- Enterprise security: Built-in authentication, authorization, and compliance features for production deployments.
- Cost efficiency: Pay only for what you use with automatic scaling and resource optimization.
- Developer productivity: Reduce time to production with managed WebSocket infrastructure and container deployment.

## Use cases

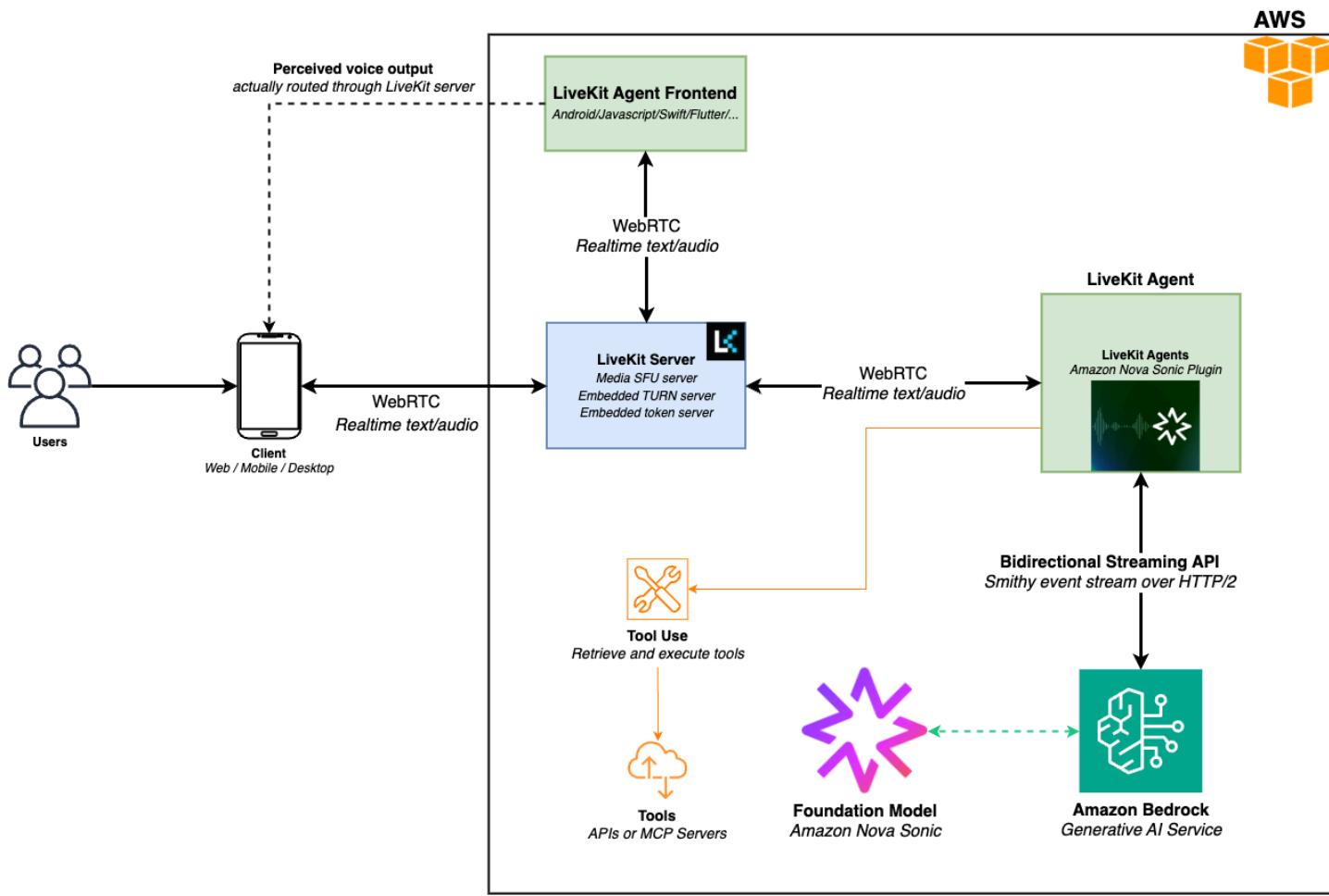
- Customer service voice assistants with secure authentication
- Enterprise voice applications requiring IAM integration
- Multi-tenant voice platforms with isolated deployments
- Voice-enabled applications requiring compliance and audit trails

For detailed documentation on deploying Nova Sonic with AgentCore, visit the [Amazon Bedrock AgentCore Documentation](#).

## LiveKit

LiveKit is an open-source platform for building real-time audio and video applications. Integration with Amazon Nova 2 Sonic enables developers to build conversational voice interfaces without managing complex audio pipelines or signaling protocols.

For detailed implementation examples and code examples, visit the [LiveKit AWS Integration Documentation](#).



## How it works:

- Client layer: Web, mobile, or desktop applications connect using LiveKit's client SDKs, which handle audio capture, WebRTC streaming, and playback.
- LiveKit Server: Acts as the real-time communication hub, managing WebRTC connections, routing audio streams, and handling session state with low-latency optimization.
- LiveKit Agent: Python-based agent that receives audio from the server, processes it through the Nova Sonic plugin, and streams responses back. Includes built-in features like voice activity detection and turn management.
- Amazon Nova 2 Sonic: Processes the audio stream through bidirectional streaming API, performing speech recognition, natural language understanding, and generating conversational responses with synthesized speech.

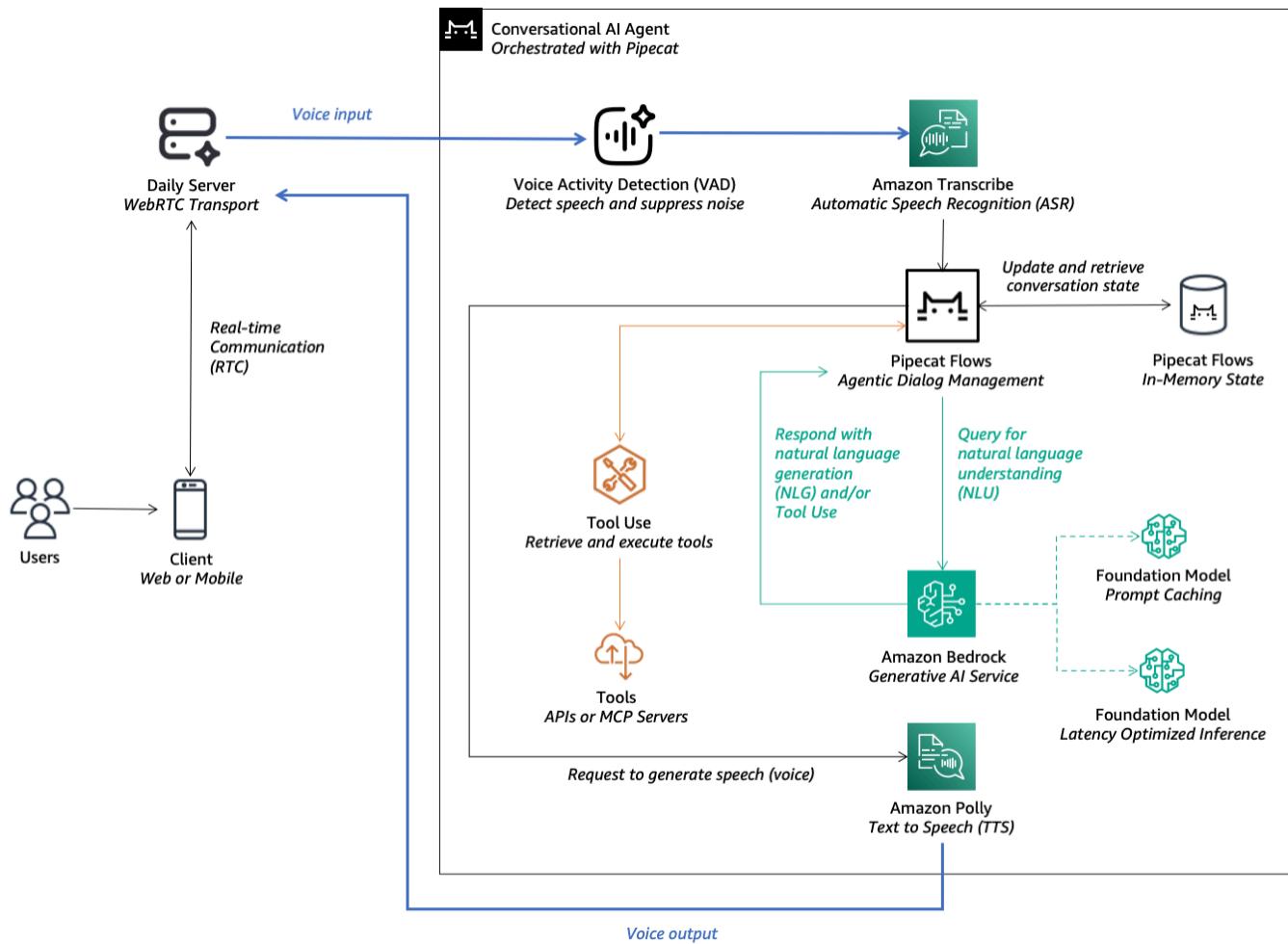
## Pipecat

Pipecat is a framework for building voice and multimodal conversational AI applications. It provides a modular, pipeline-based architecture that orchestrates multiple components to create intelligent voice applications with Amazon Nova Sonic and other AWS services.

For detailed implementation examples and code samples, visit the [PipeCat AWS Integration Documentation](#).

### Key features:

- Pipeline architecture: Modular Python-based framework for composing voice AI components including ASR, NLU, TTS, and more.
- Pipecat flows: State management framework for building complex conversational logic and tool execution.
- WebRTC Support: Built-in integration with Daily and other WebRTC providers for real-time audio streaming.
- AWS Integration: Native support for Amazon Bedrock, Amazon Transcribe, and Amazon Polly.



The architecture includes:

- **WebRTC Transport:** Real-time audio streaming between client devices and application server.
- **Voice activity detection (VAD):** Silero VAD with configurable speech detection and noise suppression.
- **Speech recognition:** Amazon Transcribe for accurate, real-time speech-to-text conversion.
- **Natural language understanding:** Amazon Nova Pro on Bedrock with latency-optimized inference.
- **Tool execution:** Pipecat Flows for API integration and backend service calls.
- **Response generation:** Amazon Nova Pro for coherent, context-aware responses.
- **Text-to-speech:** Amazon Polly with generative voices for lifelike speech output.

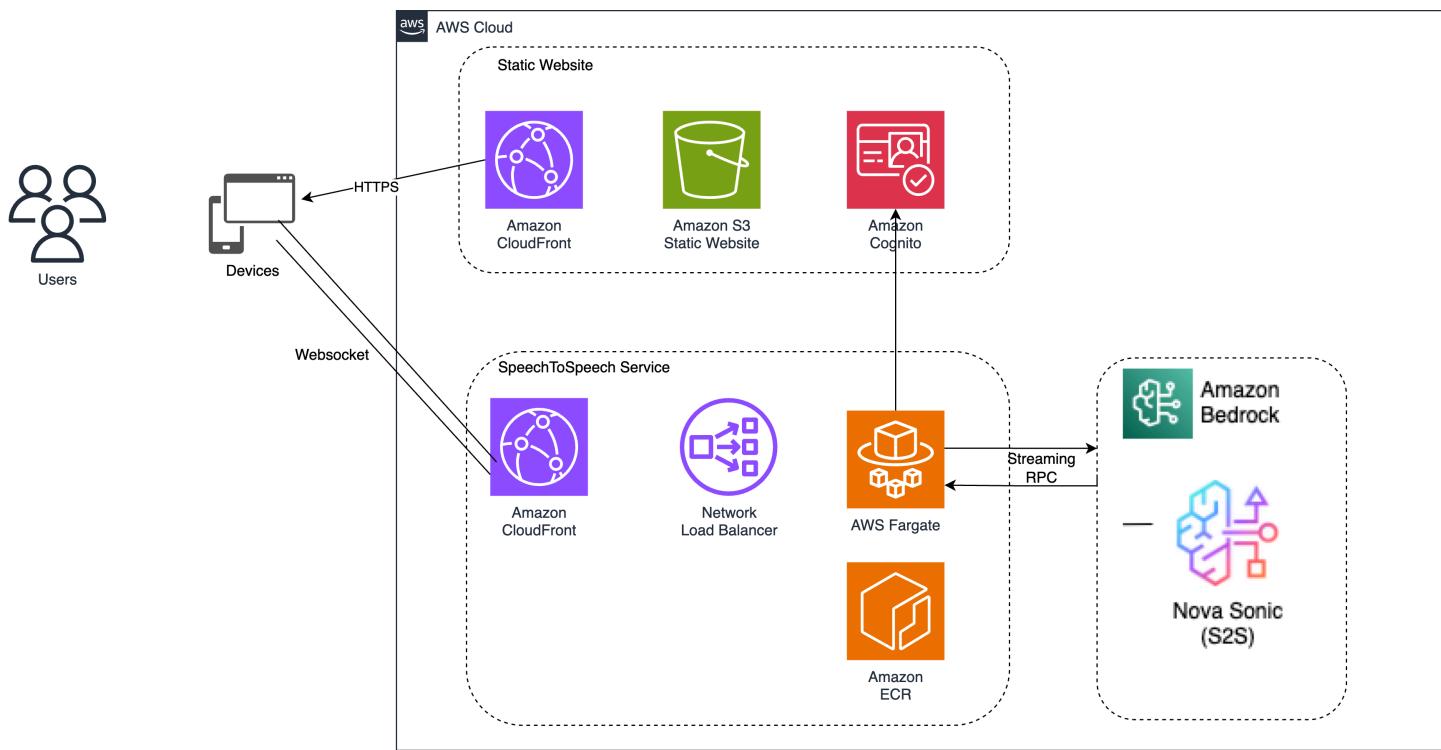
## Deploy on AWS

Deploy your Nova Sonic applications to AWS using infrastructure as code with AWS CDK (Cloud Development Kit). This approach provides repeatable, version-controlled deployments with best practices built in.

### Deployment options

- **Amazon ECS (Elastic Container Service):** Fully managed container orchestration with Application Load Balancer integration, auto-scaling, and serverless Fargate execution.
- **Amazon EKS (Elastic Kubernetes Services):** Managed Kubernetes for complex orchestration, advanced networking, multi-region deployments, and extensive tooling ecosystem.
- **AWS CDK:** AWS CDK allows you to define cloud infrastructure using familiar programming languages.

For a complete, production-ready example of deploying Nova Sonic with AWS CDK, see the [Speech-to-Speech CDK Sample](#) on GitHub. This sample demonstrates:



- Complete CDK infrastructure setup with TypeScript
- WebSocket server implementation for real-time communication
- Container deployment with ECS and Fargate

- Application Load Balancer configuration for WebSocket support
- VPC networking and security group setup
- CloudWatch monitoring and logging
- Best practices for production deployments

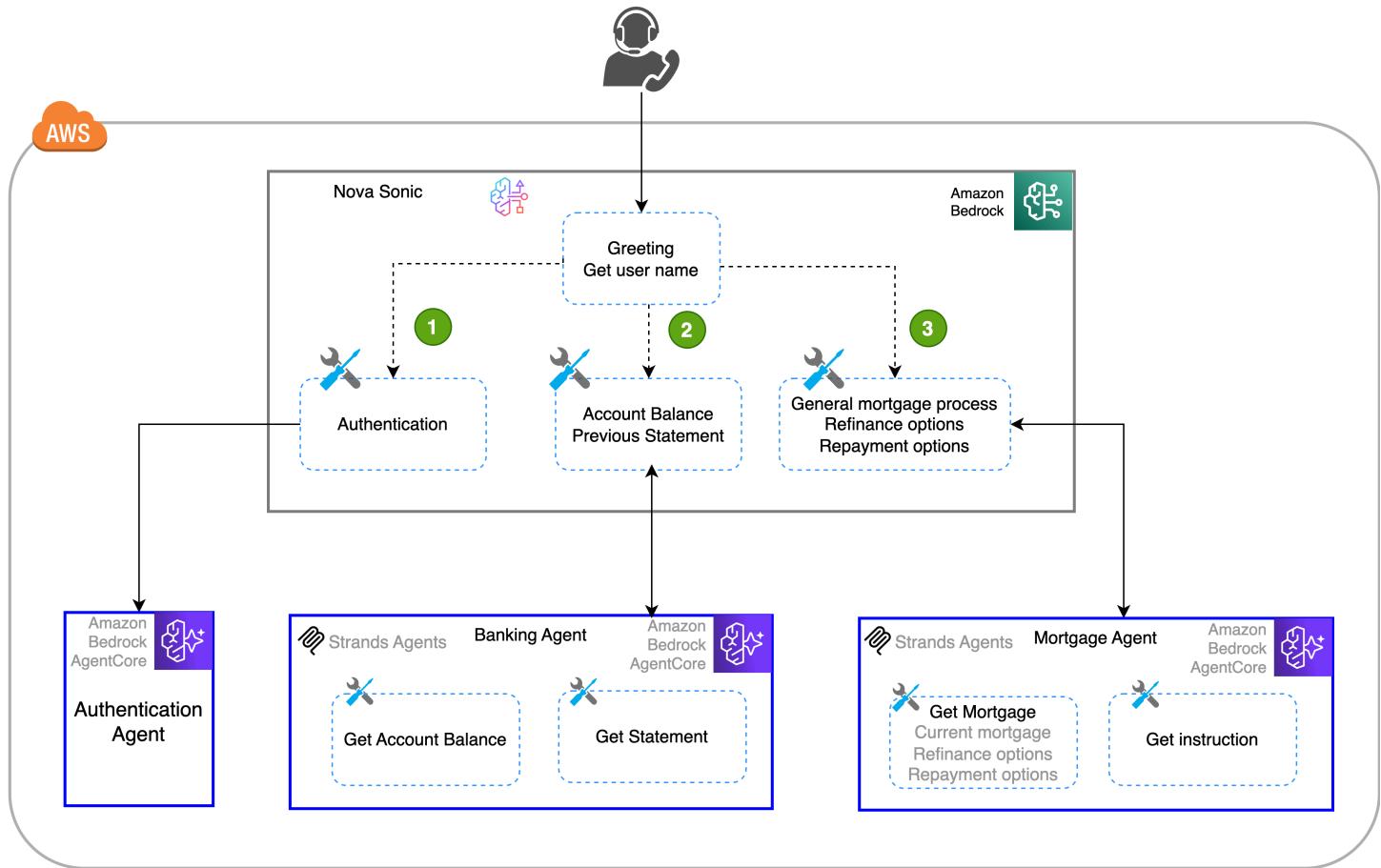
## Multi-agentic systems

Multi-agent architecture is a widely used pattern for designing AI assistants that handle complex tasks. In a voice assistant powered by Nova 2 Sonic, this architecture coordinates multiple specialized agents, where each agent operates independently to enable parallel processing, modular design, and scalable solutions.

Nova Sonic serves as the orchestrator in a multi-agent system, performing two key functions:

**Conversation flow management:** Ensures all necessary information is collected before proceeding to the next step in the conversation.

**Intent classification:** Analyzes user inquiries and routes them to the appropriate specialized sub-agent.



The diagram above shows a banking voice assistant that uses a multi-agent architecture. The conversation flow begins with a greeting and collecting the user's name, then handles inquiries related to banking or mortgages through specialized sub-agents.

Conversation flow example:

1. User connects to voice assistant.
2. Nova 2 Sonic: "Hello! What's your name?"
3. User: "My name is John"
4. Nova 2 Sonic: "Hi John, how can I help you today?"
5. User: "I want to check my account balance"
6. Nova 2 Sonic: [Routes to Authentication Agent]
7. Authentication Agent: "Please provide your account ID"
8. User: "12345"
9. Authentication Agent: [Verifies identity]

10Nova 2 Sonic: [Routes to Banking Agent]

11Banking Agent: "Your current balance is \$5,431,10"

While this example demonstrates sub-agents using the Strands Agents framework deployed on Amazon Bedrock AgentCore, the architecture is flexible. You can choose:

- Your preferred agent framework
- Any LLM provider
- Custom hosting options
- Different orchestration patterns

#### **Benefits:**

- Modularity: Each agent focuses on a specific domain, making the system easier to maintain and update.
- Scalability: Add new agents without modifying existing ones, allowing your system to grow with your needs.
- Parallel processing: Multiple agents can work simultaneously, improving response times for complex queries.
- Specialization: Each agent can be optimized for its specific task, using the most appropriate tools and knowledge bases.
- Fault isolation: If one agent fails, others continue to function, improving overall system reliability.

Refer to [this blog](#) for more details and code examples.

See the [Nova Sonic Workshop Multi-Agent Lab](#) for hands-on samples.

#### **Telephony integration**

Amazon Nova 2 Sonic integrates with telephony providers to enable AI-powered voice applications accessible via phone calls. This guide covers integration with Twilio, Vonage, and other SIP-based systems for building contact center solutions and voice agents.

**Twilio:** Cloud communications platform with programmable voice and media streaming capabilities.

**Vonage:** Global communications APIs with voice, WebSocket audio streaming, and SIP connectivity.

AWS provides a comprehensive sample implementation demonstrating Nova Sonic in a contact center environment with real-time analytics and telephony integration.

Repository: [Sample Sonic Contact Center with Telephony](#)

# Advanced systems with Amazon Nova

## Overview

This section provides comprehensive guidance for power users implementing advanced patterns with Nova models. These capabilities enable sophisticated applications including tool integration, retrieval-augmented generation (RAG), and autonomous AI agents.

## Extended thinking in Amazon Nova 2

Amazon Nova 2 Lite introduces **extended thinking** capabilities that enable the model to engage in deeper reasoning for complex problems. This optional feature gives you control over when and how the model allocates additional computational resources to think through challenging tasks.

### How Extended Thinking Works

Amazon Nova 2 introduces extended thinking as a **hybrid capability**. You have complete control:

- **Extended thinking OFF (default):** Amazon Nova 2 operates with efficient latent reasoning, optimal for everyday tasks and high-volume applications.
- **Extended thinking ON:** Amazon Nova 2 engages in explicit, step-by-step reasoning best for complex problems requiring deep analysis.

### Reasoning effort

The following shows different levels of reasoning effort.

#### Low Effort (`maxReasoningeffort:"low"`)

**Best for:** Tasks with added complexity requiring structured thinking. For example, you can use this for code review and improvement suggestions where the model needs to carefully consider existing code quality, analysis tasks that require thoughtful consideration of multiple factors, or problem-solving scenarios that benefit from a methodical approach. Low effort is ideal for compound tasks where basic reasoning improves accuracy without requiring deep multi-step planning.

## Medium effort (`maxReasoningEffort: "medium"`)

**Best for:** Multi-step tasks and coding workflows. For example, you can use this for software development and debugging where the model needs to understand existing code structure before implementing changes, code generation that requires coordination across multiple files or components, multi-step calculations with interdependencies, or planning tasks with multiple constraints. Medium effort is optimal for agentic workflows that coordinate multiple tools and require the model to maintain context across several sequential operations.

## High Effort (`maxReasoningeffort:"high"`)

**Best for:** STEM reasoning and advanced problem-solving. For example, you can use this for advanced mathematical problems and proofs that require rigorous step-by-step verification, scientific analysis and research tasks demanding deep investigation, complex system design with architectural considerations across multiple dimensions, or critical decision-making scenarios with significant implications. High effort delivers maximum accuracy for tasks requiring sophisticated reasoning, careful evaluation of alternatives, and thorough validation of conclusions.

## Quick start: Enabling Extended Thinking

Extended thinking is controlled through the `reasoningConfig` parameter.

```
import boto3

bedrock = boto3.client('bedrock-runtime', region_name='us-east-1')

response = bedrock.converse(
    modelId='us.amazon.nova-2-lite-v1:0',
    system=[{"text": "You are a highly capable personal assistant"}],
    messages=[{
        "role": "user",
        "content": [{"text": "Provide a meal plan for a gluten free family of 4."}]
    }],
    inferenceConfig={
        "temperature": 0.7,
        "topP": 0.9,
        "maxTokens": 10000
    },
    additionalModelRequestFields={
        "reasoningConfig": {
            "type": "enabled",
            "maxReasoningEffort": "low"
        }
    }
)
```

```
        }
    }

content_list = response["output"]["message"]["content"]

for item in content_list:
    if "reasoningContent" in item:
        reasoning_text = item["reasoningContent"]["reasoningText"]["text"]
        print("==== REASONING ===")
        print(reasoning_text)
        print()
    elif "text" in item:
        print("==== ANSWER ===")
        print(item["text"])
```

### Reasoning Parameters:

- **type:** enabled or disabled (default: disabled)
- **maxReasoningEffort:** low, medium, or high

#### Note

Temperature, topP and topK cannot be used with maxReasoningEffort set to high. Using these parameters together causes an error.

For full examples of code that utilizes extended thinking, see [the section called “Code library”](#).

## Response Structure

When you enable extended thinking, responses include reasoningContent blocks followed by text content blocks:

```
{
  "output": {
    "message": {
      "role": "assistant",
      "content": [
        {
```

```
        "reasoningContent": {
            "reasoningText": {
                "text": "[REDACTED]"
            }
        },
        {
            "text": "Based on the premises, we can conclude..."
        }
    ],
},
"stopReason": "end_turn"
}
```

### Note

With Amazon Nova 2, reasoning content displays as [REDACTED]. You're still charged for reasoning tokens as they contribute to improved output quality. We include this field in the response structure now to preserve the option of exposing reasoning content in the future. We are actively working with customers to determine the best approach for surfacing the model's reasoning process. Reasoning content is displayed as [REDACTED]. You are still charged for reasoning tokens as they contribute to improved response quality.

## Configuration Options

Amazon Nova 2 introduces a new `reasoningConfig` parameter that you can add to your existing converse request structure to enable reasoning:

```
additionalModelRequestFields={
    "reasoningConfig": {
        "type": "enabled", # or "disabled" (default)
        "maxReasoningEffort": "high" # "low", "medium", or "high"
    }
}
```

### Parameters:

- **type:** Toggle between "enabled" and "disabled" (default is "disabled")

- **maxReasoningEffort:** When enabled, control reasoning depth
- "low"": Moderately complex tasks
- "medium"": Complex problems requiring substantial analysis
- "high"": Most thorough reasoning for highly complex tasks

### Note

when using "high", temp, topP, and maxToken must be unset. In this mode, the model performs deeper analysis to find the best solution. This more thorough processing may generate output that exceeds 65k tokens. The exact amount depends on your request's complexity but for some problems we have seen it go up to 128K tokens. This ensures you get complete, high-quality reasoning rather than truncated results.

## Supported models

Extended thinking is currently available in: Amazon Nova 2 Lite ([us.amazon.nova-2-lite-v1:0](#))

## Extended thinking with tool use

Extended thinking works seamlessly with tool calling, allowing Amazon Nova to reason about which tools to use and how to interpret their results.

## Understanding Reasoning Tokens and Pricing

### Token types

Extended thinking tokens are billed as output tokens:

- **Input tokens:** Your original request content (standard input pricing)
- **Output tokens:** This includes reasoning tokens and the final visible response content (standard output pricing)

### Usage breakdown

All three token types are included in your usage metrics and billing. Reasoning tokens are priced the same as output tokens and will appear as "REDACTED" in the model response".

```
{  
  "usage": {  
    "inputTokens": 45,  
    "outputTokens": 1240,  
    "totalTokens": 1285  
  }  
}
```

## Frequently Asked Questions

Why does Amazon Nova 2 Lite show "[REDACTED]" for reasoning content instead of displaying the model's thinking process?

Our primary focus for this launch is ensuring Nova 2 delivers best in class intelligence for your tasks, and you'll see this reflected in the improved accuracy.

We recognize that visibility into the reasoning process is valuable, and we've heard strong customer interest in understanding how the model thinks through problems.

We are looking at ways to make this available soon.

**You are still billed for reasoning tokens**, as they represent actual work that improves your output quality

which will be captured in the outputTokens along with the answer tokens.

How do I know if extended thinking is working if reasoning is redacted?

You can confirm extended thinking is working by checking the output for reasoningContent blocks in the response (these only appear when reasoning is enabled)

## Using tools (function calling)

Tools extend Amazon Nova capabilities by connecting the model to external functionality such as APIs, databases and code execution environments. Tool use enables Amazon Nova to access real-time information, perform calculations and interact with external systems.

## Understanding the tool use workflow

Tool use with Amazon Nova involves three key phases:

### User Query and Tool Definition

Define tools by providing JSON schemas that describe each tool's functionality and input requirements. The tool configuration must include explicit details about when and how to use each tool.

### Tool Selection

When a user sends a message, Amazon Nova analyzes it to determine if a tool is necessary. This automatic tool choice examines the context and decides which tool (if any) to invoke. If Amazon Nova identifies a suitable tool, it returns the tool name and required parameters.

You are responsible for executing the tool based on the model's request. This means writing code that invokes the tool's functionality and processes the input parameters provided by the model.

### Return Results

After executing the tool, send the results back to Amazon Nova in a structured format using JSON or a combination of text and images. Amazon Nova incorporates the tool's output into the final response. If errors occur during execution, denote this in the tool response to allow Amazon Nova to adjust accordingly.

## Create a tool

Define tools using a tool configuration that includes an array of tools and optionally a tool choice parameter. Each tool specification must include:

- **Name:** Clear identifier for the tool
- **Description:** Concise explanation of the tool's functionality
- **Input Schema:** JSON schema defining required and optional parameters

### Example Tool configuration example

```
tool_config = {  
    "tools": [
```

```
{  
    "toolSpec": {  
        "name": "calculator",  
        "description": "A calculator tool that can execute a math equation",  
        "inputSchema": {  
            "json": {  
                "type": "object",  
                "properties": {  
                    "equation": {  
                        "type": "string",  
                        "description": "The full equation to evaluate"  
                    }  
                },  
                "required": ["equation"]  
            }  
        }  
    }  
}
```

## Best practices for tool definitions

- Ensure the name and description explicitly convey the tool's exact functionality; avoid tools that are overly semantically similar
- Include key differentiators in the description to help the model distinguish between similar tools
- Limit JSON schemas to two layers of nesting for best performance
- Constrain inputs using schema types (e.g., enum, int, float) rather than describing structure in plain text
- Denote required vs. optional parameters using JSON schema notation (e.g., "required": ["param1", "param2"])
- Validate your JSON schema using a standard validator before submitting
- Place long string arguments last in the schema and avoid nesting them

## Structured output with constrained decoding

Amazon Nova models leverage constrained decoding to ensure high reliability in generated outputs. This technique uses grammar to constrain possible tokens at each generation step, preventing invalid keys and enforcing correct data types based on your defined schema.

## Example Structured output example

```
tool_config = {
    "tools": [
        {
            "toolSpec": {
                "name": "ProductAnalysis",
                "description": "Analyze product information from text.",
                "inputSchema": {
                    "json": {
                        "type": "object",
                        "properties": {
                            "name": {
                                "type": "string",
                                "description": "Product name"
                            },
                            "rating": {
                                "maximum": 5,
                                "description": "Customer rating 1-5",
                                "type": ["number", "null"],
                                "minimum": 1
                            },
                            "features": {
                                "description": "Key product features",
                                "type": "array",
                                "items": {"type": "string"}
                            },
                            "category": {
                                "type": "string",
                                "description": "Product category"
                            },
                            "price": {
                                "type": "number",
                                "description": "Price in USD"
                            }
                        },
                        "required": ["name", "category", "price", "features"]
                    }
                }
            }
        ],
        "toolChoice": {
            "tool": {"name": "ProductAnalysis"}  
}
```

```
    }  
}
```

## Tool choice options

Amazon Nova supports three tool choice parameters:

Tool

The specified tool will be called once, ideal for structured output use cases

Any

One of the provided tools will be called at least once, useful for API selection scenarios

Auto

The model decides whether to call a tool and how many tools to call (default behavior)

## Call a tool

When Amazon Nova decides to call a tool, it returns a tool use block as part of the assistant message with stopReason set to "tool\_use". The tool block contains the tool name and its inputs.

### Note

Run following code sections sequentially (Calling a tool → Processing tool call → Returning tool results) in a single Python session. To run the examples again, restart your Python session.

## Example Calling a tool

```
import boto3  
import json  
  
# Create Bedrock client  
bedrock = boto3.client('bedrock-runtime', region_name='us-east-1')  
  
# Complex calculation that benefits from precise computation  
messages = [{  
    "role": "user",  
    "content": [{"
```

```
        "text": "Calculate the compound interest on $10,000 invested at 4.75% annual  
rate for 7 years, compounded quarterly. Use the formula A = P(1 + r/n)^(nt) where  
P=10000, r=0.0475, n=4, t=7"  
    }]  
}  
  
# Define tool configuration with calculator  
tool_config = {  
    "tools": [  
        {"toolSpec": {  
            "name": "calculator",  
            "description": "Perform mathematical calculations",  
            "inputSchema": {  
                "json": {  
                    "type": "object",  
                    "properties": {  
                        "expression": {  
                            "type": "string",  
                            "description": "Mathematical expression to evaluate"  
                        }  
                    },  
                    "required": ["expression"]  
                }  
            }  
        }  
    }  
}  
  
# Invoke Model  
response = bedrock.converse(  
    modelId="us.amazon.nova-2-lite-v1:0",  
    messages=messages,  
    toolConfig=tool_config  
)  
  
# Extract tool use from response  
tool = next(  
    block["toolUse"]  
    for block in response["output"]["message"]["content"]  
    if "toolUse" in block  
)  
  
print(f"Tool: {tool['name']}")
```

```
print(f"Expression: {tool['input']['expression']}")
```

## Processing tool calls

Extract the tool name and arguments from the message, then invoke the tool:

```
def calculate(expression):
    """Evaluate mathematical expression"""
    print(f"Calculating: {expression}")
    P = 10000
    r = 0.0475
    n = 4
    t = 7
    result = P * (1 + r/n) ** (n*t)
    return result

stop_reason = response["stopReason"]

if stop_reason == "tool_use":
    if tool["name"] == "calculator":
        result = calculate(tool["input"]["expression"])
```

## Returning tool results

Return tool results using the ToolResultBlock schema:

```
messages.append(response["output"]["message"])

# Add the tool result
messages.append({
    "role": "user",
    "content": [
        "toolResult": {
            "toolUseId": tool['toolUseId'],
            "content": [{"json": {"result": result}}],
            "status": "success"
        }
    ]
})

# Send the tool result to the model
response = bedrock.converse(
```

```
modelId="us.amazon.nova-2-lite-v1:0",
messages=messages,
toolConfig=tool_config
)

# Extract and display final response
final_text = next(
    block["text"]
    for block in response["output"]["message"]["content"]
    if "text" in block
)

print(f"\nFinal Response:\n{final_text}")
```

## Error handling

Report errors back to Amazon Nova to allow request modification and retry:

```
tool_result_message = {
    "role": "user",
    "content": [
        {
            "toolResult": {
                "toolUseId": tool["toolUseId"],
                "content": [{"text": "A validation exception occurred on field: sample.field"}],
                "status": "error"
            }
        }
    ]
}
```

## Security considerations

- Validate that tools exist before invoking them
- Ensure inputs are formatted correctly
- Verify appropriate permissions are in place before tool execution
- Rely on session details rather than allowing Amazon Nova to inject user information into tool calls
- Remember that LLMs can hallucinate tool calls; always validate before execution

## Built-in system tools

Amazon Nova 2.0 models include fully managed built-in tools that require no custom implementation. Enable these tools with a simple toggle in the Converse API.

### Code Interpreter

Code Interpreter allows Nova to securely execute Python code in isolated sandbox environments. This tool is designed for mathematical computations, logical operations, and iterative algorithms.

#### Note

Code Interpreter is available in the IAD, PDX, and NRT AWS Regions. To ensure your requests are routed to a supported Region, use Global CRIS. When using Bedrock API keys, you'll need to manually add InvokeTool permissions to the policy definitions. The default Bedrock role does not allow the InvokeTool action.

Enable Code Interpreter by specifying the systemTool parameter:

```
import boto3
import json

bedrock = boto3.client('bedrock-runtime', region_name='us-east-1')

tool_config = {
    "tools": [
        "systemTool": {
            "name": "nova_code_interpreter"
        }
    ]
}

response = bedrock.converse(
    modelId="us.amazon.nova-2-lite-v1:0",
    messages=[{
        "role": "user",
        "content": [{
            "text": "What is the average of 10, 24, 2, 3, 43, 52, 13, 68, 6, 7, 902,
82"
        }]
    }]
)
```

```
        },
        toolConfig=tool_config,
        inferenceConfig={"maxTokens": 10000, "temperature": 0}
    )

# Pretty print the response
for block in response["output"]["message"]["content"]:
    if "toolUse" in block:
        print("==== Tool Use ===")
        print(f"Tool: {block['toolUse']['name']}")
        print(f"Code:\n{block['toolUse']['input']['snippet']}\n")

    elif "toolResult" in block:
        print("==== Tool Result ===")
        result = block['toolResult']['content'][0]['json']
        print(f"Output: {result['stdOut']}")
        if result['stdErr']:
            print(f"Error: {result['stdErr']}")
        print(f"Exit Code: {result['exitCode']}\n")

    elif "text" in block:
        print("==== Final Answer ===")
        print(block["text"])
```

The interpreter runs code in a sandbox and returns results in a standard schema:

```
{
    "stdOut": "String",
    "stdErr": "String",
    "exitCode": "int",
    "isError": "boolean"
}
```

## Web Grounding

Web grounding enables Amazon Nova to access real-time information from the internet, providing up-to-date responses and reducing hallucinations. Enable by specifying the `nova_grounding` system tool:

```
tool_config = {
    "tools": [
        "systemTool": {"name": "nova_grounding"}
```

```
}]  
}
```

For detailed information about Web Grounding, see [the section called “Web Grounding”](#).

## Model Context Protocol (MCP)

The Model Context Protocol (MCP) is an open standard that enables secure, two-way connections between data sources and AI-powered tools. Instead of writing custom adapters for each API or service, run an MCP server and let Amazon Nova discover its tools automatically through a client bridge.

Once connected, Amazon Nova treats MCP tools like any other external integration: it decides when to call them, sends required parameters and incorporates results into responses. Using Amazon Nova with Strands makes this easier with a built-in MCPClient that manages discovery, connection and result mapping automatically.

## Building AI agents

Amazon Nova models are optimized for building AI agents with Amazon Nova Act. The models provide improved tool use, better reasoning for multi-step tasks, enhanced ability to maintain context across complex agent workflows and support for remote MCP tools.

### Create an agent

AI agents built with Nova can orchestrate multiple tool calls, maintain context across extended interactions and course-correct when needed. Extended thinking transforms agentic workflows by enabling systematic reasoning through complex goals. Consider using a planning framework SDK such as Strands Agents to make the planning and execution process of your agent systems more robust.

### Agent design patterns

When designing agents with Nova:

- Enable reasoning on medium or high for best results for complex multi-step workflows requiring planning and verification
- Implement tool choice auto to allow flexible tool selection across agent interactions

- Design error handling that allows agents to recover and retry with modified approaches
- Maintain conversation history to preserve context across agent interactions
- Implement robust content filtering and moderation mechanisms across uncontrolled content that your agent system consumes. For example, Amazon offers Amazon Bedrock Guardrails, a feature designed to apply safeguards across multiple foundation models, knowledge bases and agents. These guardrails can filter harmful content, block denied topics, and redact sensitive information such as personally identifiable information.

## Multi-tool agent example

```
tool_config = {
    # "tools": [
        #{ "toolSpec": {
            # "name": "calculator",
            # "description": "Perform mathematical calculations",
            # "inputSchema": {
                # "json": {
                    # "type": "object",
                    # "properties": {
                        # "expression": {
                            # "type": "string",
                            # "description": "Mathematical expression to evaluate"
                        }
                    },
                    # "required": ["expression"]
                }
            }
        },
        #},
        #{
            # "toolSpec": {
                # "name": "database_query",
                # "description": "Query financial database for historical data",
                # "inputSchema": {
                    # "json": {
                        # "type": "object",
                        # "properties": {
                            # "query": {
                                # "type": "string",
                                # "description": "SQL query to execute"
                            }
                        }
                    }
                }
            }
        }
    ]
}
```

```
#}
#},
#"required": ["query"]
#}
#}
#}
#
#]
}

response = client.converse(
#modelId=" us.amazon.nova-2-lite-v1:0",
#messages=[{
    #"role": "user",
    #"content": [
        {"text": "Analyze our Q3 financial performance across all business units, calculate year-over-year growth rates with statistical significance testing, and recommend budget allocation strategies for Q4."}
    ]
}],
#toolConfig=tool_config,
#inferenceConfig={"maxTokens": 10000, "temperature": 1, "topP": 0.9},
#additionalModelRequestFields={
    #"reasoningConfig": {
        #"type": "enabled",
        #"maxReasoningEffort": "low"
    }
}
)
```

## Invoke an agent

Agent invocation involves managing the conversation flow, processing tool calls, and maintaining state across multiple interactions.

# Streaming agent responses

Stream responses to provide real-time visibility into the agent's reasoning and actions:

```
import boto3  
  
response = client.converse_stream(  
    #modelId=" us.amazon.nova-2-lite-v1:0",  
    #messages=[{
```

```
#"role": "user",
#"content": [
    #text": "Design a scalable microservices architecture for an e-commerce
platform handling 1M+ daily transactions. Consider data consistency, fault tolerance,
performance, security, and cost optimization."
}]
#}],
#inferenceConfig={"maxTokens": 10000, "temperature": 10},
#additionalModelRequestFields={
    #"reasoningConfig": {
        #"type": "enabled",
        #"maxReasoningEffort": "low"
    }
}
#)

# Process the streaming response
reasoning_complete = False
for event in response["stream"]:
    #if "contentBlockDelta" in event:
        #delta = event["contentBlockDelta"]["delta"]

        #if "reasoningContent" in delta:
            #reasoning_text = delta["reasoningContent"]["reasoningText"]["text"]
            #print(f"{reasoning_text}", end="", flush=True)
        #elif "text" in delta:
            #if not reasoning_complete:
                #print(f"

Final Architecture Design:
")
        #reasoning_complete = True
        #print(f"{delta['text']}", end="", flush=True)
```

## Agent state management

Maintain conversation history and tool results to preserve context; the example below demonstrates this for a single turn, but the developer can determine how to orchestrate overall agent system based on workflow requirements. In addition, Amazon Web Services tools such as Strands manage agent context and tool state on behalf of the developer.

```
messages = []
```

```
messages = []

# Initial user query
messages.append({
    "role": "user",
    "content": [{"text": user_query}]
})

# Get agent response
response = client.converse(
    #modelId=" us.amazon.nova-2-lite-v1:0",
    #messages=messages,
    #toolConfig=tool_config,
    #inferenceConfig=inf_params
)

# Add assistant response to history
messages.append(response["output"]["message"])

# Process tool calls and add results
if response["stopReason"] == "tool_use":
    #tool = next(
    #    #block["toolUse"]
    #    #for block in response["output"]["message"]["content"]
    #    #if "toolUse" in block
    #)
    ## Execute tool
    #result = execute_tool(tool["name"], tool["input"])

    ## Add tool result to conversation
    messages.append({
        "role": "user",
        "content": [
            {"toolResult": {
                "toolUseId": tool["toolUseId"],
                "content": [{"json": result}],
                "status": "success"
            }}
        ]
    })
    ## Continue conversation
    #response = client.converse(
```

```
#modelId=" us.amazon.nova-2-lite-v1:0",
#messages=messages,
#toolConfig=tool_config,
#inferenceConfig=inf_params
#)
```

## Agent Best Practices

For more information about Agent Best Practices, see [the section called “Best practices”](#).

For guidance on developing Conversational AI agents, see [Speech-to-Speech](#).

## Web Grounding

Web Grounding enables Amazon Nova to search the web for current information and provide responses with citations. This feature is useful for queries requiring up-to-date information beyond the model's training data.

### How Web Grounding works

When Web Grounding is enabled for a prompt, the following steps are performed:

- 1. Request Configuration:** Your application sends a user prompt to the Amazon Bedrock API with nova\_grounding enabled as a systemTool.
- 2. Search & Analysis:** The model determines if search is needed, performs one or more searches for relevant information, and evaluates whether additional searches are required to expand its understanding or dive deeper on specific subtopics.
- 3. Response generation:** Amazon Nova automatically synthesizes information from search results to generate a final API response grounded in real-time information, complete with citations to its sources.

### How to use Web Grounding

For full examples of code that utilizes Web Grounding, see the [Code Samples](#) section.

To include Web Grounding in your results, specify the following systemTool parameter in your toolConfig block:

```
import boto3
```

```
from botocore.config import Config

# Create the Bedrock Runtime client with extended timeout
bedrock = boto3.client(
    "bedrock-runtime",
    region_name="us-east-1",
    config=Config(read_timeout=3600)
)

# Define the tool configuration
tool_config = {
    "tools": [
        "systemTool": {
            "name": "nova_grounding"
        }
    ]
}

# Send the request
response = bedrock.converse(
    modelId="us.amazon.nova-2-lite-v1:0",
    messages=[{
        "role": "user",
        "content": [{"text": "What are the latest developments in quantum computing?"}]
    }],
    toolConfig=tool_config
)

# Extract text with interleaved citations
output_with_citations = ""
content_list = response["output"]["message"]["content"]
for content in content_list:
    if "text" in content:
        output_with_citations += content["text"]
    elif "citationsContent" in content:
        citations = content["citationsContent"]["citations"]
        for citation in citations:
            url = citation["location"]["web"]["url"]
            output_with_citations += f" [{url}]"

print(output_with_citations)
```

## Regional availability

Web Grounding is currently only available in US regions and supported only by US CRIS profiles.

## Response structure

The following is an example response. The response has been shortened for brevity:

```
{  
  "output": {  
    "message": {  
      "content": [  
        {  
          "text": "Recent quantum computing developments include...",  
          "citationsContent": [  
            {  
              "location": {  
                "web": {  
                  "url": "https://example.com/quantum-news",  
                  "domain": "example.com"  
                }  
              }  
            }  
          ]  
        }  
      ]  
    }  
  }  
}
```

Each citation includes:

- **text**: A segment of the model's generated response.
- **citationsContent**: The primary container for the citation data related to a text segment.
- **citations**: A container within **citationsContent** that holds the location of a citation.
- **location**: A container within **citations** that holds the source of a citation.
- **web**: A container within **location** that holds the web source details.
- **url**: The full web address (URL) of the citation's source.
- **domain**: The root domain of the source url.

## Grounding safety

Your data never leaves AWS infrastructure. Model-generated queries stay within AWS services and are never sent to the broader internet. Our expansive internal web search index and knowledge graphs prioritize trustworthy and high-quality sources and filter malicious content on ingress. Finally, we protect your application against indirect prompt injection and misinformation with runtime filtering (note that this mitigation is limited for non-English languages).

## Error handling

Do not include a toolSpec entry with the name nova\_grounding. Including a tool with this name will result in an error.

The following is a list of potential errors that can occur when using Web Grounding:

- malformed\_tool\_use
- max\_tokens
- malformed\_model\_output

## Permissions required for built in tools

To ensure your role can access Web Grounding on Amazon Bedrock, you have two options:

1. **Enable BedrockFullAccess on your IAM role:** If your role has BedrockFullAccess, it will have automatic access to Web Grounding.
2. **Add Specific Permissions (if needed):** If you require more granular access control, add this policy to your role's IAM policy, replacing the account ID with your AWS account ID:

```
{  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": ["bedrock:InvokeTool"],  
            "Resource": ["arn:aws:bedrock:{111122223333}:system-tool/  
amazon.nova_grounding"]  
        }  
    ]  
}
```

Web Grounding has the `aws:requestedRegion` condition key set to "unspecified". If your existing policies or Service Control Policies (SCPs) enforce this condition, you may encounter access issues. Updating the condition to allow an "unspecified" requestedRegion can resolve this problem.

 **Note**

If you enable the Web Grounding tool, you are responsible for your use, and any use by your end users, of output that incorporates grounded information. You will know when your output includes grounded information from citations or links to the source material. You must retain and display these citations and links in the output you provide to your end users.

 **Note**

Web Grounding is an additional cost. For more information, go to [AWS Bedrock pricing](#).

# Customizing Amazon Nova 2.0 models

Amazon Nova offers the most comprehensive suite of customization options to adapt the foundation models to your specific business needs, from simple fine-tuning in Amazon Bedrock to advanced training pipelines in Amazon SageMaker.

## Customization overview

Model customization allows you to specialize Amazon Nova models for your domain, use cases and quality requirements. You can choose from several customization techniques and platforms based on your technical requirements, data availability and desired outcomes.

Customization techniques:

- **Continued Pre-Training (CPT)** - Teach models domain-specific knowledge using raw text data
- **Supervised Fine-Tuning (SFT)** - Customize through input-output examples
- **Reinforcement Fine-Tuning (RFT)** - Optimize using reward signals and human feedback
- **Distillation** - Transfer knowledge from larger to smaller models

## Customization platform availability

The following table summarizes the availability of customizations per platform.

Platform	CPT	SFT	RFT	Distillation
Amazon Bedrock	No	Yes	Yes	Yes
SageMaker AI Training Jobs	No	Yes	Yes	Yes
SageMaker AI HyperPod	Yes	Yes	Yes	Yes

# Customization on Amazon Bedrock

Amazon Bedrock provides a fully managed fine-tuning experience for Amazon Nova models, making it easy to customize models without managing infrastructure.

Supported methods:

Supervised Fine-Tuning (SFT)

Teach models through input-output examples to customize response style, format and task-specific behavior.

Reinforcement Fine-Tuning (RFT)

Maximize accuracy and align the model with real-world feedback and simulations using reward signals.

Model Distillation

Transfers knowledge from larger "teacher" models to smaller "student" models. This process creates efficient models that maintain a significant portion of the original model's performance. The teacher model generates responses to diverse prompts and these outputs train the student model to produce similar results. This approach is more effective than standard fine-tuning when you lack sufficient high-quality labeled data.

 **Note**

For implementation details, see [Customization on Amazon Bedrock](#).

Key features:

- Fully managed infrastructure with no setup required
- Simple API-based training job submission
- Direct deployment to Amazon Bedrock inference endpoints

When to use Amazon Bedrock fine-tuning:

- You need quick customization with minimal setup

- Your use case fits standard fine-tuning patterns
- You prefer flexible customization from simple to increasingly complex training
- You want seamless integration with Amazon Bedrock inference

## Customization on SageMaker AI

SageMaker AI provides advanced training capabilities when you need full control over the customization process, access to multiple training methods and the ability to build simple to increasingly complex training pipelines.

Available training methods:

### Continued Pre-Training (CPT)

Teaches models domain-specific knowledge at scale using raw text data. Ideal for specialized technical fields, legal documents, medical literature, or any domain with unique terminology and concepts. Requires large volumes of unlabeled text (billions of tokens recommended).

### Supervised Fine-Tuning (SFT)

Customizes models through direct input-output examples. Best for teaching specific response styles, formats and task behaviors. Supports text, image and video inputs. You can choose between Parameter Efficient Fine-Truning (PEFT) or Full Fine-Tuning methods. The prior is suited for adapter-based customization with 2k-10k samples. The latter is suited for full model customization with tens of thousands to hundreds of thousands of samples.

### Reinforcement Fine-Tuning (RFT)

Optimizes models using reward signals for complex problem-solving tasks like domain-specific reasoning, code generation, and scientific analysis. Best used after SFT establishes baseline capabilities. You can use regex patterns and custom python code for verifiable rewards, or LLM-as-a-Judge for non-verifiable rewards. You can also integrate any other custom reward functions through a Lambda function such as single-turn remote reward functions in a customer's own environment.

### Model Distillation

Transfers knowledge from larger "teacher" models to smaller "student" models. This process creates efficient models that maintain a significant portion of the original model's performance.

The teacher model generates responses to diverse prompts and these outputs train the student model to produce similar results. This approach is more effective than standard fine-tuning when you lack sufficient high-quality labeled data.

 **Note**

For implementation details on distillation, see [Model distillation](#).

Advanced capabilities:

- **Iterative training** - Chain multiple training methods (for example, SFT to RFT) with checkpoint reuse for targeted improvements
- **Reasoning Mode Support** - Train Nova 2 models with explicit reasoning steps for complex analytical tasks

Infrastructure options:

SageMaker AI Training Jobs

Managed training with automatic resource provisioning for streamlined model customization workflows.

SageMaker AI HyperPod

Resilient, large-scale training clusters for enterprise workloads requiring maximum control and scale.

## Fine-tuning Amazon Nova models

You can customize the Amazon Nova models using the [fine-tuning method](#) with labeled proprietary data on Amazon Bedrock to gain more performance for your use case than the models provide out-of-the-box. That is, fine-tuning provides enhancements beyond what is gained with zero- or few-shot invocation and other prompt engineering techniques. You can fine-tune Amazon Nova models when a sufficient amount of high-quality, labeled training data that is available for the following use cases:

- When you have a niche or specialized tasks in a specific domain.

- When you want model outputs aligned with brand tone, company policies, or proprietary workflows.
- When you need better results across a wide number of tasks and thus need to introduce examples in training. This situation is in contrast to providing instructions and examples in prompts, which also impacts token cost and request latency.
- When you have tight latency requirements and can benefit from smaller models that are tailored to a specific use case.

## Available models

Fine-tuning is available for the Nova 2 Lite Amazon Nova 2.0 model and their supported text, image and video modalities.

## On-demand inference on Custom Models

On-demand (OD) inference allows you to run inference on your custom Amazon Nova models without maintaining provisioned throughput endpoints. This helps you optimize costs and scale efficiently. With On-demand inference, you are charged based on usage, measured in tokens, both in and out.

## Compatibility requirements

The following compatibility requirements apply:

- OD inference is supported for Amazon Nova Pro, Lite and Micro custom understanding models. OD inference is not supported for Nova custom content generation models.
- OD inference is supported for Amazon Nova custom understanding models trained after July 16, 2025. Custom models trained before July 16, 2025 are not compatible with OD inference.
- Amazon Bedrock customization: OD inference is supported for models customized with Amazon Bedrock customization and for student models that were distilled from a teacher model with Amazon Bedrock.
- SageMaker AI customization: For models customized in SageMaker AI, OD inference is supported only for Parameter-efficient fine-tuned (PEFT) models when the model is hosted on Amazon Bedrock. This includes Direct Preference Optimization plus PEFT. OD inference is not supported for Full-rank fine-tuned models.

## Model training and inference

When you train a new custom Amazon Nova Pro, Lite, or Micro model on Amazon Bedrock or SageMaker AI using PEFT after July 16, 2025, the model will automatically be compatible with both provisioned and on-demand inference options. You can select your preferred inference method when you deploy your model.

To use OD inference with a model trained after July 16, 2025, complete the following steps:

1. Create a new fine-tuning job with either the [Amazon Bedrock customization API](#) or the [SageMaker AI customization API](#).
2. Deploy the newly trained model to Amazon Bedrock using the [CreateCustomModel API](#).
3. Deploy for on-demand inference using the CustomModelDeployment API.

## Rate limits

The following requests per minute (RPM) and tokens per minute (TPM) limits apply to on-demand inference requests:

Base Model for Custom Model	RPM per Custom Model Deployment	TPM per Custom Model Deployment
Nova 2 Lite	2,000	4,000,000

To learn more about the quotas available for Amazon Nova, see [Quotas for Amazon Nova](#).

## Latency

You can expect an end-to-end latency difference (that is, Time To First Token (TTFT)) of 20-55% between the base model invocation and the adapter. The exact latency value varies by model size and is in line with industry standards.

# Code and Troubleshooting

This chapter provides code examples and troubleshooting guidance to help you work effectively with Amazon Nova models.

## Code library

This section provides code examples for common Amazon Nova operations using either the Converse API or the InvokeModel API.

### Converse API Examples

#### Basic request

Send a basic text request to Amazon Nova models using the Converse API.

Non-streaming

```
import boto3
from botocore.config import Config

# Create the Bedrock Runtime client, using an extended timeout configuration
# to support long-running requests.
bedrock = boto3.client(
    "bedrock-runtime",
    region_name="us-east-1",
    config=Config(read_timeout=3600),
)

# Invoke the model
response = bedrock.converse(
    modelId="us.amazon.nova-2-lite-v1:0",
    messages=[
        {
            "role": "user",
            "content": [{"text": "Write a short story. End the story with 'THE\nEND'."}],
        }
    ],
    system=[{"text": "You are a children's book author."}], # Optional
```

```
inferenceConfig={ # These parameters are optional
    "maxTokens": 1500,
    "temperature": 0.7,
    "topP": 0.9,
    "stopSequences": ["THE END"],
},
additionalModelRequestFields={ # These parameters are optional
    "inferenceConfig": {
        "topK": 50,
    }
},
)

# Extract the text response
content_list = response["output"]["message"]["content"]
for content in content_list:
    if "text" in content:
        print(content["text"])
```

## Streaming

```
import boto3
from botocore.config import Config

# Create the Bedrock Runtime client, using an extended timeout configuration
# to support long-running requests.
bedrock = boto3.client(
    "bedrock-runtime",
    region_name="us-east-1",
    config=Config(connect_timeout=3600, read_timeout=3600),
)

# Invoke the model
response = bedrock.converse_stream(
    modelId="us.amazon.nova-2-lite-v1:0",
    messages=[
        {
            "role": "user",
            "content": [{"text": "Write a short story. End the story with 'THE END'."}],
        }
    ],
    system=[{"text": "You are a children's book author."}], # Optional
```

```
inferenceConfig={ # These parameters are optional
    "maxTokens": 1500,
    "temperature": 0.7,
    "topP": 0.9,
    "stopSequences": ["THE END"],
},
additionalModelRequestFields={ # These parameters are optional
    "inferenceConfig": {
        "topK": 50,
    }
},
)

# Handle streaming events
for event in response["stream"]:
    if "contentBlockDelta" in event:
        delta = event["contentBlockDelta"]["delta"]
        if "text" in delta:
            print(delta["text"], end="", flush=True)
```

## Multimodal input using embedded asset

Process multimodal content by embedding document, image, video, or audio data directly in the request. This example uses image data. For details on the content structure for other modalities, see the [ContentBlock details in the Amazon Bedrock API documentation](#).

### Non-streaming

```
import boto3
from botocore.config import Config

# Read a document, image, video, or audio file
with open("sample_image.png", "rb") as image_file:
    binary_data = image_file.read()
    data_format = "png"

# Define message with image
messages = [
{
    "role": "user",
    "content": [
        {
```

```
        "image": {
            "format": data_format,
            "source": {
                "bytes": binary_data # For Invoke API, encode as Base64
        string
            },
            },
            ],
            {"text": "Provide a brief caption for this asset."},
        ],
    }
]

# Create the Bedrock Runtime client, using an extended timeout configuration
# to support long-running requests.
bedrock = boto3.client(
    "bedrock-runtime",
    region_name="us-east-1",
    config=Config(read_timeout=3600),
)

# Invoke model
response = bedrock.converse(
    modelId="us.amazon.nova-2-lite-v1:0",
    messages=messages,
)

# Extract the text response
content_list = response["output"]["message"]["content"]
for content in content_list:
    if "text" in content:
        print(content["text"])
```

## Streaming

```
import boto3
from botocore.config import Config

# Read a document, image, video, or audio file
with open("sample_image.png", "rb") as image_file:
    binary_data = image_file.read()
    data_format = "png"
```

```
# Define message with image
messages = [
    {
        "role": "user",
        "content": [
            {
                "image": {
                    "format": data_format,
                    "source": {
                        "bytes": binary_data # For Invoke API, encode as Base64
string
                    },
                },
            ],
            {"text": "Provide a brief caption for this asset."},
        ],
    }
]

# Create the Bedrock Runtime client, using an extended timeout configuration
# to support long-running requests.
bedrock = boto3.client(
    "bedrock-runtime",
    region_name="us-east-1",
    config=Config(connect_timeout=3600, read_timeout=3600),
)

# Invoke model with streaming
response = bedrock.converse_stream(
    modelId="us.amazon.nova-2-lite-v1:0",
    messages=messages,
)

# Handle streaming events
for event in response["stream"]:
    if "contentBlockDelta" in event:
        delta = event["contentBlockDelta"]["delta"]
        if "text" in delta:
            print(delta["text"], end="", flush=True)
```

## Multimodal input using S3 URI

Process multimodal content by referencing documents, images, videos, or audio files stored in S3. This example uses an image reference. For details on the content structure for other modalities, see the [ContentBlock details in the Amazon Bedrock API documentation](#).

### Non-streaming

```
import boto3
from botocore.config import Config

# Define message with image
messages = [
    {
        "role": "user",
        "content": [
            {
                "image": {
                    "format": "png",
                    "source": {
                        "s3Location": {
                            "uri": "s3://path/to/your/asset",
                            # "bucketOwner": "<account_id>" # Optional
                        }
                    },
                },
            ],
            {"text": "Provide a brief caption for this asset."},
        ],
    }
]

# Create the Bedrock Runtime client, using an extended timeout configuration
# to support long-running requests.
bedrock = boto3.client(
    "bedrock-runtime",
    region_name="us-east-1",
    config=Config(read_timeout=3600),
)

# Invoke model
response = bedrock.converse(
    modelId="us.amazon.nova-2-lite-v1:0",
```

```
    messages=messages,
)

# Extract the text response
content_list = response["output"]["message"]["content"]
for content in content_list:
    if "text" in content:
        print(content["text"])
```

## Streaming

```
import boto3
from botocore.config import Config

# Define message with image
messages = [
{
    "role": "user",
    "content": [
        {
            "image": {
                "format": "png",
                "source": {
                    "s3Location": {
                        "uri": "s3://path/to/your/asset",
                        # "bucketOwner": "<account_id>" # Optional
                    }
                },
            },
        },
        {"text": "Provide a brief caption for this asset."},
    ],
},
]

# Create the Bedrock Runtime client, using an extended timeout configuration
# to support long-running requests.
bedrock = boto3.client(
    "bedrock-runtime",
    region_name="us-east-1",
    config=Config(connect_timeout=3600, read_timeout=3600),
)
```

```
# Invoke model with streaming
response = bedrock.converse_stream(
    modelId="us.amazon.nova-2-lite-v1:0",
    messages=messages,
)

# Handle streaming events
for event in response["stream"]:
    if "contentBlockDelta" in event:
        delta = event["contentBlockDelta"]["delta"]
        if "text" in delta:
            print(delta["text"], end="", flush=True)
```

## Extended thinking (reasoning)

Enable extended thinking for complex problem-solving tasks.

### Non-streaming

```
import boto3
from botocore.config import Config

# Create the Bedrock Runtime client, using an extended timeout configuration
# to support long-running requests.
bedrock = boto3.client(
    "bedrock-runtime",
    region_name="us-east-1",
    config=Config(read_timeout=3600),
)

# Invoke the model
response = bedrock.converse(
    modelId="us.amazon.nova-2-lite-v1:0",
    messages=[
        {
            "role": "user",
            "content": [
                {
                    "text": 'How many capital letters appear in the following passage. Your response must include only the number: "Wilfred ordered an anvil from ACME. Shipping was expensive."'
                }
            ]
        }
    ]
)
```

```
        ],
    }
],
additionalModelRequestFields={
    "reasoningConfig": {
        "type": "enabled",
        "maxReasoningEffort": "low", # "low" | "medium" | "high"
    }
},
)

# Extract response content
content_list = response["output"]["message"]["content"]
for content in content_list:
    # Extract the reasoning response
    if "reasoningContent" in content:
        print("\n== Reasoning ==")
        print(content["reasoningContent"]["reasoningText"]["text"])
    # Extract the text response
    if "text" in content:
        print("\n== Text ==")
        print(content["text"])
```

## Streaming

```
import boto3
from botocore.config import Config

# Create the Bedrock Runtime client, using an extended timeout configuration
# to support long-running requests.
bedrock = boto3.client(
    "bedrock-runtime",
    region_name="us-east-1",
    config=Config(connect_timeout=3600, read_timeout=3600),
)

# Invoke the model
response = bedrock.converse_stream(
    modelId="us.amazon.nova-2-lite-v1:0",
    messages=[
        {
            "role": "user",
            "content": [

```

```
        {
            "text": 'How many capital letters appear in the following passage. Your response must include only the number: "Wilfred ordered an anvil from ACME. Shipping was expensive."
        }
    ],
},
],
additionalModelRequestFields={
    "reasoningConfig": {
        "type": "enabled",
        "maxReasoningEffort": "low", # "low" | "medium" | "high"
    },
},
)

# Process the streaming response
reasoning_output = ""
text_output = ""
for event in response["stream"]:
    if "contentBlockDelta" in event:
        delta = event["contentBlockDelta"]["delta"]

        if "reasoningContent" in delta:
            if len(reasoning_output) == 0:
                print("\n\n== Reasoning ==")
            reasoning_text_chunk = delta["reasoningContent"]["text"]
            print(reasoning_text_chunk, end="", flush=True)
            reasoning_output += reasoning_text_chunk

        elif "text" in delta:
            if len(text_output) == 0:
                print("\n\n== Text ==")
            text_chunk = delta["text"]
            print(text_chunk, end="", flush=True)
            text_output += text_chunk
```

## Built-in tool: Nova Grounding with citations

Use Nova Grounding to retrieve real-time information from the web with citations.

## Non-streaming

```
import boto3
from botocore.config import Config

# Define the list of tools the model may use
tool_config = {"tools": [{"systemTool": {"name": "nova_grounding"}}]}

# Create the Bedrock Runtime client, using an extended timeout configuration
# to support long-running requests.
bedrock = boto3.client(
    "bedrock-runtime",
    region_name="us-east-1",
    config=Config(read_timeout=3600),
)

messages = [
    {
        "role": "user",
        "content": [
            {"text": "What is the latest news about renewable energy sources?"}
        ],
    }
]

# Invoke the model
response = bedrock.converse(
    modelId="us.amazon.nova-2-lite-v1:0", messages=messages, toolConfig=tool_config
)

# Extract the text with interleaved citations
output_with_citations = ""
content_list = response["output"]["message"]["content"]
for content in content_list:
    if "text" in content:
        output_with_citations += content["text"]

    elif "citationsContent" in content:
        citations = content["citationsContent"]["citations"]
        for citation in citations:
            url = citation["location"]["web"]["url"]
            output_with_citations += f"[{url}]"
```

```
print(output_with_citations)
```

## Streaming

```
import boto3
from botocore.config import Config

# Define the list of tools the model may use
tool_config = {"tools": [{"systemTool": {"name": "nova_grounding"}}]}

# Create the Bedrock Runtime client, using an extended timeout configuration
# to support long-running requests.
bedrock = boto3.client(
    "bedrock-runtime",
    region_name="us-east-1",
    config=Config(read_timeout=3600),
)

messages = [
    {
        "role": "user",
        "content": [
            {"text": "What is the latest news about renewable energy sources?"}
        ],
    }
]

# Invoke the model with streaming
response = bedrock.converse_stream(
    modelId="us.amazon.nova-2-lite-v1:0", messages=messages, toolConfig=tool_config
)

# Process the streaming response with interleaved citations
for event in response["stream"]:
    if "contentBlockDelta" in event:
        delta = event["contentBlockDelta"]["delta"]

        if "text" in delta:
            print(delta["text"], end="", flush=True)

        elif "citation" in delta:
            url = delta["citation"]["location"]["web"]["url"]
            print(f"[{url}]", end="", flush=True)
```

## Built-in tool: Code Interpreter

Use the Code Interpreter tool to execute Python code for calculations and data analysis.

### Non-streaming

```
import boto3
from botocore.config import Config

# Define the list of tools the model may use
tool_config = {"tools": [{"systemTool": {"name": "nova_code_interpreter"}}]}

# Create the Bedrock Runtime client, using an extended timeout configuration
# to support long-running requests.
bedrock = boto3.client(
    "bedrock-runtime",
    region_name="us-east-1",
    config=Config(read_timeout=3600),
)

messages = [
    {
        "role": "user",
        "content": [
            {
                "text": "What is the average of 10, 24, 2, 3, 43, 52, 13, 68, 6, 7, 902, 82?"
            }
        ],
    }
]

# Invoke the model
response = bedrock.converse(
    modelId="us.amazon.nova-2-lite-v1:0", messages=messages, toolConfig=tool_config
)

# Extract the text and the code the was executed
content_list = response["output"]["message"]["content"]
for content in content_list:
    if "text" in content:
        print("\n== Text ==")
        print(content["text"])
```

```
    elif "toolUse" in content and content["toolUse"]["name"] ==  
        "nova_code_interpreter":  
            print("\n== Code Interpreter: input.snippet ==")  
            print(content["toolUse"]["input"]["snippet"])
```

## Streaming

```
import boto3  
from botocore.config import Config  
import json  
  
# Define the list of tools the model may use  
tool_config = {"tools": [{"systemTool": {"name": "nova_code_interpreter"}}]}  
  
messages = [  
    {  
        "role": "user",  
        "content": [  
            {  
                "text": "What is the average of 10, 24, 2, 3, 43, 52, 13, 68, 6, 7,  
902, 82?"  
            }  
        ],  
    }  
]  
  
# Create the Bedrock Runtime client, using an extended timeout configuration  
# to support long-running requests.  
bedrock = boto3.client(  
    "bedrock-runtime",  
    region_name="us-east-1",  
    config=Config(connect_timeout=3600, read_timeout=3600),  
)  
  
# Invoke the model with streaming  
response = bedrock.converse_stream(  
    modelId="us.amazon.nova-2-lite-v1:0", messages=messages, toolConfig=tool_config  
)  
  
# Process the streaming response  
current_block_start = None  
response_text = ""  
for event in response["stream"]:
```

```
if "contentBlockStart" in event:  
    current_block_start = event["contentBlockStart"]["start"]  
  
elif "contentBlockStop" in event:  
    current_block_start = None  
  
elif "contentBlockDelta" in event:  
    delta = event["contentBlockDelta"]["delta"]  
  
    if (  
        current_block_start  
        and "toolUse" in current_block_start  
        and current_block_start["toolUse"]["name"] == "nova_code_interpreter"  
    ):  
        # This is code interpreter content  
        tool_input = json.loads(delta["toolUse"]["input"])  
        print("\n== Executed Code Snippet ==")  
        print(tool_input["snippet"], end="", flush=True)  
  
    elif "text" in delta:  
        # This is text response content  
        if len(response_text) == 0:  
            print("\n== Text ==")  
        text = delta["text"]  
        response_text += text  
        print(text, end="", flush=True)
```

## Tool use

Define custom tools for the model to use during conversation.

Non-streaming

```
import boto3  
from botocore.config import Config  
  
def get_weather(city):  
    # Mock function to simulate weather API  
    return {"temperatureF": 48, "conditions": "light rain"}
```

```
# Define the toolSpec for the weather tool
weather_tool = {
    "toolSpec": {
        "name": "get_weather",
        "description": "Get the current weather conditions in a given location",
        "inputSchema": {
            "json": {
                "type": "object",
                "properties": {
                    "city": {
                        "type": "string",
                        "description": "The city and state, e.g. San Francisco, CA",
                    }
                },
                "required": ["city"],
            }
        },
    },
}

# Define the list of tools the model may use
tool_config = {"tools": [weather_tool]}

# Create the Bedrock Runtime client, using an extended timeout configuration
# to support long-running requests.
bedrock = boto3.client(
    "bedrock-runtime",
    region_name="us-east-1",
    config=Config(read_timeout=3600),
)

# Start tracking message history
messages = []

messages.append(
{
    "role": "user",
    "content": [
        {
            "text": "Suggest some activities to do in Seattle based on the
current weather."
        }
    ],
}
```

```
)\n\n# Invoke the model\nresponse = bedrock.converse(\n    modelId="us.amazon.nova-2-lite-v1:0", messages=messages, toolConfig=tool_config\n)\n\nassistant_message = response["output"]["message"]\n\n# Add the assistant response to the message history\nmessages.append(assistant_message)\n\ncontent_list = assistant_message["content"]\nstop_reason = response["stopReason"]\n\nif stop_reason == "tool_use":\n    # Extract the toolUse details\n    tool_use = next(\n        content["toolUse"] for content in content_list if "toolUse" in content\n    )\n    tool_name = tool_use["name"]\n    tool_use_id = tool_use["toolUseId"]\n\n    if tool_name == "get_weather":\n        # Call the tool\n        weather = get_weather(tool_use["input"]["city"])\n\n        # Send the result back to the model\n        messages.append(\n            {\n                "role": "user",\n                "content": [\n                    {\n                        "toolResult": {\n                            "toolUseId": tool_use_id,\n                            "content": [{"json": weather}],\n                        }\n                    }\n                ],\n            }\n        )\n\n    # Submit the tool result back to the model\n    response = bedrock.converse(
```

```
        modelId="us.amazon.nova-2-lite-v1:0",
        messages=messages,
        toolConfig=tool_config,
    )

    content_list = response["output"]["message"]["content"]
    for content in content_list:
        # Extract the text response
        if "text" in content:
            print("\n== Text ==")
            print(content["text"])
else:
    # A tool call was not needed
    for content in content_list:
        # Extract the text response
        if "text" in content:
            print("\n== Text ==")
            print(content["text"])
```

## Streaming

```
import boto3
from botocore.config import Config
import json

def get_weather(city):
    # Mock function to simulate weather API
    return {"temperatureF": 48, "conditions": "light rain"}

# Define the toolSpec for the weather tool
weather_tool = {
    "toolSpec": {
        "name": "get_weather",
        "description": "Get the current weather conditions in a given location",
        "inputSchema": {
            "json": {
                "type": "object",
                "properties": {
                    "city": {
                        "type": "string",
                        "description": "The city and state, e.g. San Francisco, CA",
                    }
                }
            }
        }
    }
}
```

```
        }
    },
    "required": ["city"],
}
},
}
}

# Define the list of tools the model may use
tool_config = {"tools": [weather_tool]}

# Create the Bedrock Runtime client, using an extended timeout configuration
# to support long-running requests.
bedrock = boto3.client(
    "bedrock-runtime",
    region_name="us-east-1",
    config=Config(read_timeout=3600),
)

# Start tracking message history
messages = []

messages.append(
{
    "role": "user",
    "content": [
        {
            "text": "Suggest some activities to do in Seattle based on the
current weather."
        }
    ],
}
)

# Invoke the model with streaming
response = bedrock.converse_stream(
    modelId="us.amazon.nova-2-lite-v1:0", messages=messages, toolConfig=tool_config
)

# Process the streaming response
assistant_message = {"role": "assistant", "content": []}
current_tool_use = None
stop_reason = None
```

```
for event in response["stream"]:
    if "contentBlockStart" in event:
        start = event["contentBlockStart"]["start"]
        if "toolUse" in start:
            current_tool_use = start["toolUse"]
            current_tool_use["input"] = ""

    elif "contentBlockDelta" in event:
        delta = event["contentBlockDelta"]["delta"]
        if "toolUse" in delta:
            current_tool_use["input"] += delta["toolUse"]["input"]
        elif "text" in delta:
            print(delta["text"], end="", flush=True)

    elif "contentBlockStop" in event:
        if current_tool_use:
            # Parse the accumulated tool input
            current_tool_use["input"] = json.loads(current_tool_use["input"])
            assistant_message["content"].append({"toolUse": current_tool_use})
            current_tool_use = None

    elif "messageStop" in event:
        stop_reason = event["messageStop"]["stopReason"]
        if stop_reason == "end_turn":
            exit

# Add the assistant response to the message history
messages.append(assistant_message)

if stop_reason == "tool_use":
    # Extract the toolUse details
    tool_use = next(
        content["toolUse"]
        for content in assistant_message["content"]
        if "toolUse" in content
    )
    tool_name = tool_use["name"]
    tool_use_id = tool_use["toolUseId"]

    if tool_name == "get_weather":
        # Call the tool
        weather = get_weather(tool_use["input"]["city"])

    # Send the result back to the model
```

```
messages.append(
    {
        "role": "user",
        "content": [
            {
                "toolResult": {
                    "toolUseId": tool_use_id,
                    "content": [{"json": weather}],
                }
            }
        ],
    }
)

# Submit the tool result back to the model with streaming
response = bedrock.converse_stream(
    modelId="us.amazon.nova-2-lite-v1:0",
    messages=messages,
    toolConfig=tool_config,
)

# Handle the final streaming response
print("\n== Text ==")
for event in response["stream"]:
    if "contentBlockDelta" in event:
        delta = event["contentBlockDelta"]["delta"]
        if "text" in delta:
            print(delta["text"], end="", flush=True)
```

## InvokeModel API Examples

The examples below focus on the few key areas where the Invoke API's request and response structures differ slightly from those of the Converse API. In most other ways, the two APIs are compatible, so you should be able to easily adapt the Converse API examples above to work with the InvokeModel API.

### Basic request

Send a basic text request to Amazon Nova 2 models using the InvokeModel API.

## Non-streaming

```
import json

import boto3
from botocore.config import Config

# Configure the request
request_body = {
    "messages": [
        {
            "role": "user",
            "content": [{"text": "Write a short story. End the story with 'THE END'."}],
        }
    ],
    "system": [{"text": "You are a children's book author."}], # Optional
    "inferenceConfig": { # These parameters are optional
        "maxTokens": 1500,
        "temperature": 0.7,
        "topP": 0.9,
        "topK": 50,
        "stopSequences": ["THE END"],
    },
}
bedrock = boto3.client(
    "bedrock-runtime",
    region_name="us-east-1",
    config=Config(read_timeout=3600),
)

# Invoke the model
response = bedrock.invoke_model(
    modelId="us.amazon.nova-2-lite-v1:0", body=json.dumps(request_body)
)
response_body = json.loads(response["body"].read())

# Extract the text response
content_list = response_body["output"]["message"]["content"]
for content in content_list:
    if "text" in content:
        print(content["text"])
```

## Streaming

```
import json

import boto3
from botocore.config import Config

# Configure the request
request_body = {
    "messages": [
        {
            "role": "user",
            "content": [{"text": "Write a short story. End the story with 'THE END'."}],
        }
    ],
    "system": [{"text": "You are a children's book author."}], # Optional
    "inferenceConfig": { # These parameters are optional
        "maxTokens": 1500,
        "temperature": 0.7,
        "topP": 0.9,
        "topK": 50,
        "stopSequences": ["THE END"],
    },
}
bedrock = boto3.client(
    "bedrock-runtime",
    region_name="us-east-1",
    config=Config(connect_timeout=3600, read_timeout=3600),
)

# Invoke the model with streaming
response = bedrock.invoke_model_with_response_stream(
    modelId="us.amazon.nova-2-lite-v1:0", body=json.dumps(request_body)
)

# Process the streaming response
for event in response["body"]:
    chunk = json.loads(event["chunk"]["bytes"])
    if "contentBlockDelta" in chunk:
        delta = chunk["contentBlockDelta"]["delta"]
        if "text" in delta:
```

```
    print(delta["text"], end="", flush=True)
```

## InvokeModel API with reasoning

Use the InvokeModel API with reasoning enabled for complex problem-solving.

### Non-streaming

```
import json

import boto3
from botocore.config import Config

# Configure the request
request_body = {
    "messages": [
        {
            "role": "user",
            "content": [
                {
                    "text": 'How many capital letters appear in the following passage. Your response must include only the number: "Wilfred ordered an anvil from ACME. Shipping was expensive."
                }
            ],
        }
    ],
    "reasoningConfig": {
        "type": "enabled",
        "maxReasoningEffort": "low", # "low" | "medium" | "high"
    },
}

bedrock = boto3.client(
    "bedrock-runtime",
    region_name="us-east-1",
    config=Config(read_timeout=3600),
)

# Invoke the model
response = bedrock.invoke_model(
    modelId="us.amazon.nova-2-lite-v1:0", body=json.dumps(request_body)
```

```
)  
response_body = json.loads(response["body"].read())  
  
# Extract response content  
content_list = response_body["output"]["message"]["content"]  
for content in content_list:  
    # Extract the reasoning response  
    if "reasoningContent" in content:  
        print("\n== Reasoning ==")  
        print(content["reasoningContent"]["reasoningText"]["text"])  
    # Extract the text response  
    if "text" in content:  
        print("\n== Text ==")  
        print(content["text"])
```

## Streaming

```
import json  
  
import boto3  
from botocore.config import Config  
  
# Configure the request  
request_body = {  
    "messages": [  
        {  
            "role": "user",  
            "content": [  
                {  
                    "text": "How many capital letters appear in the following  
passage. Your response must include only the number: \"Wilfred ordered an anvil from  
ACME. Shipping was expensive.\""  
                }  
            ],  
        },  
    ],  
    "reasoningConfig": {  
        "type": "enabled",  
        "maxReasoningEffort": "low",  # "low" | "medium" | "high"  
    },  
}  
  
bedrock = boto3.client(
```

```
"bedrock-runtime",
region_name="us-east-1",
config=Config(connect_timeout=3600, read_timeout=3600),
)

# Invoke the model with streaming
response = bedrock.invoke_model_with_response_stream(
    modelId="us.amazon.nova-2-lite-v1:0", body=json.dumps(request_body)
)

# Process the streaming response
for event in response["body"]:
    chunk = json.loads(event["chunk"]["bytes"])

    if "contentBlockDelta" in chunk:
        delta = chunk["contentBlockDelta"]["delta"]

        # Extract the reasoning response
        if "reasoningContent" in delta:
            print("\n== Reasoning ==")
            print(delta["reasoningContent"]["reasoningText"]["text"], end="", flush=True)

        # Extract the text response
        if "text" in delta:
            print("\n== Text ==")
            print(delta["text"], end="", flush=True)
```

## Troubleshooting

This section provides solutions to common issues when working with Amazon Nova models.

### Authentication and setup

#### Missing permissions

**Symptoms:** Unable to access Nova models or features

**Solution:**

- Ensure your IAM role has `AmazonBedrockFullAccess` or appropriate permissions
- Request specific model access through the Amazon Bedrock console

- Verify permissions for model access and tool usage

## Model access denied

**Symptoms:** Model access request fails

**Solution:**

- Request specific model access through the Amazon Bedrock console
- Verify your account has been granted access to the requested model
- Check regional availability of the model

## Regional availability issues

**Symptoms:** Feature not available in selected region

**Solution:**

- Web Grounding is only available in US regions with US CRIS profiles
- Verify the model and features are available in your selected region
- Switch to a supported region if necessary

## Timeout configuration

**Symptoms:** Requests timing out before completion

**Cause:** Default timeout too short for complex operations

**Solution:** Configure extended timeout settings

```
from botocore.config import Config

bedrock = boto3.client(
    'bedrock-runtime',
    region_name='us-east-1',
    config=Config(
        read_timeout=3600      # 60 minutes
    )
)
```

**Note**

Amazon Nova inference requests can take up to 60 minutes for complex operations.

## API response issues

Understanding stop reasons:

`end_turn`

Normal completion. No action needed.

`max_tokens`

Token limit reached.

**Solution:** Increase `maxTokens` parameter in `inferenceConfig`.

`content_filtered`

Content violated AWS Responsible AI policy.

**Solution:** Review and modify your input to comply with content policies.

`malformed_model_output`

Invalid output format.

**Solution:** Check your output schema and constraints; verify JSON schema is properly formatted.

`malformed_tool_use`

Invalid tool call format.

**Solution:** Verify tool definitions match expected schema; check tool input parameters are correctly formatted.

`service_unavailable`

Built-in tool service unavailable.

**Solution:** Retry the request after a brief delay; check AWS service health dashboard.

## invalid\_query

Invalid query to built-in tool.

**Solution:** Review query format and parameters; ensure query meets tool requirements.

## max\_tool\_invocations

Tool retries exhausted.

**Solution:** Simplify the task or break it into smaller steps; review tool error messages for specific issues.

# Reasoning mode errors

Truncated responses with high reasoning effort

**Solution:** For high reasoning effort, unset these parameters: temperature, topP, maxToken. This allows the model to use optimal settings for complex reasoning tasks.

Insufficient tokens for reasoning

**Error:** "maxTokens is insufficient"

**Solution:** Automatically retry with increased limit

```
token_limits = {  
    "low": 15000,  
    "medium": 30000,  
    "high": 50000  
}  
  
try:  
    response = client.converse(  
        modelId="us.amazon.nova-2-lite-v1:0",  
        messages=messages,  
        inferenceConfig={  
            "maxTokens": token_limits[max_effort]  
        },  
        additionalModelRequestFields={  
            "reasoningConfig": {  
                "type": "enabled",  
                "maxReasoningEffort": max_effort  
            }  
    )
```

```
        }
    )
except Exception as e:
    if "maxTokens is insufficient" in str(e):
        higher_limit = int(token_limits[max_effort] * 1.5)
        # Retry with higher limit
```

## Tool use issues

### Schema validation failures

Tool schema validation errors

**Solution:**

- Limit JSON schemas to two layers of nesting for best performance
- Ensure all required fields are properly defined
- Validate schema against JSON Schema specification

Model not using tools correctly

**Solution:**

- Ensure tool name clearly describes its purpose
- Provide detailed description of tool functionality
- Explicitly define input schema with clear parameter descriptions
- Include examples in the description when helpful

Inconsistent tool calling behavior

**Solution:** Set temperature to 0 for tool calling:

```
inferenceConfig={
    "temperature": 0,
    "maxTokens": 10000
}
```

This enables greedy decoding for more reliable tool use.

Tool choice conflicts

**Problem:** Error when using custom tools with web search or code interpreter

**Solution:** Do not include a custom toolSpec with name nova\_grounding - this conflicts with the system tool. Use the system tool configuration instead:

```
# Correct - use system tool
tool_config = {
    "tools": [
        "systemTool": {"name": "nova_grounding"}
    ]
}

# Incorrect - don't create custom tool with this name
# tool_config = {
#     "tools": [
#         "toolSpec": {"name": "nova_grounding", ...}
#     ]
# }
```

## Web Grounding issues

Access control problems

**Problem:** Web Grounding and Code Interpreter not working

**Solution:** Ensure your IAM policy includes:

```
{
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["bedrock:InvokeTool"],
            "Resource": ["arn:aws:bedrock::{YOUR_ACCOUNT_ID}:system-tool/
amazon.nova_grounding"]
        }
    ]
}
```

Service Control Policy issues

**Problem:** Web Grounding blocked by SCP

**Solution:** If you have Service Control Policies with aws:requestedRegion condition, update them to allow "unspecified" region for Web Grounding functionality.

## Media processing limitations

Poor understanding of multilingual content in images/videos

**Limitation:** Nova models have limited understanding of multilingual content in visual media

**Workaround:**

- Provide text translations alongside images
- Use text-based inputs for multilingual content when possible

People identification

**Problem:** Model refuses to identify people in images

**Expected Behavior:** Models will refuse to identify or name individuals in images, documents, or videos for privacy and safety reasons

**Workaround:** Ask about general characteristics or context instead of specific identities

Spatial reasoning limitations

**Problem:** Inaccurate localization or layout analysis

**Limitation:** Limited capabilities for precise spatial reasoning

**Workaround:**

- Use bounding box detection for object localization
- Provide clear reference points in your prompts
- Break complex spatial queries into simpler components

Small text in images/videos

**Problem:** Cannot read small text in media

**Solution:**

- Crop images to focus on relevant text sections
- Increase resolution of source media
- Provide text separately if available

## Document and file handling

### Unsupported content

**Problem:** PDF processing fails

#### Causes:

- PDFs with CMYK color profiles
- PDFs containing SVG images

#### Solution:

- Convert PDFs to RGB color profile
- Rasterize SVG images before including in PDFs

### Token estimation

**Problem:** Unexpected token usage with PDFs

**Guideline:** Estimate approximately 2,560 tokens per standard 8.5×11" PDF page

**Solution:** Adjust maxTokens accordingly based on document length

# API and SDK reference

This section provides complete technical reference documentation for Amazon Nova APIs and SDKs.

## Topics

- [Request and response schema](#)
- [SDK reference](#)

## Request and response schema

The request schema is nearly identical between the Invoke API and Converse API. The primary difference is how binary data (images, video, audio) is encoded: the Converse API uses binary arrays while the Invoke API uses Base64-encoded strings.

## Complete request structure

The following shows the complete request structure for Amazon Nova models. All fields are optional unless marked as required:

```
{
  "system": [
    {
      "text": "string"
    }
  ],
  "messages": [ // Required
    {
      "role": "user", // Required - first turn must be user
      "content": [ // Required
        {
          "text": "string"
        },
        {
          "image": {
            "format": "jpeg" | "png" | "gif" | "webp", // Required
            "source": { // Required
              "bytes": image // Binary array (Converse) or Base64 string (Invoke)
            }
          }
        }
      ]
    }
  ]
}
```

```
        }
    ],
    {
        "video": {
            "format": "mkv" | "mov" | "mp4" | "webm" | "three_gp" | "flv" | "mpeg" |
"mpg" | "wmv",
            "source": {
                // Option 1: S3 location
                "s3Location": {
                    "uri": "string", // e.g., s3://my-bucket/object-key
                    "bucketOwner": "string" // Optional, e.g., "123456789012"
                },
                // Option 2: File bytes
                "bytes": video // Binary array (Converse) or Base64 string (Invoke)
            }
        }
    ],
    {
        "audio": { // Nova 2 Omni and Sonic only
            "format": "mp3" | "opus" | "wav" | "aac" | "flac" | "mp4" | "ogg" | "mkv",
            "source": {
                // Option 1: S3 location
                "s3Location": {
                    "uri": "string",
                    "bucketOwner": "string" // Optional
                },
                // Option 2: File bytes
                "bytes": audio // Binary array (Converse) or Base64 string (Invoke)
            }
        }
    }
],
{
    "role": "assistant",
    "content": [
        {
            "text": "string" // For prefilling assistant response
        }
    ]
},
],
"inferenceConfig": { // All optional
    "maxTokens": int, // 1-5000, default: dynamic
```

```
"temperature": float, // 0.00001-1, default: 0.7
"topP": float, // 0-1, default: 0.9
"topK": int, // 0-128, default: not used
"stopSequences": ["string"],
"reasoningConfig": { // Nova 2 Lite and Sonic only
    "type": "enabled" | "disabled", // default: "disabled"
    "maxReasoningEffort": "low" | "medium" | "high"
}
},
"toolConfig": { // Optional
    "tools": [
        {
            "toolSpec": {
                "name": "string", // Max 64 characters
                "description": "string",
                "inputSchema": {
                    "json": {
                        "type": "object",
                        "properties": {
                            "arg1": {
                                "type": "string",
                                "description": "string"
                            }
                        },
                        "required": ["string"]
                    }
                }
            }
        }
    ]
},
"toolChoice": { // Choose one option
    "auto": {},
    "any": {},
    "tool": {
        "name": "string"
    }
}
}
```

## Key request parameters:

- **system:** System prompt providing context and instructions

- **messages:** Array of conversation turns with role (user or assistant) and content
- **inferenceConfig:** Controls model output behavior (temperature, tokens, and so on.)
- **toolConfig:** Tool specifications for function calling

 **Note**

When using the Converse API, the `topK` and `reasoningConfig` parameters must be placed in `additionalModelRequestFields` instead of `inferenceConfig`.

The following sections provide detailed explanations of each request parameter:

### **system**

**system** – (Optional) The system prompt for the request. A system prompt provides context and instructions to Amazon Nova, such as specifying a particular goal or role.

### **messages**

**messages** – (Required) The input messages array containing conversation turns.

- **role** – (Required) The role of the conversation turn. Valid values are `user` and `assistant`. The first message must always use the `user` role.
- **content** – (Required) An array of content blocks. Each block specifies a content type (`text`, `image`, `video`, or `audio`):
  - **text** – Text content for the conversation turn. If combined with `image` or `video`, interpreted as accompanying text.
  - **image** – (Not supported for Nova 2 Lite) Image content with:
    - **format** – (Required) Image format: `jpeg`, `png`, `webp`, or `gif`
    - **source.bytes** – (Required) Image data as binary array (Converse API) or Base64 string (Invoke API)
  - **video** – (Not supported for Nova 2 Lite) Video content with:
    - **format** – (Required) Video format: `mkv`, `mov`, `mp4`, `webm`, `three_gp`, `flv`, `mpeg`, `mpg`, or `wmv`
    - **source** – (Required) Video source via S3 URI (`s3Location.uri` and optional `bucketOwner`) or file bytes (`bytes`)
  - **audio** – (Amazon Nova Sonic and only) Audio content with:

- **format** – (Required) Audio format: mp3, opus, wav, aac, flac, mp4, ogg, or mkv
- **source** – (Required) Audio source via S3 URI or file bytes

## **inferenceConfig**

**inferenceConfig** – (Optional) Configuration parameters controlling model output generation.

- **maxTokens** – (Optional) Maximum tokens to generate before stopping. Amazon Nova models may stop before reaching this limit. Maximum value is 5,000. If not specified, uses a dynamic default based on request context.
- **temperature** – (Optional) Randomness in responses. Valid range: 0.00001-1 (default: 0.7). Lower values produce more deterministic output.
- **topP** – (Optional) Nucleus sampling threshold. Amazon Nova samples from tokens whose cumulative probability reaches topP. Valid range: 0-1 (default: 0.9). Adjust either temperature or topP, not both.
- **topK** – (Optional) Sample from top K tokens only. Removes low-probability responses. Valid range: 0-128 (default: not used).

 **Note**

For Converse API, pass topK in additionalModelRequestFields.

- **stopSequences** – (Optional) Array of strings that stop generation when encountered.
- **reasoningConfig** – (Amazon Nova Sonic and only) Reasoning configuration:
  - **type** – (Optional) enabled or disabled (default: disabled)
  - **maxReasoningEffort** – Computational effort: low, medium, or high. With low and medium, reasoning streams incrementally; high outputs reasoning in a final chunk.

 **Note**

For Converse API, pass reasoningConfig in additionalModelRequestFields.

## **toolConfig**

**toolConfig** – (Optional) Tool configuration following [ToolConfiguration schema](#).

- **tools** – Array of tool specifications with name, description and `inputSchema`
- **toolChoice** – (Optional) Controls tool selection:
  - **auto** – Model decides whether and which tools to use
  - **any** – Model must use at least one tool
  - **tool** – Model must use the specified tool by name

## Complete response structure

The following shows the complete response structure for Amazon Nova models:

```
{  
  "ResponseMetadata": {  
    "RequestId": "string",  
    "HTTPStatusCode": int,  
    "HTTPHeaders": {  
      "date": "string",  
      "content-type": "application/json",  
      "content-length": "string",  
      "connection": "keep-alive",  
      "x-amzn-requestid": "string"  
    },  
    "RetryAttempts": 0  
  },  
  "output": {  
    "message": {  
      "role": "assistant",  
      "content": [  
        {  
          "reasoningContent": { // Optional - if reasoning enabled  
            "reasoningText": {  
              "text": "[REDACTED]"  
            }  
          }  
        },  
        {  
          "toolUse": { // Optional - if tool called  
            "toolUseId": "string",  
            "name": "string",  
            "input": {} // Tool-specific arguments  
          }  
        },  
        {  
          "response": {  
            "text": "[REDACTED]"  
          }  
        }  
      ]  
    }  
  }  
}
```

```
{  
    "text": "string" // Optional - text response  
},  
{  
    "image": { // Optional - Nova 2 Omni only  
        "format": "png",  
        "source": {  
            "bytes": image // Binary array (Converse) or Base64 string (Invoke)  
        }  
    }  
}  
]  
}  
,  
"stopReason": "string", // See stop reasons below  
"usage": {  
    "inputTokens": int,  
    "outputTokens": int,  
    "totalTokens": int  
},  
"metrics": {  
    "latencyMs": int  
}  
}
```

### Stop reasons:

- **end\_turn**: Natural end of response
- **max\_tokens**: Reached maxTokens limit
- **content\_filtered**: Violated content policy
- **malformed\_model\_output**: Invalid model output
- **malformed\_tool\_use**: Invalid tool use output
- **service\_unavailable**: Built-in tool service unreachable
- **invalid\_query**: Invalid query to built-in tool
- **max\_tool\_invocations**: Tool retries exhausted

The following sections provide detailed explanations of each response field:

## output

output – (Required) Contains the model's response message.

- message – (Required) The assistant's response message with role and content array.
- content – (Required) Array of content blocks that can include:
  - reasoningContent – (Optional) Returned if reasoning was enabled. Contains reasoning text, which will always be [REDACTED] in the response.
  - toolUse – (Optional) Returned if a tool was called. Contains tool use ID, name and input arguments.
  - text – (Optional) Returned if the model responded with text content.
  - image – (Optional, only) Returned if the model generated an image. Format will always be PNG.

## stopReason

stopReason – (Required) Indicates why the model stopped generating output:

- end\_turn – Natural end of response reached
- max\_tokens – Reached maxTokens limit or model's maximum output limit
- content\_filtered – Output violated AWS Responsible AI policy
- malformed\_model\_output – Model produced invalid output
- malformed\_tool\_use – Model produced invalid tool use output
- service\_unavailable – Built-in tool service could not be reached
- invalid\_query – Query to built-in tool was invalid
- max\_tool\_invocations – Built-in tool did not produce valid result after retries

## usage

usage – (Required) Token usage information:

- inputTokens – Total tokens ingested by the model
- outputTokens – Number of tokens generated
- totalTokens – Sum of input and output tokens

## metrics

metrics – (Required) Performance metrics:

- latencyMs – Total inference completion time in milliseconds

## SDK reference

AWS software development kits (SDKs) are available for many popular programming languages. Each SDK provides an API, code examples and documentation that make it easier for developers to build applications in their preferred language.

SDK documentation	Code examples
<a href="#">AWS SDK for C++</a>	<a href="#">AWS SDK for C++ code examples</a>
<a href="#">AWS CLI</a>	<a href="#">AWS CLI code examples</a>
<a href="#">AWS SDK for Go</a>	<a href="#">AWS SDK for Go code examples</a>
<a href="#">AWS SDK for Java</a>	<a href="#">AWS SDK for Java code examples</a>
<a href="#">AWS SDK for JavaScript</a>	<a href="#">AWS SDK for JavaScript code examples</a>
<a href="#">AWS SDK for Kotlin</a>	<a href="#">AWS SDK for Kotlin code examples</a>
<a href="#">AWS SDK for .NET</a>	<a href="#">AWS SDK for .NET code examples</a>
<a href="#">AWS SDK for PHP</a>	<a href="#">AWS SDK for PHP code examples</a>
<a href="#">AWS Tools for PowerShell</a>	<a href="#">AWS Tools for PowerShell code examples</a>
<a href="#">AWS SDK for Python (Boto3)</a>	<a href="#">AWS SDK for Python (Boto3) code examples</a>
<a href="#">AWS SDK for Ruby</a>	<a href="#">AWS SDK for Ruby code examples</a>
<a href="#">AWS SDK for Rust</a>	<a href="#">AWS SDK for Rust code examples</a>
<a href="#">AWS SDK for SAP ABAP</a>	<a href="#">AWS SDK for SAP ABAP code examples</a>
<a href="#">AWS SDK for Swift</a>	<a href="#">AWS SDK for Swift code examples</a>

# Nova Forge

Nova Forge is the easiest and most cost-effective way to build your own frontier models using Amazon Nova. Customers can start their development from early model checkpoints, blend proprietary data with Amazon Nova-curated training data and host their custom models securely on AWS.

## Key benefits

### **Setup access checkpoints across all phases of model development**

Start your model development on SageMaker AI using early Nova checkpoints across pre-training, mid-training, or post-training phases. This lets you introduce your proprietary data at the optimal point in the model training, maximizing the model's learning from your data.

### **Blend your proprietary data with Amazon Nova-curated training data**

Blend proprietary data with Amazon Nova-curated training data using Amazon provided SageMaker AI recipes. This approach lets you build a model that deeply understands your organization's proprietary knowledge, while minimizing risks like catastrophic forgetting and preserving foundational capabilities like reasoning.

### **Perform reinforcement learning with reward functions in your environment**

Integrate reward functions in your environment for Reinforcement Fine Tuning (RFT). This allows the model to learn from feedback generated in your environment from your applications.

### **Use the built-in responsible AI toolkit to implement safety guardrails**

Use the responsible AI toolkit available in Nova Forge to configure the safety and content moderation settings of your custom model. You can adjust settings to meet your specific business needs in areas like safety, security and handling sensitive content.

## Connect with experts at the AWS Generative AI Innovation Center

Engage our generative AI experts through the Custom Model Program offered by the AWS Generative AI Innovation Center. Your team can work with experienced generative AI strategists, applied scientists and engineers to train and optimize models for your needs.

For complete details on Nova Forge capabilities, setup and workflows, refer to the Nova Forge User Guide.

# Responsible use

Building safety, security and trust measures with AI models is a shared responsibility between AWS and our customers. Our goal is to align our models to the [AWS Acceptable Use Policy](#) and mitigate undesired outcomes while providing a delightful customer experience. Our approach to Responsible AI (RAI) is structured around our [core dimensions of responsible AI](#), which are covered in the following list. For each of these dimensions, we developed guidelines that govern our decision-making throughout the entire model development life cycle. This life cycle encompasses every stage, from initial data collection and pre-training, to the implementation of post-deployment runtime mitigations.

- *Fairness* - Considering impacts on different groups of stakeholders
- *Explainability* - Understanding and evaluating system outputs
- *Privacy and Security* - Appropriately obtaining, using and protecting data and models
- *Safety* - Preventing harmful output and misuse
- *Controllability* - Having mechanisms to monitor and steer AI system behavior
- *Veracity and robustness* - Achieving correct system outputs, even with unexpected or adversarial inputs
- *Governance* - Incorporating best practices into the AI supply chain, including providers and deployers
- *Transparency* - Enabling stakeholders to make informed choices about their engagement with an AI system

## Guidelines

The guidelines we use to direct our model development includes but is not limited to moderating content that glorifies, facilitates, or promotes the following:

- Participation in dangerous activities, self harm, or use of dangerous substances.
- Use, misuse, or trade of controlled substances, tobacco, or alcohol.
- Physical violence or gore.
- Child abuse or child sexual abuse material.
- Animal abuse or animal trafficking.

- Misinformation that positions individuals or groups as responsible for deliberate deception, undermining an institution with general public credibility, or endangering human health or livelihood.
- Malware, malicious content, or any content that facilitates cyber-crime.
- Disrespect, discrimination, or stereotype towards an individual or group.
- Insults, profanity, obscene gestures, sexually explicit language, pornography, hate symbols, or hate groups.
- Full nudity that is outside of a scientific, educational, or reference context.
- Bias against a group based on a demographic characteristic.

## Recommendations

**Appropriateness for Use:** Because AI model outputs are probabilistic, Amazon Nova may produce inaccurate or inappropriate content. Customers should evaluate outputs for accuracy and appropriateness for their use case, especially if they will be directly surfaced to end users. Additionally, if Amazon Nova is used in customer workflows that produce consequential decisions, customers must evaluate the potential risks of their use case and implement appropriate human oversight, testing and other use-case specific safeguards to mitigate such risks.

**Prompt Optimizations:** In the event of encountering moderation by Amazon Nova, consider examining the prompts used with respect to the guidelines above. Optimizing the prompts to reduce the likelihood of generating undesired outcomes is the recommended strategy to produce the expected outputs using Amazon Nova models. Pay attention where the input is controlled by users, including pixel content that could impact the performance of the model. Please see the prompt guidelines section in this user guide for further details.

**Privacy:** Amazon Nova is available in Amazon Bedrock. Amazon Bedrock is a managed service and does not store or review customer prompts or customer prompt completions and prompts and completions are never shared between customers, or with Amazon Bedrock partners. AWS does not use inputs or outputs generated through the Amazon Bedrock service to train Amazon Bedrock models, including Amazon Nova. See Section [50.3](#) of the AWS Service Terms and the [AWS Data Privacy FAQ](#) for more information. For service-specific privacy information, see the Privacy and Security section of the [Amazon Bedrock FAQs](#) documentation. Amazon Nova takes steps to avoid completing prompts that could be construed as requesting private information. If a user is concerned that their private information has been included in a Amazon Nova completion, the user should contact us [here](#).

**Security:** All Amazon Bedrock models, including Amazon Nova, come with enterprise security that enables customers to build generative AI applications that support common data security and compliance standards, including GDPR and HIPAA. Customers can use AWS PrivateLink to establish private connectivity between customized Amazon Nova and on-premise networks without exposing customer traffic to the internet. Customer data is always encrypted in transit and at rest and customers can use their own keys to encrypt the data, e.g., using AWS Key Management Service. Customers can use AWS Identity and Access Management to securely control access to Amazon Bedrock resources, including customized Amazon Nova. Also, Amazon Bedrock offers comprehensive monitoring and logging capabilities that can support customer governance and audit requirements. For example, Amazon CloudWatch can help track usage metrics that are required for audit purposes and AWS CloudTrail can help monitor API activity and troubleshoot issues as Amazon Nova is integrated with other AWS systems. Customers can also choose to store the metadata, prompts and completions in their own encrypted Amazon Simple Storage Service (Amazon S3) bucket.

**Intellectual Property:** AWS offers uncapped intellectual property (IP) indemnity coverage for outputs of generally available Amazon Nova models (see Section 50.10 of the [Service Terms](#)). This means that customers are protected from third-party claims alleging IP infringement or misappropriation (including copyright claims) by the outputs generated by these Amazon Nova models. In addition, our standard IP indemnity for use of the Services protects customers from third-party claims alleging IP infringement (including copyright claims) by the Services (including Amazon Nova models) and the data used to train them.

# Quotas for Amazon Nova

Your AWS account has default quotas, formerly referred to as limits, for Amazon Bedrock. To view service quotas for Amazon Bedrock, do one of the following:

- Follow the steps at [Viewing service quotas](#) and select **Amazon Bedrock** as the service.
- Refer to [Amazon Bedrock endpoints and quotas](#) in the AWS General Reference.

 **Note**

If a quota is marked as not adjustable through Service Quotas, you can submit a request through the [limit increase form](#) to be considered for an increase.

To maintain the performance of the service and to ensure appropriate usage of Amazon Bedrock, the default quotas assigned to an account might be updated depending on regional factors, payment history, fraudulent usage, and/or approval of a quota increase request.

## Request an increase for Amazon Nova quotas

You can request a quota increase for your account by following the steps below:

- If a quota is marked as **Yes** in the **Adjustable** column in [Amazon Bedrock endpoints and quotas](#) in the AWS General Reference, you can adjust it by following the steps at [Requesting a Quota Increase](#) in the *Service Quotas User Guide* in the [Service Quotas User Guide](#).
- The **On-demand model invocation** quotas in [Amazon Bedrock endpoints and quotas](#) in the AWS General Reference aren't adjustable through Service Quotas. Contact your AWS account manager to be considered for an increase.

 **Note**

Due to overwhelming demand, priority will be given to customers who generate traffic that consumes their existing quota allocation. Your request might be denied if you don't meet this condition.

- You can submit a request through the [limit increase form](#) to be considered for an increase even if a quota is marked as **No** in the **Adjustable** column in [Amazon Bedrock endpoints and quotas](#) in the AWS General Reference,

## Monitoring Amazon Nova

Monitoring is an important part of maintaining the reliability, availability and performance of Amazon Nova and your other AWS solutions. AWS provides the following monitoring tools to watch Amazon Nova, report when something is wrong and take automatic actions when appropriate:

Because Amazon Nova is available through Amazon Bedrock, follow all monitoring guidance at [Monitor the health and performance of Amazon Bedrock.](#)

# Security in Amazon Nova

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from data centers and network architectures that are built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The [shared responsibility model](#) describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the [AWS Compliance Programs](#). To learn about the compliance programs that apply to Amazon Nova, see [AWS Services in Scope by Compliance Program](#).
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements and applicable laws and regulations.

Because Amazon Nova is available through Amazon Bedrock, follow all security guidance at [Security in Amazon Bedrock](#).

# Create Amazon Nova resources with AWS CloudFormation

Because Amazon Nova is available through Amazon Bedrock, follow all CloudFormation guidance at [Create Amazon Bedrock resources with CloudFormation](#).

For more information, including examples of JSON and YAML templates for YOUR-RESOURCE-PLURAL, see the [YOUR-SERVICE-RESOURCE-TOPNODE-IN-CLOUDFORMATION, modify this link in the local-phrases.ent file](#) in the *AWS CloudFormation User Guide*.

## Learn more about CloudFormation

To learn more about CloudFormation, see the following resources:

- [AWS CloudFormation](#)
- [AWS CloudFormation User Guide](#)
- [CloudFormation API Reference](#)
- [AWS CloudFormation Command Line Interface User Guide](#)

# Document history for the Amazon Nova Developer Guide

The following table describes the documentation releases for Amazon Nova 2.0.

Change	Description	Date
<a href="#"><u>Initial release of Amazon Nova 2.0</u></a>	Initial release of the Amazon Nova 2.0 Developer Guide with new models: Nova 2 Lite, Amazon Nova Sonic and . This guide introduces enhanced multimodal capabilities, reasoning features and improved performance across all model tiers.	December 2, 2025