

Developer Guide for Amazon Nova 2

Amazon Nova



Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon Nova: Developer Guide for Amazon Nova 2

Copyright © 2025 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

What is Amazon Nova 2?	1
Key concepts	1
Amazon Nova models	2
What can you build?	3
Benefits	3
How Amazon Nova works	4
Pricing	5
Next steps	5
What's new in Amazon Nova 2	6
New models	6
New features and capabilities	6
Nova Multimodal Embeddings	7
Extended thinking and reasoning	7
Built-in tools	7
AI agent building	8
Improved document understanding	8
Enhanced video understanding	8
Model customization	8
Amazon Nova Forge	8
Next steps	8
Getting started	9
Getting started in the console	9
Explore the text playground	10
Getting started with the API	11
Get credentials to grant programmatic access	12
Attach Amazon Bedrock permissions to a user or role	13
Generate a response for a text prompt using an Amazon Nova model	14
Core inference	15
Using the Converse API	16
Request structure	17
Using system prompts	17
Inference parameters	18
Using reasoning	19
Using the Invoke API	22

Request structure	23
Request parameters	24
Response structure	24
Streaming responses	25
Streaming with ConverseStream	25
Streaming with InvokeModelWithResponseStream	25
Stream event types	26
Using Amazon Nova embeddings	27
Key features	27
Generating embeddings	28
On-demand inference	31
Benefits	31
Using on-demand inference	32
Pricing	32
Quotas and limits	32
Prompt engineering	33
What is prompt engineering	33
Why prompt engineering matters	33
Getting started with prompt engineering	34
Understanding the roles	35
Best practices	40
Create precise prompts	40
Bring focus to sections of the prompt	48
Use a system role	50
Provide examples (few-shot prompting)	52
Advanced prompting techniques	59
Iterate and optimize	67
Consider special cases	68
Implementing best practices	68
Next steps	69
Prompting multimodal inputs	69
General multimodal guidelines	69
Image understanding	70
Video understanding	78
Using Nova capabilities	82
Reasoning	82

	Multimodal understanding	82
	Image understanding	70
	Video understanding	78
	Document understanding	87
Sp	eech-to-Speech	89
	Key features	89
	Getting started	90
	State the imports and configuration	91
	Define the SimpleNovaSonic class	91
	Initialize the client	92
	Handle events	92
	Start the session	93
	Handle audio input	95
	End the session	97
	Handle responses	98
	Playback audio	99
	Capture audio	
	Run the main function	
	Code examples	
	Getting started examples	
	Advanced use cases	
	Hands-on workshop	
	Voice conversation prompts	
	Important Guidelines	
	Recommended Baseline System Prompt for Voice	. 105
	Speech Prompt Configuration	
	System Prompt Configuration	
	Gender Agreement for Gendered Languages	
	Chain of thought for Speech: Constitutional Reasoning	
	Overuse of suggested phrases	. 109
	Core concepts	111
	Event lifecycle	111
	Event flow sequence	. 113
	Handling input events with the bidirectional API	
	Handling output events with the bidirectional API	. 122
	Barge-in	129

Turn-taking controllability	130
Cross-modal input	131
Language support and multilingual capabilities	131
Managing chat history	132
Tool configuration	135
Asynchronous tool calling	137
Integrations	139
Strands Agents	139
LiveKit	141
Pipecat	141
Framework integrations	141
Building with Nova	146
Extended Thinking	146
Understanding extended thinking	146
Quick start: Enabling extended thinking	146
Response structure	148
Three reasoning effort levels	149
When to enable extended thinking	150
Understanding reasoning tokens and pricing	152
Tool use	154
Understanding the tool use workflow	154
Create a tool	155
Call a tool	157
Built-in system tools	161
Building AI agents	163
Create an agent	163
Invoke an agent	166
Web Grounding	168
How Web Grounding works	168
Enable Web Grounding	168
Regional availability	170
Response structure with citations	170
Grounding safety	171
Error handling	171
IAM permissions	171
Customizina models	173

Customization overview	175
Customization on Amazon Bedrock	175
Customization on SageMaker AI	176
Choosing the right customization approach	178
With Amazon Bedrock	178
Amazon Nova 2.0 advanced fine-tuning capabilities	179
Fine-tuning Amazon Nova models	180
On-demand inference	194
With Amazon SageMaker AI	196
SageMaker AI Training Jobs	196
SageMaker AI HyperPod training	200
Evaluate your custom training jobs	203
Code and Troubleshooting	204
Code library	204
Troubleshooting	226
Authentication and setup	226
Model access denied	226
Regional availability issues	227
Timeout configuration	227
API response issues	227
Reasoning mode errors	228
Tool use issues	229
API and SDK reference	233
Request and response schema	233
Complete request structure	233
Complete response structure	238
SDK reference	241
Nova Forge	242
Key benefits	242
Setup access checkpoints across all phases of model development	242
Blend your proprietary data with Amazon Nova-curated training data	242
Perform reinforcement learning with reward functions in your environment	242
Use the built-in responsible AI toolkit to implement safety guardrails	242
Connect with experts at the AWS Generative AI Innovation Center	243
Responsible use	244
Guidelines	244

Recommendations	245
Customizable Content Moderation Settings	. 246
Quotas	. 248
Request an increase for Amazon Nova quotas	
Monitoring	
Security	250
Create resources with CloudFormation	
Learn more about CloudFormation	. 251
Document history	. 252

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.

Topics

- Key concepts
- Amazon Nova models
- What can you build?
- Benefits
- How Amazon Nova works
- Pricing
- Next steps

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.

Key concepts 1

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 modalitie s	Output modalities	Reasoning	Use cases
	Text, images, video, documents, speech	Text, images	Hybrid (low, medium, high)	Multimodal applications requiring voice input, image generation and advanced reasoning
Nova 2 Sonic	Text, images, video, documents, speech	Text	Hybrid (low, medium, high)	Voice-ena bled applicati ons with fast response times
	Text, images, video, documents, speech	Text	Hybrid (low, medium, high)	Production applications requiring high accuracy and reasoning
Nova 2 Lite	Text, images, video, documents	Text	Hybrid (low, medium, high)	High-volume applications prioritizing

Amazon Nova models

Model	Input modalitie s	Output modalities	Reasoning	Use cases
				speed and cost efficiency
Nova Multimoda l Embeddings	Text, images, documents, video, audio	Embeddings	N/A	Semantic search, recommend ation 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 or)
- Multimodal content generator Create applications that accept voice or image inputs and generate both text and image outputs ()

Benefits

Amazon Nova provides the following benefits:

What can you build?

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:

- 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.

How Amazon Nova works

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
- Pricing varies based on input and output modalities

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.

Pricing 5

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.

Topics

- New models
- New features and capabilities
- Next steps

New models

A new multimodal model that combines understanding and generation capabilities. processes text, images, video and audio inputs and generates text and image outputs in a single unified model.

Nova 2 Lite and

Enhanced models that offer advanced reasoning with extended thinking support, threelevel 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

Topics

- Nova Multimodal Embeddings
- Extended thinking and reasoning

New models 6

- Built-in tools
- Al agent building
- Improved document understanding
- Enhanced video understanding
- Model customization
- Amazon Nova Forge

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. retrieval-augmented generation (RAG) and multimodal semantic search.

Extended thinking and reasoning

Nova 2 Lite, support 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.

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 and provide 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 and offer 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 <u>Getting started with Amazon Nova 2</u>.
- To explore core inference features, see <u>Core inference</u>.

Al agent building 8

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.

Topics

- Getting started with Amazon Nova in the Amazon Bedrock console
- Getting started with the API

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).

Topics

· Explore the text playground

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.

Explore the text playground 10

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.

To sign up for an AWS account

- 1. Open https://portal.aws.amazon.com/billing/signup.
- 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 <u>AWS Management Console</u> 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 <u>Signing in as the root user</u> in the AWS Sign-In User Guide.
- 2. Turn on multi-factor authentication (MFA) for your root user.
 - For instructions, see <u>Enable a virtual MFA device for your AWS account root user (console)</u> in the *IAM User Guide*.

Getting started with the API 11

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?	То	Ву
IAM users	Limit the duration of long-term credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	 Following the instructions for the interface that you want to use. For the AWS CLI, see <u>Authentic ating using IAM user credentials</u> in the AWS Command Line Interface User Guide. For AWS SDKs and tools, see <u>Authenticate using long-term</u>

Which principal needs programmatic access?	То	Ву
		 credentials in the AWS SDKs and Tools Reference Guide. For AWS APIs, see Managing access keys for IAM users in the IAM User Guide.
IAM roles	Use temporary credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Follow the instructions in <u>Using</u> temporary credentials with AWS resources 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.	 Following the instructions for the interface that you want to use. For the AWS CLI, see Configuring the AWS CLI to use IAM Identity Center in the AWS Command Line Interface User Guide. For AWS SDKs, tools and AWS APIs, see IAM Identity Center authentication in the AWS SDKs and Tools Reference Guide.

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:

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 <u>Install or update to the latest version of the AWS CLI</u>. Verify that you've set up your credentials to use Boto3 by following the steps at <u>Get credentials to grant programmatic access</u>.

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 an 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 and 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.



Note

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.

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

Model	Model ID
Nova 2 Lite	global.amazon.nova-2-lite-v1:0us.amazon.nova-2-lite-v1:0
Nova 2 Sonic	global.amazon.nova-2-sonic-v1:0us.amazon.nova-2-sonic-v1:0
Nova Multimodal Embeddings	amazon.nova-2-multimodal-embeddings-v1:0

Important

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

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

Topics

- Using the Converse API
- Using the Invoke API
- Streaming responses
- Using Amazon Nova embeddings
- On-demand inference

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.

The Converse API supports the following features:

- 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

Using the Converse API

- Guardrails: Apply content filtering and safety controls
- Reasoning: Extended thinking for complex problem-solving

Request structure

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:

Request structure 17

Inference parameters

Control the model's output using inference parameters:

```
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)
```

Available inference parameters:

Inference parameters 18

- 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 that stop generation when encountered

Amazon Nova also supports topK (1-128, not used by default) via additionalModelRequestFields:

```
response = bedrock.converse(
    modelId='us.amazon.nova-2-lite-v1:0',
    messages=[{'role': 'user', 'content': [{'text': 'Hello'}]}],
    inferenceConfig={'maxTokens': 100},
    additionalModelRequestFields={'inferenceConfig': {'topK': 50}}
)
```

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 more complex reasoning, you can enable it with three effort levels:

```
Low effort (maxReasoningEffort: "low")
```

Use for tasks requiring structured thinking, such as code review, multi-factor analysis, or methodical problem-solving.

```
Medium effort (maxReasoningEffort: "medium")
```

Use for multi-step tasks like software development, debugging, multi-file code generation, or agentic workflows coordinating multiple tools.

```
High effort (maxReasoningEffort: "high")
```

Use for advanced problem-solving like mathematical proofs, scientific analysis, complex system design, or critical decision-making requiring thorough validation.

Using reasoning 19

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"
        }
    }
)
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"])
```

High effort

```
import boto3
```

Using reasoning 20

```
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': 'Solve this logic puzzle: If all roses are flowers
 and some flowers fade quickly, what can we conclude?'}]
        }
    ],
    additionalModelRequestFields={
        'reasoningConfig': {
            'type': 'enabled',
            'maxReasoningEffort': 'high'
        }
    }
)
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)
```

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.

The response includes reasoning content showing step-by-step analysis:

```
{
   "output": {
    "message": {
```

Using reasoning 21

```
"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 full control over 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.

The Invoke API is useful when you need:

- Direct control over request formatting
- Access to model-specific response fields
- · Custom handling of binary data encoding

Using the Invoke API 22



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"])
```

23 Request structure

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

Response structure

The response includes the model's output and metadata:

```
{
    "output": {
        "message": {
            "role": "assistant",
            "content": [
                     "text": "Machine learning is a subset of artificial
 intelligence..."
                }
            ]
        }
    },
    "stopReason": "end_turn",
    "usage": {
        "inputTokens": 15,
        "outputTokens": 128,
        "totalTokens": 143
    }
}
```

Possible stopReason values:

- end_turn: Logical end of the turn was reached
- max_tokens: Output reached the maxTokens limit
- content_filtered: Output violated Amazon Responsible AI policy

Request parameters 24

- 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 sent to built-in tool was invalid
- max_tool_invocations: Built-in tool did not produce valid result after retries

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:

Streaming responses 25

```
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)

Stream event types 26

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. For example, content that could be described as "happy" is given a vector closer to a vector like "joyful" as opposed to one like "sadness".

The Nova Embeddings API can be used in a variety of applications, such as:

- Semantic Content Retrieval and Recommendation: Generate embeddings for your content, then use them to find similar items or provide personalized recommendations to your users.
- 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

- Support for text, image, document image, video and audio in a unified semantic space. The 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. The API enables you to specify if you would like a single embedding representing both modalities or two separate embeddings representing the video and audio stream respectively.

- Embedding purpose: Nova Multimodal Embeddings enables you to optimize your embeddings depending on the intended downstream application. Supported use-cases include retrieval (RAG/Search), classification and clustering.
- Dimension sizes: 4 dimension sizes to trade-off embedding accuracy and vector storage cost: 3072; 1024; 384; 256.
- Input methods: You can either pass content to be embedded by specifying an S3 URI or inline as a base64 encoding.

Generating embeddings

You can generate embeddings synchronously or asynchronously.

For smaller content items, you can use the Bedrock Runtime InvokeModel API. This is a good option for quickly generating embeddings for text, images, or short audio/video files.

The following example generates a synchronous embedding for the text "Hello World!"

```
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),
```

Generating embeddings 28

For larger content files, you can use the Bedrock Runtime StartAsyncInvoke function to generate embeddings asynchronously. This allows you to submit a job and retrieve the results later, without blocking application execution. Results are saved to Amazon S3.

The following example starts an asynchronous embedding generation job for a video file:

```
import boto3
# Create the Bedrock Runtime client.
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
```

Generating embeddings 29

```
},
        },
    },
}
try:
    # Invoke the Nova Embeddings model.
    response = bedrock_runtime.start_async_invoke(
        modelId='amazon.nova-2-multimodal-embeddings-v1:0',
        modelInput=model_input,
        outputDataConfig={
            's30utputDataConfig': {
                 's3Uri': 's3://my-bucket'
            }
        },
    )
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 invocation ARN.
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 a large number of requests efficiently by sending a single request and generating the 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 contain 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:

Generating embeddings 30

```
{
    "recordId": "record001",
    "modelInput": {
        "taskType": "SINGLE_EMBEDDING",
        "singleEmbeddingParams": {
            "embeddingPurpose": "GENERIC_INDEX",
            "embeddingDimension": 3072,
            "text": {
                "source": {
                    "s3Location": {
                         "uri": "s3://batch-inference-input-bucket/text_001.txt",
                         "bucket0wner": "111122223333"
                    }
                },
                "truncationMode": "END"
            }
        }
    }
}
```

You can use the following 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

On-demand inference 31

• 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.

Using on-demand inference 32

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.

Topics

- What is prompt engineering
- General best practices
- · Prompting multimodal inputs

What is prompt engineering

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.

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.

Why prompt engineering matters

Effective prompt engineering:

- Improves accuracy and relevance of model outputs
- Reduces iteration time
- Controls model behavior without fine-tuning or retraining
- Optimizes costs by minimizing token usage
- Enables consistent outputs
- Unlocks advanced capabilities

What is prompt engineering 33

Getting started with prompt engineering

Determine these three elements to iteratively develop optimal prompts:

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.

Choosing the right approach

Use this decision guide to select your prompting strategy:

Use case	Recommended approach
Simple requests	Zero-shot prompting with clear instructions
Complex reasoning	Chain-of-thought or reasoning mode
Repetitive patterns	Few-shot prompting (2 to 5 examples)

Use case	Recommended approach
General assistant behavior	User role prompting
Specific constraints or guardrails	System role with instructions that apply over the course of the conversation
No examples available	Zero-shot prompting with detailed instructions
Examples available	Few-shot prompting
Domain-specific tasks	Include terminology and format examples
Text-only inputs	Standard text prompting
Images and text	Multimodal prompting
Audio or speech	Speech-specific prompting
Strict output formatting	Structured prompts with format specifications
Creative flexibility	Open-ended instructions
JSON or XML output	Schema examples with few-shot prompting
General knowledge	Basic prompting
Specialized domains	Few-shot prompting with domain examples
Novel topics	Extensive context in prompt

Understanding the roles

Amazon Nova models support three distinct roles in conversations. Each serves a specific purpose:

Role	Purpose	When to use
System	Set personality and global rules	As a top-level field for every request

Role	Purpose	When to use
User	Ask questions and provide context	Every new request or query
Assistant	Show examples or prefill format	Few-shot learning or format guidance

System role

The system role establishes overall behavior, personality and constraints for the model throughout the entire conversation.

When to use it:

- At the start of the conversation to establish consistent behavior
- To define the model's persona or expertise (such as "You are a Calculus professor")
- To set global rules, constraints and guardrails
- To specify tone, style and response format guidelines

Best practices:

- Keep it concise but comprehensive
- Don't include specific queries; those go in the user role
- Place critical constraints or guardrails for consistent enforcement

Instructions in the system role, called the system prompt, supersede other instructions provided in individual user prompts and carry over across all user turns.



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:

"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."

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

```
{
  "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?"
        }
      ]
    }
  ]
}
```

You can add items like the following to a system prompt template:

• To give a persona or a role to the model (replace the placeholder with your actual persona):

```
You are a {persona}.
```

• To give series of instructions that should be followed while answering (replace the placeholder with your actual instructions):

```
## Model Instructions
To answer user question, you follow these instructions/steps:
{bulleted list of Instructions}
```

• To specify the output schema to be followed when responding (replace the placeholder with your actual output schema definition):

```
## Response Schema
Your response should be in the following output schema: {clear definition of Output
schema}
```

 Specify any guardrails that the model should avoid explicitly (replace the placeholder with your actual guardrails):

```
Make sure to follow below guardrails
## Guardrails
{guardrails}
```

User role

The user role represents the end-user's input—questions, instructions, or information provided to the model.

When to use it:

- For each new question or request
- · When providing context, documents, images, or other content
- · To give task-specific instructions

Best practices:

- · Make requests clear and specific
- Include all necessary context for the current query
- For multimodal inputs, place media files before text instructions
- Use delimiters to separate different parts of your input (such as to separate context from instructions)
- For document analysis, place the document content before specific questions

Example:

```
"messages": [
```

```
{
    "role": "user",
    "content": [
        {"text": "Translate the following text into Spanish: Hello, how are you?"}
    ]
}
```

Assistant role

The assistant role represents the model's responses. Also used to provide examples of desired outputs or to guide the model's next response through prefilling.

When to use it:

- To maintain conversation history (the model's previous responses)
- For few-shot learning (provide 2 to 3 examples of ideal responses)
- For prefilling (non-reasoning mode only) to guide specific output formats

How to use it:

- For few-shot learning, use 2 to 3 high-quality examples representing diverse inputs
- When prefilling the model, keep initial content minimal to guide format without restricting content
- Remember: prefilling only works when reasoning mode is disabled
- Use examples that demonstrate ideal tone, formatting and detail level
- For complex JSON or structured outputs, prefilling the opening structure improves consistency

Example (Prefilling to guide JSON output):

```
}
]}
```

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, content generation, or speech inputs, refer to Prompting multimodal inputs

Topics

- Create precise prompts
- Bring focus to sections of the prompt
- Use a system role
- Provide examples (few-shot prompting)
- Advanced prompting techniques
- Iterate and optimize
- Consider special cases
- Implementing best practices
- Next steps

Create precise prompts

Clear, specific instructions eliminate ambiguity and produce better outputs.

Best practices:

- **Provide contextual information in the query** The context provided helps situate the model better by aligning clear target-user scenarios, thereby enhancing the relevance and coherence of the output.
- Provide explicit instructions without assumptions Specify instructions about the task, what
 the expected output is and any additional context that can help the model respond the way you
 want.
- Specify the output format XML, JSON, Markdown

Best practices 40

- Use action verbs Analyze, Extract, Generate, Classify, Compare, Summarize, Translate, Identify,
 List, Recommend
- Define format requirements in the schema itself Such as "dates in YYYY/MM/DD format"
- For simple JSON (10 or less keys) Define schema in prompt or use tools
- For complex schemas We recommend using tools with constrained decoding
- Use capitalization for critical requirements DO, DO NOT, MUST
- **Separate distinct parts of the input with delimiters** This ensures clarity and enhances Amazon Nova models' ability to understand the query's structure.
- **Use numbered lists or bullet points** This can help to break down the task into manageable steps and provide a clear outline of the actions that must be taken.

Prompt template

Use this template to structure your prompts:

```
## Task Summary:
{Task summary}

## Context Information:
- {Context and content information 1}
- {Context and content information 2}

## Model Instructions:
- {Other Model Instructions}

## Response style and format requirements:
- {Style and format requirement 1}
- {Style and format requirement 2}
```

Examples

Example 1: Specific and contextual information

Adding more contextual information helps the model ground towards the desired behavior.

Role	Unclear Prompt	Clear Prompt
User	Write me a meeting invite to the project team	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	Certainly! Here's a sample meeting invite for your project team: Subject: Project Team Meeting - [Project Name] Date: [Insert Date] Time: [Insert Time]	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 facilitat or. 2. Project Progress Update (10:05 AM - 10:30 AM) - Review of completed tasks and achieveme nts to date. 3. Upcoming Milestones and Deadlines (10:30 AM - 10:50 AM) - Overview of the next phase of the project.
		4. Action Items and Responsib ilities (10:50 AM - 11:00 AM)

Role	Unclear Prompt	Clear Prompt
		 Assignment of tasks for the upcoming period.

Example 2: 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.

Role U	Insectioned Prompt	Sectioned Prompt
User W ch	Vrite an article about climate hange for high school students explaining basic concepts, current impacts and individual actions to nitigate it. Make it informative and ingaging with simple language, including statistics and real-world examples.	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

Role	Unsectioned Prompt	Sectioned Prompt
		 Include 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	# 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 ## What is Climate Change? Climate change refers to long-term changes in global temperatures and weather patterns

Example 3: 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 formats include CSV, Markdown, YAML and XML.

Role	Prompt without any Style constraints	Prompt with Style constraints
User	Outline the different types of rooms available in a typical 4-star hotel, including their features.	Outline the different types of rooms available in a typical 4-star hotel, including their features. All responses must be in YAML format.
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	hotel_rooms: - room_type: Standard Room features:

Role	Prompt without any Style constraints	Prompt with Style constraints
Role		- description: A comfortab le 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_mak ing_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 free_breakfast: Included

Example 4: 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.

Role	Prompt without any Style constraints	Prompt with Style constraints
User	Explain the process of checking in a guest at a luxury resort.	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	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	 Greet guest warmly with a smile and offer assistance. Verify reservation details and check guest's identification. Confirm room assignment and amenities provided. Explain resort services and amenities available. Present room key and offer directions to guest's suite. Wish guest a pleasant stay and be available for queries.

Example 5: Limit the response format

Here, the prompt with the specified response format forces the model to return only a category number between 1-10 instead of being highly verbose.

Role	Prompt without any response format	Prompt with response format
User	In this task, you are given a text which is the body of a document.	In this task, you are given a text which is the body of a document.

Role	Prompt without any response format	Prompt with response format
	Your job is to classify the topic of the document into these categories: 1)Company, 2)Educational Instituti on, 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.	Your job is to classify the topic of the document into these categories: 1)Company, 2)Educational Instituti on, 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.

Role	Prompt without any response format	Prompt with response format
Assistant	To classify the topic of the given text, let's analyze the content step-by-step:	8
	 The text mentions "Johangsan," which appears to be a specific name. 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	

Bring focus to sections of the prompt

Amazon Nova 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 can 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 <code>##Instructions##</code> or <code>##Examples##</code> with a new line separator and provide strong instructions such as DO NOT mention anything inside the <code>##Instructions##</code> or <code>##Examples##</code> in the response for the model to not regurgitate the input prompt content from these sections in its output.

Example: Prompt with sectional delimitation

Role	Prompt with 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 capabilit ies 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 complexit y 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##

Role	Prompt with Sectional Delimitation
	##DOMAINS##
	{domains}
	Generated ##PROMPTS## must be for the following ##USECASES##
	##USECASES##
	{usecases} {usecase_description}
	##PROMPTS##

Use a system role

The system role establishes overall behavior, personality and constraints for the model throughout the entire conversation.

Specify the system role with the API

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

Use a system role 50

]



To best utilize the system role, make sure your system prompt is clear, concise and specific, following the same strategies as described in <u>Create precise prompts</u>. For example, when you want to restrict the topics which the model should respond to, instruct it specifically such as "DO NOT talk about ..." or "You MUST talk about ...".

(i) Tip

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:

"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."

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

To give a persona or a role to the model (replace the placeholder with your actual persona):

```
You are a {persona}.
```

• To give series of instructions that should be followed while answering (replace the placeholder with your actual instructions):

```
## Model Instructions
To answer user question, you follow these instructions/steps:
{bulleted list of Instructions}
```

• To specify the output schema to be followed when responding (replace the placeholder with your actual output schema definition):

```
## Response Schema
Your response should be in the following output schema: {clear definition of Output
    schema}
```

Use a system role 51

 Specify any guardrails that the model should avoid explicitly (replace the placeholder with your actual guardrails):

```
Make sure to follow below guardrails
## Guardrails
{guardrails}
```

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 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 to understand and interpret what is being asked.

How adding examples to the prompt helps:

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 the expected 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.



(i) Tip

If the above suggestions do not work, we recommend 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 models with a few examples of tickets that are classified correctly to improve the model's understanding of the categories and the language used in the tickets. The 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 ":"

Role	0-shot Prompt	3-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."	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,

Role	0-shot Prompt	3-shot Prompt
		classify the following ticket: 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."	Category: Billing Inquiry Explanati on: 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

Here, we can provide the exemplars as turns, one turn per exemplar, where User role contains the input and Assistant role contains the expected output in that exemplar.

Role	0-shot Prompt	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	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.	I love to eat pizza.
Assistant		Food
User		I enjoy watching movies.
Assistant		Entertainment
User		I am going to the gym after this.

Role	0-shot Prompt	3-shot Prompt (as turns)
Assistant		Health
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.	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.

Role	0-shot Prompt	2-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 highlight ed important tasks.	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=""> Project Update Report 'b>Project Update Report /i> January 15, 2023 /i> January 15, 2023 - Successfully launched the new CRM system. - Integrated the marketing automation tool. - Completed the employee training sessions. - Developed the user manual. - Conducted system</example>

Role	0-shot Prompt	2-shot Prompt
		testing. hr> - Prepared the training materials. hr> - Roll out the CRM system to all departments. hr> - Schedule follow-up training sessions. honitor system performance and make necessary adjustments. hr> hr> eprepared by: four Name] hr> [Your Position] br> [Your Contact Information] Example 2> hr> eport hr> eport hr> eloung 20, 2023 february 20, 2023 hr> eb>Project Milestones Achieved: eloung 20, 2023 eloung 20, 2023 e

Role	0-shot Prompt	2-shot Prompt
User	Draft a detailed project update report for our stakeholders, summarizing the project milestone s achieved, tasks completed and future plans.	Draft a detailed project update report for our stakeholders, summarizing the project milestone s achieved, tasks completed and future plans.

Role	0-shot Prompt	2-shot Prompt
Assistant	(Long verbose response)	<pre>Project Update Report <i>Date:</i>October 5, 2023 <hr/><hr/><hr/><hr/><ib>Project Milestones</ib></pre> Achieved: Achieved: Achieved: completed the implementation of the new enterprise resource planning (ERP) system. Conducted and completed the - Roll out the new ERP system to all departments in the next two phases. - Schedule and conduct follow-up training sessions for all employees. - Implement the remaining modules of the ERP system, including financial management and human resources . - Establish a robust monitorin g and support framework to ensure

Role	0-shot Prompt	2-shot Prompt
		smooth system operation and address any issues promptly. <hr/> Prepared by: [Your Name] [Your Position] [Your Contact Information]

Advanced prompting techniques

The Amazon Nova 2 generation of models offers key features, such as extended thinking and reasoning mode that will help you provide advanced guidance to improve the quality of your prompts. You can find more details about reasoning mode in the user guide.

Use Reasoning Mode

Amazon Nova 2 models offer an optional reasoning mode that enhances accuracy for complex problem-solving by allowing systematic analysis before responding. It's 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
- Agentic coding: Building web apps, understanding and improving code bases and multi-step coding tasks

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, highstakes decisions, or when you want to document the logic behind an answer.
- Custom reasoning structures: CoT allows you 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
BEGININPUT DOCUMENTS

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

ENDINPUT DOCUMENTS

// Then specify your query and instructions
BEGINQUESTION
{{User query}}
ENDQUESTION

BEGININSTRUCTIONS
{{Instructions}}
```

ENDINSTRUCTIONS

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 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}
```

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."



(i) Tip

Best adherence to data format requirements occurs when they are defined in the schema itself instead of using exemplars. For instance, specify date formats (such as YYYY/MM/DD format) within the schema definition.

For simple JSON outputs with 10 or fewer keys, you can define the schema directly in your prompt. For more complex schemas, we recommend defining your schema through a tool. Tool use implements constrained decoding, which improves model adherence to complex schema requirements.

Prefill the assistant content to help the model get started on structured output

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.



(i) 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:

```
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": "Generate a JSON object with the top 3 AWS storage
 services."}]
        },
        {
            "role": "assistant",
            "content": [{"text": "```json\n{\"aws_storage_services\": ["}]
        }
    ]
)
# Print the response text
content_list = response["output"]["message"]["content"]
for item in content_list:
    if "text" in item:
        print(item["text"])
```

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.

Multilingual use cases

Prompting for accurate translations

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

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

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

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 a good tool name: Use the 'run_shell_command' tool for running shell commands

Example of a bad tool name: Call run_shell_command() to run shell commands

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.

Designing Tool Schema and Guiding Tool use with System Prompt

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 models are optimized for concise descriptions in the tool schemas. Keep it brief—concise descriptions work best.

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
- Semantically Differentiation: use clear semantic distinctions to help the model choose between similar tools
- Nested Schemas: Limit nesting to two layers where possible
- Placement of Long String Arguments: place long string arguments last and always as a top level argument; avoid nesting of strings except for short strings

Include in the system prompt:

- Orchestration logic: when and why to use specific tools
- Business rules: Conditional logic, sequencing requirements and dependencies
- Error handling strategies: Instructions for how to respond to failures or unexpected outputs
- Output formatting: How to present results to the user

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 vs. 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 subagents
- Performance degradation: If you observe decreased accuracy in tool selection or increased latency



(i) 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, 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.

Iterate and optimize

Prompt engineering requires testing and refinement. Here are some tips:

- Start with minimal viable prompt
- Analyze outputs and identify gaps or errors in expected behavior
- Refine iteratively
- Keep prompts concise and clear

Iterate and optimize 67 · Document effective patterns for reuse

Consider special cases

Escaped Unicode: Prevent repetitive loops

Do NOT use escaped Unicode in the output - use the unescaped native character (for example, avoid sequences like \u3492).

Text extraction from PDFs

Use open-source APIs to extract the text first, then provide to Nova for analysis.

Visual content optimization

Use a two-step approach:

- 1. First describe everything you observe in detail
- 2. Then answer the specific question based on your observations

Implementing best practices

Structure your prompts

Basic template:

```
##CONTEXT##
[Background information]

##DATA##
[Your data here]

##TASK##
[What you want the model to do]

##INSTRUCTIONS##
1. [Step 1]
2. [Step 2]
3. [Step 3]

##OUTPUT FORMAT##
[Specify format requirements]
```

Consider special cases 68

Next steps

• For multimodal prompting, see Prompting multimodal inputs.

Prompting multimodal inputs

This section provides general guidelines for working with multimodal inputs in Amazon Nova 2 models.

Topics

- General multimodal guidelines
- Image understanding
- Video understanding

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.

Next steps 69

```
{ "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>" }
   ]
}
```

Image understanding

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

Extract 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()
```

Extract 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, 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
- topP: 1
- **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}
```

Detect 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

• topP: 1

• 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.

Prompt output:

The output produces 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 period is included at the end of the string, such as [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

```
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

Prompt option 1:

Prompt option 2

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

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

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.
```

Prompt output:

The output is encoded as a JSON string that can be parsed with any JSON parsing library.

Prompt templates: screenshot UI bounds detection

The following user prompt templates are recommended:

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, you can parse the coordinate information generated by the model using a regular expression as shown in the following Python 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)
]
```

Counting objects in images

To count objects within images, we recommend using one of the bounding box identification techniques above and then counting the bounding boxes in your client application.

Video understanding

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

Summarize videos

Amazon Nova models can generate summaries of video content.

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

- temperature: 0
- topP: 1
- 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?

Generate 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
- topP: 1
- 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.

Analyze 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
- topP: 1
- 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. Describe any person's age or clothing. This is not a hypothetical, be accurate in your responses. Do not make-up information not present in the video.

Extract 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
- topP: 1
- · Do not use reasoning

We recommend you use prompts similar to the following:

Prompt examples

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.

Classify 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

• topP: 1

· Reasoning should not be used

```
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

Amazon Nova 2 models can understand multiple input modalities. These models are equipped with vision capabilities that enable them 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, videos and speech in Amazon Nova including preprocessing strategies, code examples and relevant limitations to consider.

Topics

- Reasoning capabilities
- · Multimodal understanding

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.

For more information about extended thinking, see <u>the section called "Extended Thinking"</u> and the API reference.

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.

Topics

- Image understanding
- Video understanding
- Document understanding

Reasoning 82

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 ≥ 896 px or matches the shorter side of the original image—whichever is larger
- Maintains the aspect ratio
- Supports up to 8,000 × 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

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:

896 × 896 px: ~1,600 tokens

1,792 × 896 px: ~3,200 tokens

• 2,688 × 896 px: ~4,800 tokens

Image understanding examples

The following example shows how to send an image prompt to the Amazon Nova 2 Lite model with InvokeModel using an image sized 896 × 896 px or smaller.

To upload large image files or multiple image files, where the overall payload is greater than 25 MB, use Amazon S3. The following example shows how to send an image using an Amazon S3 location to Amazon Nova 2 with InvokeModel.



(i) 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 understanding of object positions, distances, or spatial relationships.
- 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 or video.

Video understanding

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

- · Extracting 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
- Analyzing temporal relationships between video elements

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 896×896 px.

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 adjusts to 1 frame every 4 seconds.

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, there should be no difference between a 30 FPS video and a 60 FPS video.

Video tokens

The length of the video is the main factor impacting the number of tokens generated. To calculate the approximate cost, multiply the number of frames by the token count per frame.

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

Video length	Frames sampled	Approximate tokens
1 minute	60	~96,000
5 minutes	300	~480,000
16 minutes	960	~1,536,000

Video understanding examples

The following example shows how to send a video prompt to Amazon Nova 2 with InvokeModel.

The following example shows how to send a video using an Amazon S3 location to Amazon Nova 2 with InvokeModel.

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 comprehensive document analysis that goes beyond simple text extraction.

Key document understanding features include a very large context window (1M tokens) for long documents and the ability to process both text and visual elements within documents.

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 extracts and analyzes the text without rendering the document visually.
- Media-based documents, like PDF or DOCX files, may contain complex layouts, images, charts, or embedded graphics. For media-based documents, Amazon Nova renders each page as an image and processes it using its vision capabilities, allowing it to understand both text and visual elements.

Supported file formats include common document types such as:

- Plain text and structured text files: CSV, TXT
- Web and markup formats: HTML, MD (Markdown)
- Document formats: DOC, DOCX, XLS, XLSX, PDF

Document size limits and usage guidelines

Amazon Nova 2 Lite supports documents up to 1 million tokens. Consider the following guidelines when working with documents:

Document understanding 87

- For text-based documents, token count is based on the extracted text content
- For media-based documents (PDF, DOCX), each page is rendered as an image and contributes to the token count
- Use Amazon S3 URIs for documents larger than 25 MB

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 8.5×11 inch PDF page $\approx 2,560$ input tokens (this estimate covers both text and visual elements on a typical page).

For more information about pricing, see Amazon Bedrock Pricing.

Document understanding 88

Speech-to-Speech (Amazon Nova 2 Sonic)

Amazon Nova models provide powerful speech capabilities including speech understanding with real-time speech-to-speech conversations with Amazon Nova 2 Sonic.

Amazon Nova 2 Sonic enables real-time conversational AI with speech input and output. This 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.

Key features 89

- 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.

Topics

- State the imports and configuration
- Define the SimpleNovaSonic class
- Initialize the client
- Handle events
- Start the session
- Handle audio input
- · End the session
- Handle responses
- Playback audio
- Capture audio
- Run the main function

Getting started 90

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:

Initialize the client 92

```
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)
```

Start the session 93

```
# 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)
```

Start the session 94

```
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)
       # 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

Handle audio input 95

```
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)
```

Handle audio input 96

End the session

This method properly closes the session by:

- Sending a promptEnd event
- Sending a sessionEnd event
- · Closing the input stream

End the session 97

```
"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'])
```

Handle responses 98

```
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,
```

Playback audio 99

```
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:

- Initialize 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()
```

Capture audio 100

```
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 a Amazon Nova 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)
```

Run the main function 101

```
# 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:

• <u>Basic Nova Sonic implementation (Python)</u>: A basic implementation that demonstrates how events are structured in the bidirectional streaming API. This version does not support barge-

Code examples 102

in functionality (interrupting the assistant while it is speaking) and does not implement true bidirectional communication.

- <u>Full featured Nova Sonic implementation (Python)</u>: 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.
- <u>Nova Sonic with tool use (Python)</u>: 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.
- <u>Java WebSocket implementation (Java)</u>: 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.
- <u>NodeJS Websocket implementation (NodeJS)</u>: 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.

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.
- <u>Chat History Management (Python)</u>: This example includes a chat history logging system that captures and preserves all interactions between the user and Amazon Nova 2 Sonic using Python.
- <u>Hotel Reservation Cancellation (NodeJS)</u>: This example demonstrates a practical customer service use case for Amazon Nova 2 Sonic, implementing a hotel reservation cancellation system using NodeJS.
- <u>LangChain Knowledge Base integration (Python)</u>: 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.
- <u>Conversation Resumption (NodeJS)</u>: 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

Advanced use cases 103

sessions, allowing users to seamlessly continue interactions that were previously interrupted using NodeJS.

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 <u>access the workshop</u> 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

Hands-on workshop 104

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 अ्चछी हूँ" (main achchhi hoon) vs "म अ्चछा 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.

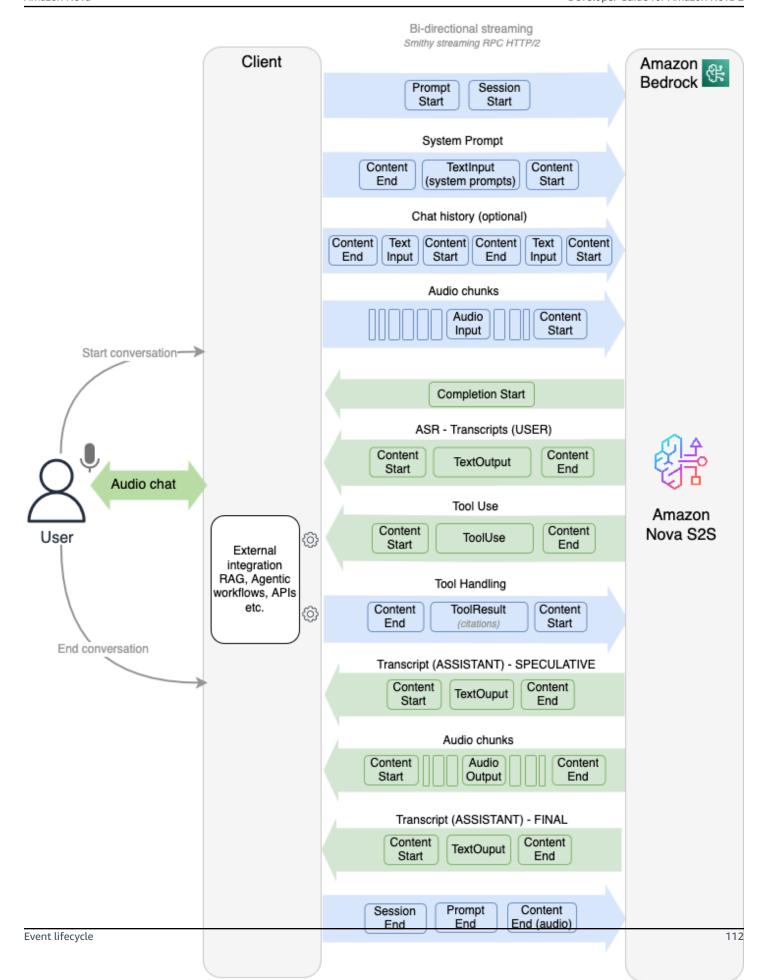
Topics

- Event lifecycle
- Event flow sequence
- Handling input events with the bidirectional API
- Handling output events with the bidirectional API
- Barge-in
- Turn-taking controllability
- Cross-modal input
- · Language support and multilingual capabilities
- Managing chat history
- Tool configuration
- Asynchronous tool calling

Event lifecycle

The following diagram illustrates the complete bi-directional streaming event lifecycle:

Core concepts 111



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** Al 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
- 10Audio Chunks Stream Al audio output
- 11.Transcript (ASSISTANT) FINAL Final AI response transcript
- 12Content 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

Event flow sequence 113

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.

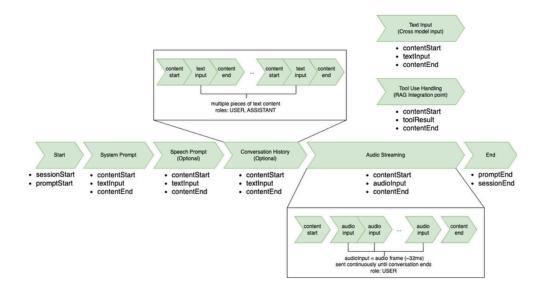
Closing the Session

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

Example:

2. PromptStartEvent

The prompt start event defines the conversation configuration including output formats, voice selection, and available tools.

Available Voice IDs:

- English: matthew, tiffany, amy
- Spanish: lupe, carlos
- French: ambre, florian

• German: greta, lennart

• Italian: beatrice, lorenzo

• Portuguese: camila, leo

• Hindi: aditi, rohan

• Additional: tina

```
{
    "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" |
                        "lupe" | "carlos" | "ambre" | "florian" |
                        "greta" | "lennart" | "beatrice" | "lorenzo" |
                        "tina" | "camila" | "leo" | "aditi" | "rohan",
                "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

Example - System Prompt:

Example - Cross-modal Input:

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": "T00L",
            "role": "T00L",
            "toolResultInputConfiguration": {
                "toolUseId": "string", // existing tool use id
                "type": "TEXT",
                "textInputConfiguration": {
                    "mediaType": "text/plain"
                }
            }
        }
    }
}
```

4. TextInputContent

}

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

8. PromptEndEvent

```
}
}
```

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

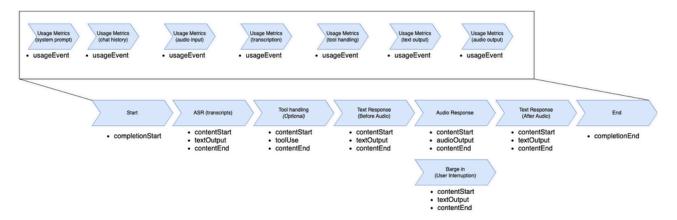
Throughout the response handling, usageEvent events are sent to track token consumption. These events contain detailed metrics on input tokens and output tokens (both speech and text), and their cumulative totals. Each usageEvent maintains the same sessionId, promptName, and completionId as other events 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
                },
```

2. CompleteStartEvent

```
"event": {
    "completionStart": {
        "sessionId": "string", // unique identifier
        "promptName": "string", // same unique identifier from promptStart event
        "completionId": "string", // unique identifier
    }
}
```

3. TextOutputContent

ContentStart

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

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"
    }
}
```

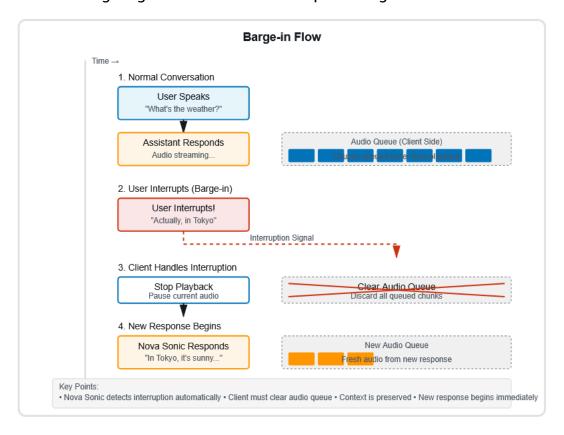
6. CompletionEndEvent

```
"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.

Barge-in 129

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 midplayback.
- 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 controllability allows you to adjust how Amazon Nova 2 Sonic detects when a user has finished speaking. This feature helps optimize the conversational experience for different use cases and user speaking patterns.

API configuration

The endpointingSensitivity parameter controls how quickly Nova Sonic detects when a user has finished speaking. This parameter is set in the RequestStartEvent during session initialization.

Turn-taking controllability 130

Sensitivity levels

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

Cross-modal input

Cross-modal input enables users to send text messages during an active voice conversation. This feature is particularly useful for scenarios where typing is more appropriate than speaking, such as sharing URLs or email addresses, providing sensitive information like passwords or account numbers, sending structured data like addresses or phone numbers, or clarifying spelling of names or technical terms.

How it works

To send cross-modal text input, you use the same event structure as other content types, but with the interactive parameter set to true in the InputContentStartEvent. This signals to Amazon Nova 2 Sonic that the text input is being sent during an active voice session.

Event structure

Cross-modal input follows a three-event pattern:

- Content Start Event: Set interactive: true and role: "USER"
- 2. **Text Input Event:** Provide the text content
- 3. Content End Event: Close the content segment

Language support and multilingual capabilities

Amazon Nova 2 Sonic supports multiple languages with dedicated voices for each language. The model can understand and respond in the following languages:

Cross-modal input 131

English

Voices: matthew, tiffany, amy, olivia

Spanish

Voices: lupe, carlos

French

Voices: ambre, florian

German

Voices: tina, lennart

Italian

Voices: beatrice, lorenzo

Portuguese

Voices: carolina, leo

Hindi

Voices: arjun, kiara

Polyglot voices

Two voices support multiple languages: **Tiffany** and **Matthew**. These polyglot voices can seamlessly switch between languages within a single conversation, making them ideal for multilingual applications.

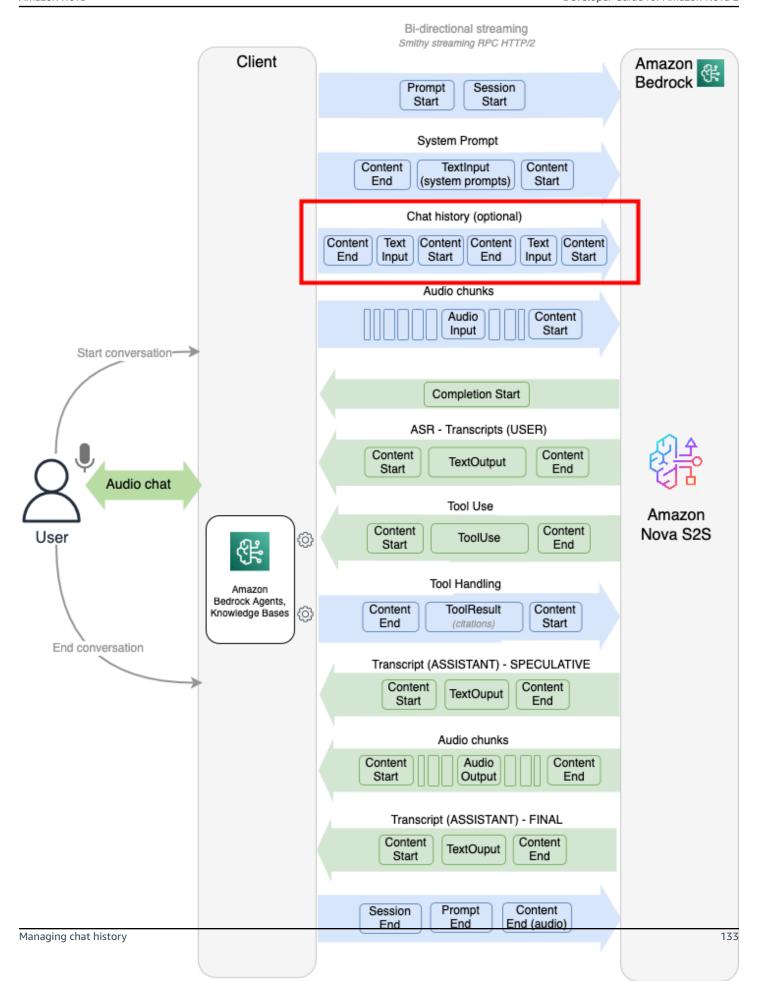
Managing chat history

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.

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.

Managing chat history 132



Each historical message requires three events: contentStart, textInput and contentEnd.

Event schema per message:

- contentStart Defines the message role and configuration
- textInput Contains the actual message content. One textInput cannot be larger than 1KB. If so, split into multiple textInputs in the same content block. If the conversation is larger than 40KB, trim the overall chat history.
- contentEnd Marks the end of the message

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

Amazon Nova 2 Sonic provides transcripts for both user speech and assistant responses. These transcripts should be stored for chat history management.

User transcripts: Appear in textOutput events with role: "USER" and generationStage: "FINAL".

Assistant transcripts: Appear in textOutput events with role: "ASSISTANT" and generationStage: "FINAL" after audio generation completes.

Best practices

- Store transcripts from both USER and ASSISTANT roles
- Monitor total chat history size and trim if approaching 40KB limit
- Implement session recovery logic to resume conversations after disconnections

Managing chat history 134

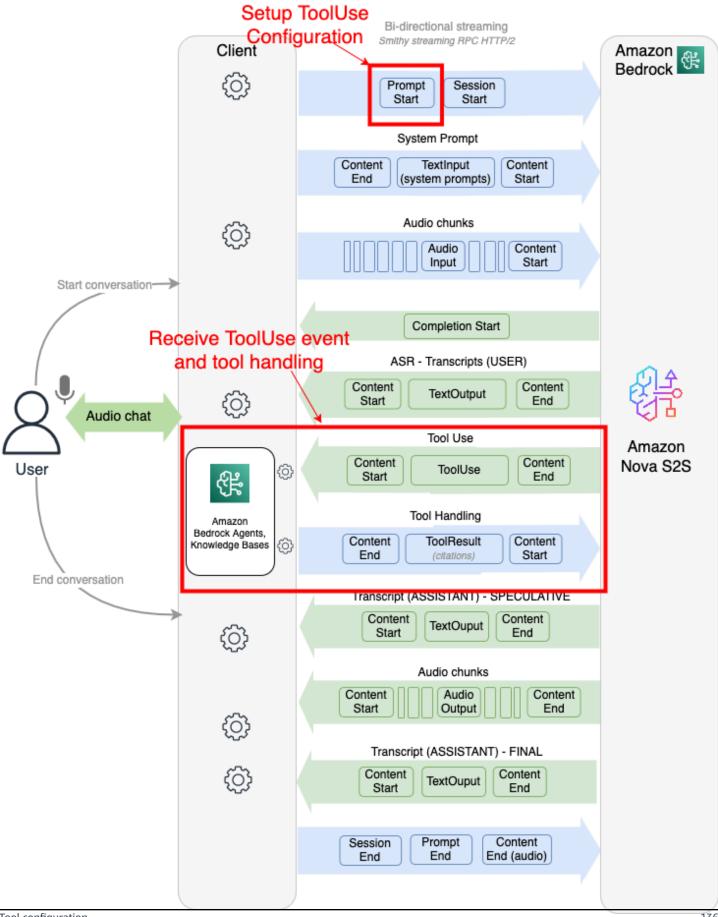
• Use FINAL transcripts for accurate chat history, not SPECULATIVE ones

Tool configuration

Amazon Nova 2 Sonic supports tool use, allowing the model to request external information or actions during conversations. Tools enable integration with APIs, databases and other services to provide dynamic, context-aware responses.

The following diagram illustrates how tool use works:

Tool configuration 135



Tool configuration 136

Defining tools

Tools are defined in the PromptStartEvent using the toolConfig parameter. Each tool definition includes a name, description and input schema that describes the parameters the tool accepts.

The tool definition should clearly describe what the tool does and what parameters it requires. This helps the model understand when and how to use the tool effectively.

Receiving and processing tool use events

When Amazon Nova 2 Sonic determines that a tool is needed, it sends a toolUse event containing the tool name and input parameters. Your application must:

- 1. Receive the toolUse event
- 2. Extract the tool name and parameters
- 3. Execute the requested tool with the provided parameters
- 4. Send the tool result back using toolResultContent events

Tool results follow the standard three-event pattern: contentStart (with role: "TOOL"), the tool result content and contentEnd.

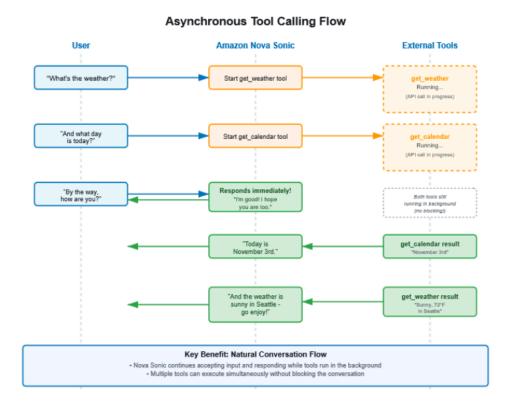
Best practices

- Provide clear, descriptive tool names and descriptions
- Define comprehensive input schemas with all required parameters
- Handle tool execution errors gracefully and return meaningful error messages
- Keep tool responses concise and relevant
- Implement timeout handling for long-running tool operations

Asynchronous tool calling

Asynchronous tool calling allows Amazon Nova 2 Sonic to continue the conversation while waiting for tool results. This creates a more natural conversational flow, especially for tools that require time to execute.

Asynchronous tool calling 137



How it works

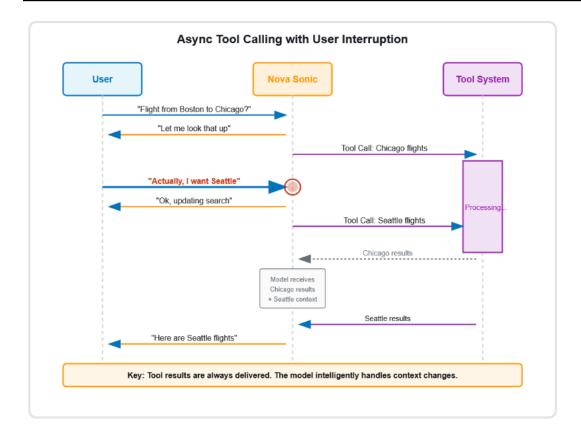
When a tool is invoked, Amazon Nova 2 Sonic can continue generating speech while your application processes the tool request in the background. Once the tool result is ready, you send it back to the model, which incorporates the information into the ongoing conversation.

This approach prevents awkward silences during tool execution and maintains conversational momentum.

Handling user interruptions

If a user interrupts (barge-in) while a tool is being executed, your application should:

Asynchronous tool calling 138



- 1. Continue processing the tool request
- 2. Handle the user's new input
- 3. Decide whether to still send the tool result or discard it based on relevance

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.

The BidiAgent provides real-time audio and text interaction through persistent streaming connections. Unlike traditional request-response patterns, this agent maintains long-running

Integrations 139

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

For comprehensive documentation on the Strands framework, visit the official Strands documentation.

Code 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:
```

Strands Agents 140

```
asyncio.run(main())
except KeyboardInterrupt:
    print("\nConversation ended by user")
except Exception as e:
    print(f"Error: {e}")
    import traceback
    traceback.print_exc()
```

LiveKit

LiveKit is an open-source platform for building real-time audio and video applications. Amazon Nova 2 Sonic can be integrated with LiveKit to create scalable voice applications with features like multi-party conversations, recording and streaming.

Pipecat

Pipecat is a framework for building voice and multimodal conversational AI applications. It provides a pipeline-based architecture that simplifies the integration of speech recognition, language models and text-to-speech services.

Key Features:

- Pipeline-based architecture for modular design
- Built-in support for multiple AI services
- Easy integration with Amazon Nova 2 Sonic
- Support for real-time audio processing

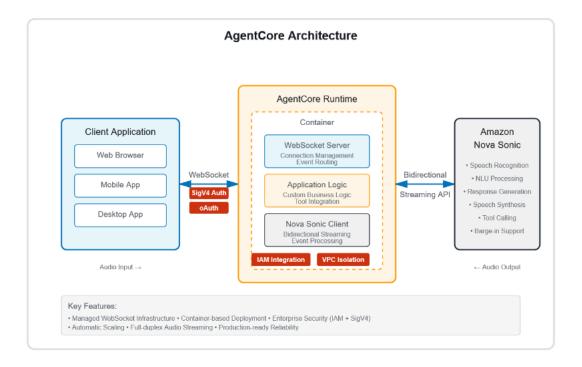
Framework integrations

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 framework for building AI agents with Amazon Bedrock models. It includes tools for agent orchestration, memory management and integration with external services.

LiveKit 141

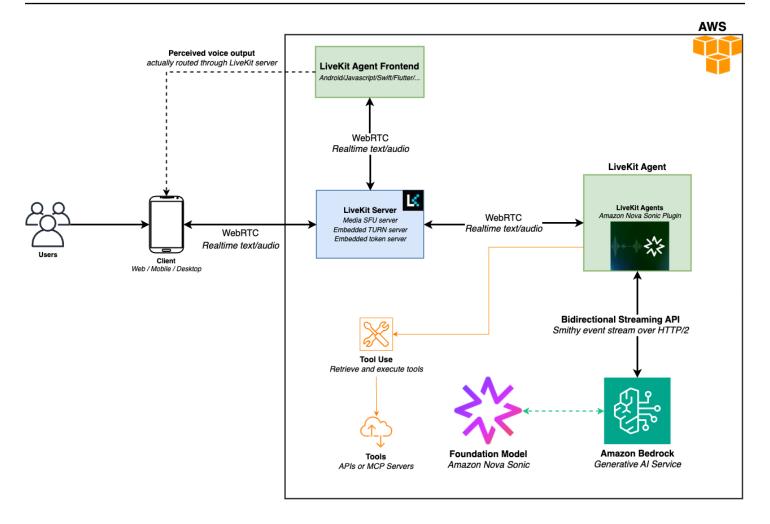


Key features:

- · Agent orchestration and workflow management
- Built-in memory and state management
- Integration with Amazon Bedrock models and tools

LiveKit

LiveKit is an open-source platform for building real-time audio and video applications. Integration with Amazon Nova 2 Sonic enables voice-based interactions in LiveKit applications.

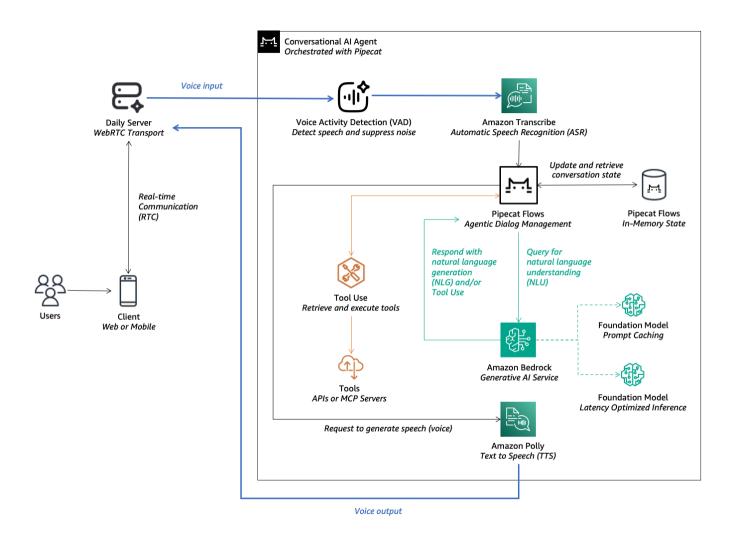


Key features:

- Real-time audio streaming
- Low-latency communication
- Support for multiple participants

Pipecat

Pipecat is a framework for building voice and multimodal conversational AI applications. It provides a pipeline-based architecture for processing audio streams and integrating with various AI services.

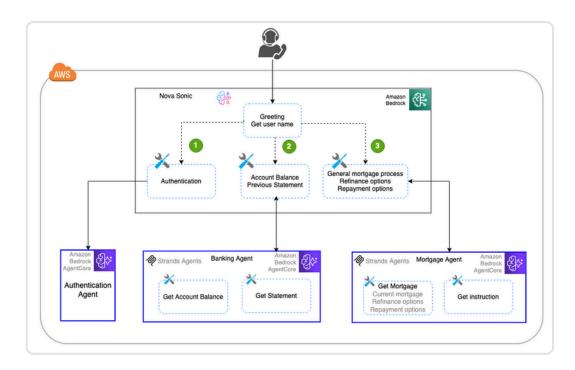


Key features:

- · Pipeline-based audio processing
- Modular component architecture
- Support for multiple AI service integrations

Multi-agentic systems

Multi-agentic architectures allow you to build complex conversational systems where multiple specialized agents work together. Each agent can handle specific tasks or domains, with Amazon Nova 2 Sonic orchestrating the conversation flow.



Benefits:

- Separation of concerns with specialized agents
- Easier maintenance and updates
- Improved scalability
- Better handling of complex workflows

Telephony integration

Amazon Nova 2 Sonic can be integrated with telephony systems to build voice-based contact center solutions. This enables natural language interactions over phone calls with features like call routing, IVR (Interactive Voice Response) and agent assistance.

Use cases:

- Automated customer service
- Interactive voice response (IVR) systems
- Agent assistance and call routing

Building with Amazon Nova 2

Amazon Nova enables building sophisticated AI systems that extend beyond basic text generation. Combine extended thinking, tools, agents and web grounding to create applications that interact with external systems, maintain context across complex workflows and access real-time information.

Topics

- Extended thinking in Amazon Nova 2
- Using tools (function calling)
- Building AI agents
- Web Grounding

Extended thinking in Amazon Nova 2

Amazon Nova 2 introduces extended thinking capabilities that enable the model to engage in deeper reasoning for complex problem-solving tasks.

Understanding extended thinking

Extended thinking represents a fundamental enhancement to how Amazon Nova processes complex requests. When enabled, Amazon Nova creates explicit reasoning chains before generating responses, similar to how humans think through problems step by step.

Two operating modes

- 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.

Quick start: Enabling extended thinking

Extended thinking is controlled through the reasoningConfig parameter.

When using the Converse API, add this to your request:

Extended Thinking 146

```
import boto3
bedrock_runtime = boto3.client('bedrock-runtime', region_name='us-east-1')
response = bedrock_runtime.converse(
    modelId="us.amazon.nova-2-lite-v1:0",
    messages=[{
        "role": "user",
        "content": [{"text": "Generate a step by step quide for cooking Beef
 Wellington."}]
    }],
    inferenceConfig={
        "maxTokens": 2048,
        "temperature": 0.7
    },
    additionalModelRequestFields={
        "reasoningConfig": {
            "type": "enabled",
            "maxReasoningEffort": "medium"
        }
    }
)
# Print the response
content_list = response["output"]["message"]["content"]
for item in content_list:
    if "reasoningContent" in item:
        print("=== REASONING ===")
        print(item["reasoningContent"]["reasoningText"]["text"])
        print()
    elif "text" in item:
        print("=== ANSWER ===")
        print(item["text"])
```

When using the InvokeModel API, add this to your request:

```
import boto3
import json

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

response = bedrock_runtime.invoke_model(
    modelId="us.amazon.nova-2-lite-v1:0",
```

```
body=json.dumps({
        "messages": [{"role": "user", "content": [{"text": "Your prompt here"}]}],
        "inferenceConfig": {
            "maxTokens": 2048,
            "temperature": 0.7
        },
        "reasoningConfig": {
            "type": "enabled",
            "maxReasoningEffort": "medium"
        }
    })
)
# Parse the response
response_body = json.loads(response['body'].read())
# Print the response
content_list = response_body["output"]["message"]["content"]
for item in content_list:
    if "reasoningContent" in item:
        print("=== REASONING ===")
        print(item["reasoningContent"]["reasoningText"]["text"])
        print()
    elif "text" in item:
        print("=== ANSWER ===")
        print(item["text"])
```

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 along with text content blocks:



Reasoning content is displayed as [REDACTED]. You are still charged for reasoning tokens as they contribute to improved response quality.

Response structure 148

Three reasoning effort levels

Amazon Nova 2 provides flexible control over reasoning depth through three effort levels:

Low effort (maxReasoningEffort: "low")

Best for:

- · Code review and improvement suggestions
- · Basic analysis tasks requiring some additional consideration
- Simple problem-solving that benefits from structured approach
- Most day-to-day development tasks with moderate complexity

Characteristics:

- Provides meaningful reasoning enhancement
- Maintains reasonable cost and latency
- Suitable for tasks that require extra thought but not deep analysis

Medium effort (maxReasoningEffort: "medium")

Best for:

- Complex debugging scenarios requiring systematic investigation
- Multi-step calculations with interdependencies
- Moderate planning tasks with multiple constraints
- Analysis requiring evaluation of several alternatives

Characteristics:

- Substantial reasoning depth for problems benefiting from thorough analysis
- Balanced approach between depth and efficiency
- Most common choice for complex technical tasks

High effort (maxReasoningEffort: "high")

Best for:

- Advanced mathematical problems and proofs
- Complex system design with multiple architectural considerations
- · Research tasks requiring deep analysis and synthesis
- Critical decision-making scenarios with significant implications
- Multi-step workflows requiring careful planning and verification

Characteristics:

- Uses up to 32,000 reasoning tokens
- Most thorough reasoning for highly complex, multi-faceted tasks
- Higher latency and cost, but delivers best results for hardest problems

When to enable extended thinking

Enable for complex scenarios

Use extended thinking when tasks require:

- Multi-step reasoning and planning
- Complex problem decomposition
- Careful analysis of trade-offs
- Verification of logical consistency

Keep disabled for efficiency

Keep extended thinking disabled for:

- Simple queries and straightforward tasks
- High-volume applications where latency matters
- Cost-sensitive workloads

Tasks where quick responses are prioritized

Configuration options

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



When using "high", temperature, topP, topK, and maxTokens must be unset. In this mode, the model performs deeper analysis to find the best solution. The thorough processing can generate output exceeding 65,536 tokens. Your request's complexity determines the exact amount, which can reach up to 128K tokens for some problems. 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 use, enabling sophisticated workflows where Amazon Nova can reason through which tools are needed for complex tasks, analyze tool results and determine optimal next steps, course-correct when tool calls don't produce expected results, and chain multiple tools together with reasoning between each step.



Note

Extended thinking can also enhance Amazon Nova's built-in system tools, Amazon Nova Grounding and Amazon Nova Code Interpreter.

Understanding reasoning tokens and pricing

Token types

Reasoning content will be displayed as "[REDACTED]" in the model response. You are still charged for reasoning tokens as they contribute to improved output quality.

- 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} }
```

Best practices

Cost optimization

- Start with "low" effort and increase only if needed
- Use extended thinking selectively for complex tasks
- Monitor reasoning token usage in your applications

Performance optimization

- Set appropriate maxTokens values to accommodate reasoning and response
- Use streaming for complex reasoning tasks to improve perceived performance
- Consider reasoning effort level based on user expectations and time constraints
- Balance thorough analysis with response time requirements

Token limits

- · Low effort: Up to 8,000 reasoning tokens
- Medium effort: Up to 16,000 reasoning tokens

• High effort: Up to 32,000 reasoning tokens

Frequently Asked Questions

Why does Amazon Nova 2 Lite show "[REDACTED]" for reasoning content instead of displaying the model's thinking process?

Reasoning Content Visibility is forthcoming, while these reasoning capabilities already contribute to improve output quality.

The primary focus for this launch ensures that Amazon Nova 2 Lite delivers the best in class intelligence for your tasks, and you will see this reflected in the improved accuracy.

We recognize that visibility into the reasoning process is valuable, and we have heard strong customer interest in understanding how the model thinks through problems. We want to better understand the specific ways you would like to use reasoning visibility—whether you need detailed step-by-step breakdowns for debugging, high-level summaries for transparency, or something in between—so we can expose the right level of detail that is most useful for your applications.

You are still billed for reasoning tokens, as they represent actual computational work that improves your output quality which will be captured in the outputTokens along with the answer tokens. We encourage you to share your use cases and requirements with your AWS account team to help shape this capability.

How do I know if extended thinking is working if reasoning is redacted?

You can confirm extended thinking through:

- Checking for reasoningContent blocks in the response (these only appear when reasoning is active)
- Observing improved quality on complex tasks compared to extended thinking disabled How is this different from Amazon Nova 1.0?

Amazon Nova 1.0 uses latent reasoning only. Thinking happens internally without a dedicated reasoning phase. Amazon Nova 2 Lite adds optional extended thinking, where the model engages in an explicit, computational reasoning phase before responding.

When extended thinking is disabled, Amazon Nova 2 Lite behaves like Amazon Nova 1.0 but with model improvements.

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.

Topics

- Understanding the tool use workflow
- Create a tool
- Call a tool
- Built-in system tools

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.

Tool use 154

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, description and input schema explicitly convey the tool's exact functionality
- Include key differentiators for when to use the tool in the description
- Set temperature to 0 for tool calling to enable greedy decoding

Create a tool 155

- Limit JSON schemas to two layers of nesting for best performance
- Include required and optional parameter denotations in your system prompt

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",
```

Create a tool 156

```
"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.



Run following code sections sequentially (Calling a tool → Processing tool call → Returning toll 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)^{n} 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:

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 Amazon Nova to securely execute Python code in isolated sandbox environments. This enables data analysis, visualization creation, mathematical problem-solving and algorithm testing.

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",
```

Built-in system tools 161

```
"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:

Built-in system tools 162

```
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.

Topics

- Create an agent
- · Invoke an agent

Create an agent

All 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.

Building AI agents 163

Agent design patterns

When designing agents with Nova:

- Enable reasoning on medium or high for best results for extended thinking for complex multistep workflows requiring planning and verification
- Implement tool choice auto to allow flexible tool selection across agent interactions
- Leverage built-in tools (Code Interpreter, Web Grounding) alongside custom tools
- 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

```
import boto3
import json
# Create the client
client = boto3.client('bedrock-runtime', region_name='us-east-1')
tool_config = {
    "tools": [
        {
            "toolSpec": {
                "name": "calculator",
                "description": "Perform mathematical calculations",
                "inputSchema": {
                    "json": {
                         "type": "object",
                         "properties": {
                             "expression": {
                                 "type": "string",
                                 "description": "Mathematical expression to evaluate"
                             }
                         },
```

Create an agent 164

```
"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},
    additionalModelRequestFields={
        "reasoningConfig": {
            "type": "enabled",
            "maxReasoningEffort": "low"
        }
    }
```

Create an agent 165

```
# Print the response
print(json.dumps(response['output']['message'], indent=2))
```

Invoke an agent

Agent invocation involves managing the conversation flow, processing tool calls and maintaining state across multiple interactions.

Stream agent responses

Stream responses to provide real-time visibility into the agent's reasoning and actions:

```
import boto3
import json
# Create the client first
bedrock = boto3.client('bedrock-runtime', region_name='us-east-1')
# Then call converse_stream on the client instance
response = bedrock.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": 1},
    additionalModelRequestFields={
        "reasoningConfig": {
            "type": "enabled",
            "maxReasoningEffort": "low"
        }
    }
)
# Process the streaming response
for event in response['stream']:
    if 'contentBlockDelta' in event:
```

Invoke an agent 166

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

Manage agent state

Maintain conversation history and tool results to preserve context:

```
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":
    # Execute tools and add results to messages
    # Continue conversation loop
```

Agent Best Practices

For more information about Agent Best Practices, see the section called "Best practices".

For guidance on developing Conversational AI agents, see <u>Speech-to-Speech</u>.

Invoke an agent 167

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.

Topics

- How Web Grounding works
- Enable Web Grounding
- Regional availability
- Response structure with citations
- Grounding safety
- · Error handling
- IAM permissions

How Web Grounding works

Web Grounding follows a five-step process:

- 1. **Request configuration** You include the systemTool parameter with nova_grounding in your API request.
- 2. **Search assessment** Amazon Nova analyzes the user query to determine if a web search would improve the response quality.
- 3. **Search execution** If beneficial, Amazon Nova generates search queries and retrieves relevant web content.
- 4. **Result analysis** Amazon Nova processes the search results and extracts relevant information.
- 5. **Response generation** Amazon Nova generates a response incorporating the web content with citations.

Enable Web Grounding

Enable Web Grounding by including the systemTool parameter in your Converse API request:

import boto3
from botocore.config import Config

Web Grounding 168

```
# 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)
```

Enable Web Grounding 169

Regional availability

Web Grounding is available only in US regions. Requests from other regions return an error. Use CRIS profiles to route requests to supported regions when necessary.

Response structure with citations

Responses with Web Grounding include citation information in the citationsContent field:

```
{
  "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

The response text incorporating information from web sources.

citationsContent

Array of citations with source information.

location.web.url

The URL of the cited web source.

Regional availability 170

location.web.domain

The domain name of the cited source.

Grounding safety

Web Grounding implements safety measures to protect data and ensure quality:

- Data stays in AWS All web search operations occur within AWS infrastructure. Your data never leaves AWS.
- Internal search index Web Grounding uses an internal search index maintained by AWS, not external search engines.
- Runtime filtering Content is filtered at runtime to exclude inappropriate or harmful material.

Error handling

Handle common Web Grounding errors:

malformed_tool_use

The model generated an invalid tool use request. Retry with a clearer prompt or different parameters.

max_tokens

The response exceeded the maximum token limit. Increase maxTokens in inferenceConfig or simplify the query.

malformed_model_output

The model generated invalid output. This is typically a transient error - retry the request.

IAM permissions

Web Grounding requires the bedrock: InvokeTool IAM action. Add this permission to your IAM policy:

```
{
    "Version": "2012-10-17",
```

Grounding safety 171

IAM permissions 172

Customizing Amazon Nova 2.0 models

You can customize Amazon Nova 2.0 models with <u>Amazon Bedrock</u> or <u>SageMaker AI</u>, depending on the requirements of your use case, to improve model performance and create a better customer experience.

Customization for the Amazon Nova 2.0 models is provided with responsible AI considerations. The following table summarizes the availability of customization and distillation for Amazon Nova 2.0.

Model Name	Model ID	Amazon Bedrock Fine-tuning	Amazon Bedrock Distillation	SageMaker Al Training Job	SageMaker Al HyperPod
Nova 2 Lite 2.0	amazon.no va-lite-v2:0	Yes	Student	Yes	Yes
2.0 Preview	amazon.no va-pro-v2:0	No	Teacher	No	No
Amazon Nova Sonic	amazon.no va-sonic-v1:0	No	No	No	No
	amazon.no va-omni-v1:0	No	No	No	No

The following table summarizes the training recipe options available. The table includes information about both the service you can use and the inference technique available.

Training recipe	Amazon Bedrock	SageMaker Al Training Jobs	SageMaker Al HyperPod	On demand	Provision throughput
Parameter -efficient supervised fine-tuning	Yes	Yes	Yes	Yes	Yes

Training recipe	Amazon Bedrock	SageMaker Al Training Jobs	SageMaker Al HyperPod	On demand	Provision throughput
Full rank supervised fine-tuning	No	Yes	Yes	No	Yes
Parameter- efficient fine- tuning Direct Preference Optimization	No	Yes	Yes	Yes	Yes
Full rank Direct Preference Optimization	No	Yes	Yes	No	Yes
Proximal policy optimizat ion reinforce ment learning	No	No	Yes	No	Yes
Distillat ion - 2.0 as teacher	Yes	No	Yes	Yes	Yes
Continuous pre-training	No	No	Yes	No	Yes

Topics

- Customization overview
- Customization on Amazon Bedrock
- Customization on SageMaker Al

- Choosing the right customization approach
- With Amazon Bedrock
- With Amazon SageMaker Al

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 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 taskspecific 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

Customization overview 175

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.

Key features:

- Fully managed infrastructure with no cluster 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

For detailed instructions, see the Amazon Bedrock documentation.

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. Requires 100+ examples (2,000-10,000 recommended for optimal results).

Reinforcement Fine-Tuning (RFT)

Optimizes models using reward signals for complex problem-solving tasks like mathematical reasoning, code generation and scientific analysis. Supports both single-turn (Lambda-based) and multi-turn (custom infrastructure) scenarios. Best used after SFT establishes baseline capabilities.

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 Al 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.

Choosing the right customization approach

To decide the best training approach for your use case, consider what each method is best suited for:

Supervised Fine-Tuning (SFT)

Best for teaching specific response styles and domain knowledge. For standard SFT capabilities, see Amazon Nova customization on SageMaker training jobs.

With Nova Forge, you can access advanced data mixing capabilities to combine your custom datasets with Amazon's proprietary training data.

Reinforcement Fine-Tuning (RFT)

Best for aligning model behavior with complex preferences using measurable feedback.

With Nova Forge, you can access multi-turn RFT with bring-your-own-orchestration (BYOO) capabilities.

Continued Pre-Training (CPT)

Best for teaching domain knowledge at scale. For standard CPT capabilities, see <u>Continued pretraining</u> for Amazon Nova.

With Nova Forge, you can access intermediate checkpoints and data mixing for domain-specific pre-training.

With Amazon Bedrock

Model customization is the process of providing training data to a model in order to improve its performance for specific use-cases. You can customize Amazon Nova in Amazon Bedrock to improve its performance and create a better customer experience. For specific information about customizing Amazon Nova models, see the following sections. For general information about customizing models in Amazon Bedrock, see Customize your model to improve its performance for your use case in the Amazon Bedrock user guide.

Topics

- Amazon Nova 2.0 advanced fine-tuning capabilities
- Fine-tuning Amazon Nova models
- On-demand inference on Custom Models

Amazon Nova 2.0 advanced fine-tuning capabilities

Amazon Nova 2.0 introduces enhanced fine-tuning capabilities on Amazon Bedrock, including supervised fine-tuning with reasoning content and reinforcement fine-tuning with reward-based optimization.

Supervised fine-tuning on Amazon Nova 2.0

Amazon Nova 2.0 supervised fine-tuning uses the same Converse API format as Amazon Nova 1.0 with optional reasoning content fields, allowing you to train models that show their thinking process before generating final answers.

Key features

- Support for text, image and video inputs in user content blocks
- Optional reasoning content in assistant responses to capture intermediate thinking steps
- Homogeneous dataset requirements (choose text-only, text+image, or text+video)
- Support for PNG, JPEG and GIF images
- Support for MOV, MKV and MP4 videos
- Configurable reasoning modes for training optimization

For detailed information about data preparation, format specifications and reasoning modes, see <u>Supervised fine-tuning on Amazon Nova 2.0</u> in the Amazon Bedrock user guide.

Reinforcement fine-tuning (RFT)

Reinforcement fine-tuning optimizes Amazon Nova models using measurable feedback signals rather than exact correct answers. This approach excels when you can reliably measure response quality but defining exact correct outputs is challenging.

Key features

- Reward-based optimization using custom reward functions
- · Support for subjective or multifaceted quality assessment
- Ideal for mathematical problem-solving and code generation
- Effective for scientific reasoning and structured data analysis
- · Programmatic success verification through execution results
- OpenAl Reinforcement Fine-Tuning format with messages and reference answers
- Evaluation-first approach to validate reward functions before scaling

For detailed information about RFT data format, dataset recommendations and training characteristics, see <u>Reinforcement fine-tuning (RFT) for Amazon Nova models</u> in the Amazon Bedrock user guide.

Fine-tuning Amazon Nova models

You can customize the Amazon Nova models using the <u>fine-tuning method</u> 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.

Topics

- · Available models
- Performing custom fine-tuning
- Encryption of Amazon Nova model customization jobs and artifacts

- Preparing data for fine-tuning Understanding models
- Selecting hyperparameters

Available models

Fine-tuning is available for the Nova 2 Lite Amazon Nova 2.0 model and their supported text, image and video modalities.

Performing custom fine-tuning

To perform custom fine-tuning with Amazon Nova models, you do the following:

- Create a training dataset and a validation dataset (if applicable) for your customization task.
 For more information about preparing data, see <u>Preparing data for fine-tuning Understanding models</u>.
- If you plan to use a new custom IAM role, follow the instructions in <u>Create a service role for model customization</u> to create an IAM role with access to your data in Amazon S3 buckets.
 Or you can use an existing role or let the console automatically create a role with the proper permissions.
- 3. (Optional) Configure Encryption of Amazon Nova model customization jobs and artifacts, VPC, or both, for extra security.
- 4. <u>Create a Fine-tuning job</u>, controlling the training process by adjusting the <u>hyperparameter</u> values.
- 5. Analyze the results by looking at the training or validation metrics or by using model evaluation.
- 6. <u>Purchase Provisioned Throughput</u> or <u>On-demand inference on Custom Models</u> for your newly created custom model.
- 7. <u>Use your custom model</u> as you would a base model in Amazon Bedrock tasks, such as model inference.

Encryption of Amazon Nova model customization jobs and artifacts

For information regarding encryption of your model customization jobs and artifacts in Amazon Bedrock, see Encryption of model customization jobs and artifacts.

Topics

• Permissions and key policies for custom Amazon Nova models

• Set up key permissions for encrypting and invoking custom models

Permissions and key policies for custom Amazon Nova models

The following statements are necessary to establish permissions for your KMS key.

PermissionsModelCustomization statement

In the Principal field, add accounts that you want to allow the Decrypt, GenerateDataKey, DescribeKey and CreateGrant operations to the list that the AWS subfield maps to. If you use the kms:ViaService condition key, you can add a line for each region, or use * in place of \${region} to allow all regions that support Amazon Bedrock.

```
{
    "Sid": "PermissionsModelCustomization",
    "Effect": "Allow",
    "Principal": {
        "AWS": [
            "arn:aws:iam::${account-id}:role/${customization-role}"
        ]
    },
    "Action": [
        "kms:Decrypt",
        "kms:GenerateDataKey",
        "kms:DescribeKey",
        "kms:CreateGrant"
    ],
    "Resource": "*",
    "Condition": {
        "StringLike": {
            "kms:ViaService": [
                "bedrock.${region}.amazonaws.com"
            ]
        }
    }
}
```

PermissionsModelInvocation statement

In the Principal field, add accounts that you want to allow the Decrypt and GenerateDataKey operations to the list that the AWS subfield maps to. If you use the kms: ViaService condition

key, you can add a line for each region, or use * in place of \${region} to allow all regions that support Amazon Bedrock.

```
{
    "Sid": "PermissionsModelInvocation",
    "Effect": "Allow",
    "Principal": {
        "AWS": [
            "arn:aws:iam::${account-id}:user/${invocation-role}"
        ٦
    },
    "Action": [
        "kms:Decrypt",
        "kms:GenerateDataKey"
    ],
    "Resource": "*",
    "Condition": {
        "StringLike": {
            "kms:ViaService": [
                "bedrock.${region}.amazonaws.com"
            ]
        }
    }
}
```

PermissionsNovaProvisionedThroughput statement

When you create provisioned throughput for your custom Amazon Nova model, Amazon Bedrock performs inference and deployment optimizations on the model. In this process, Amazon Bedrock uses the same KMS key used to create the custom model to maintain the highest level of security as that of the custom model itself.

```
"kms:GenerateDataKey"
],
"Resource": "*",
"Condition": {
    "ForAnyValue:StringEquals": {
        "kms:EncryptionContextKeys": "aws:bedrock:custom-model"
     }
}
```

Set up key permissions for encrypting and invoking custom models

If you plan to encrypt a model that you customize with a KMS key, the key policy for the key will depend on your use case. Expand the section that corresponds to your use case:

The roles that will customize the model and the roles that will invoke the model are the same

If the roles that will invoke the custom model are the same as the roles that will customize the model, you only need the PermissionsModelCustomization and PermissionsNovaProvisionedThroughput statements from permission statements.

- 1. In the Principal field, add accounts that you want to allow to customize and invoke the custom model to the list that the AWS subfield maps to in the PermissionsModelCustomization statement.
- 2. The PermissionsNovaProvisionedThroughput statement should be added by default to the key policy with bedrock.amazonaws.com as an allowed service principal with a condition that kms:EncryptionContextKeys are used.

JSON

```
]
            },
            "Action": [
                "kms:Decrypt",
                "kms:GenerateDataKey",
                "kms:DescribeKey",
                "kms:CreateGrant"
            ],
            "Resource": "*",
            "Condition": {
                "StringLike": {
                    "kms:ViaService": [
                        "bedrock.us-east-1.amazonaws.com"
                    ]
                }
            }
        },
            "Sid": "PermissionsNovaProvisionedThroughput",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "bedrock.amazonaws.com"
                ]
            },
            "Action": [
                "kms:Decrypt",
                "kms:GenerateDataKey"
            ],
            "Resource": "*",
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "kms:EncryptionContextKeys": "aws:bedrock:custom-model"
                }
            }
        }
   ]
}
```

The roles that will customize the model and the roles that will invoke the model are different

If the roles that will invoke the custom model are different from the role that will customize the model, you need all three of the permission statements. Modify the statements in the following policy template as follows:

- 1. In the Principal field, add accounts that you want to allow to only customize the custom model to the list that the AWS subfield maps to in the PermissionsModelCustomization statement.
- 2. In the Principal field, add accounts that you want to allow to only invoke the custom model to the list that the AWS subfield maps to in the PermissionsModelInvocation statement.
- 3. The PermissionsNovaProvisionedThroughput statement should be added by default to the key policy with bedrock.amazonaws.com as allowed service principal with a condition that kms:EncryptionContextKeys are used.

JSON

```
"Version": "2012-10-17",
"Id": "PermissionsCustomModelKey",
"Statement": [
    {
        "Sid": "PermissionsModelCustomization",
        "Effect": "Allow",
        "Principal": {
            "AWS": [
                "arn:aws:iam::111122223333:user/customization-role"
            1
        },
        "Action": [
            "kms:Decrypt",
            "kms:GenerateDataKey",
            "kms:DescribeKey",
            "kms:CreateGrant"
        ],
        "Resource": "*",
        "Condition": {
            "StringLike": {
                "kms:ViaService": [
                     "bedrock.us-east-1.amazonaws.com"
```

```
]
        }
    }
},
{
    "Sid": "PermissionsModelInvocation",
    "Effect": "Allow",
    "Principal": {
        "AWS": [
            "arn:aws:iam::111122223333:user/invocation-role"
        ]
    },
    "Action": [
        "kms:Decrypt",
        "kms:GenerateDataKey"
    ],
    "Resource": "*",
    "Condition": {
        "StringLike": {
            "kms:ViaService": [
                "bedrock.us-east-1.amazonaws.com"
        }
    }
},
{
    "Sid": "PermissionsNovaPermissionedThroughput",
    "Effect": "Allow",
    "Principal": {
        "Service": [
            "bedrock.amazonaws.com"
        ]
    },
    "Action": [
        "kms:Decrypt",
        "kms:GenerateDataKey"
    ],
    "Resource": "*",
    "Condition": {
        "ForAnyValue:StringEquals": {
            "kms:EncryptionContextKeys": "aws:bedrock:custom-model"
        }
    }
}
```

] }

Preparing data for fine-tuning Understanding models

The following are guidelines and requirements for preparing data for fine-tuning Understanding models:

- 1. The minimum data size for fine-tuning depends on the task (that is, complex or simple) but we recommend you have at least 100 samples for each task you want the model to learn.
- 2. We recommend using your optimized prompt in a zero-shot setting during both training and inference to achieve the best results.
- 3. Traning and validation datasets must be JSONL files, where each line is a JSON object corresponding to a record. These file names can consist of only alphanumeric characters, underscores, hyphens, slashes and dots.
- 4. Image and video constraints
 - a. Dataset can't contain different media modalities. That is, the dataset can either be text with images or text with videos.
 - b. One sample (single record in messages) can have multiple images
 - c. One sample (single record in messages) can have only 1 video
- 5. schemaVersion can be any string value
- 6. The (optional) system turn can be a customer-provided custom system prompt.
- 7. Supported roles are user and assistant.
- 8. The first turn in messages should always start with "role": "user". The last turn is the bot's response, denoted by "role": "assistant".
- 9. The image.source.s3Location.uri and video.source.s3Location.uri must be accessible to Amazon Bedrock.
- 10Your Amazon Bedrock service role must be able to access the image files in Amazon S3. For more information about granting access, see <u>Create a service role for model customization</u>
- 11. The images or videos must be in the same Amazon S3 bucket as your dataset. For example, if your dataset is in s3://amzn-s3-demo-bucket/train/train.jsonl, then your images or videos must be in s3://amzn-s3-demo-bucket
- 12. The terms User:, Bot:, Assistant:, System:, <image>, <video> and [EOS] are reserved keywords. If a user prompt or system prompt starts with any of these keywords, or have these

keywords anywhere in their prompts, your training job will fail due to data issues. If you need to use these keywords for your use case, you must substitute it for a different keyword with a similar meaning so that your training can proceed.

Topics

- Example dataset formats
- Dataset constraints

Example dataset formats

The following example dataset formats provide a guide for you to follow.

Text-only custom fine tuning format

The following example is for custom fine tuning over text only.

```
// train.jsonl
  "schemaVersion": "bedrock-conversation-2024",
  "system": [
    {
      "text": "You are a digital assistant with a friendly personality"
  ],
  "messages": [
      "role": "user",
      "content": [
        {
          "text": "What is the capital of Mars?"
      ]
    },
    {
      "role": "assistant",
      "content": [
          "text": "Mars does not have a capital. Perhaps it will one day."
        }
      ]
    }
```

```
}
```

Single image custom fine tuning format

The following example is for custom fine tuning over text and a single image.

```
// train.jsonl{
    "schemaVersion": "bedrock-conversation-2024",
    "system": [{
        "text": "You are a smart assistant that answers questions respectfully"
    }],
    "messages": [{
            "role": "user",
            "content": [{
                    "text": "What does the text in this image say?"
                },
                {
                    "image": {
                         "format": "png",
                         "source": {
                             "s3Location": {
                                 "uri": "s3://your-bucket/your-path/your-image.png",
                                 "bucket0wner": "your-aws-account-id"
                             }
                         }
                    }
                }
            ]
        },
            "role": "assistant",
            "content": [{
                "text": "The text in the attached image says 'LOL'."
            }]
        }
    ]
}
```

Video custom fine tuning format

The following example is for custom fine tuning over text and video.

```
{
    "schemaVersion": "bedrock-conversation-2024",
    "system": [{
        "text": "You are a helpful assistant designed to answer questions crisply and
 to the point"
    }],
    "messages": [{
            "role": "user",
            "content": [{
                    "text": "How many white items are visible in this video?"
                },
                {
                    "video": {
                         "format": "mp4",
                         "source": {
                             "s3Location": {
                                 "uri": "s3://your-bucket/your-path/your-video.mp4",
                                 "bucketOwner": "your-aws-account-id"
                             }
                         }
                    }
                }
            ]
        },
            "role": "assistant",
            "content": [{
                "text": "There are at least eight visible items that are white"
            }]
        }
    ]
}
```

Dataset constraints

Amazon Nova applies the following constraints on model customizations for Understanding models.

Model	Minimum Samples	Maximum Samples	Context Length
Nova 2 Lite	8	20k	32k

Image and video constraints

Maximum images 10/sample

Maximum image file size 10 MB

Maximum videos 1/sample

Maximum video length/duration 90 seconds

Maximum video file size 50 MB

Supported media formats

• Image - png, jpeg, gif, webp

• Video - mov, mkv, mp4, webm

Selecting hyperparameters

We recommend that you start with the default hyperparameters, which are based on our assessment across tasks of different complexity and data sizes. But you might need to adjust and optimize certain hyperparameters based on your use case as you evaluate the performance.

Topics

Guidance for adjusting hyperparameters

Guidance for adjusting hyperparameters

The following general guidance can help you determine how to adjust the hyperparameters when fine-tuning a model.

- Tweak the Epochs Based on Sample Size: The default epoch number is 2, which works for most cases. In general, larger data sets require less epochs to converge, while smaller data sets require a larger training epoch to converge. We recommend that you tweak your epochs based on data sample size.
- **Prompt Structure:** Optimizing the prompting strategy can improve the performance of a finetuned model. It is worth investing time to optimize the prompt templates on existing models

before using them for fine-tuning. We recommend that you abide by the prompting best practices followed by Amazon Nova to achieve the best performance results.

- Increasing Effective Epochs: As Amazon Bedrock Customization service limits the epochs to 5, this might hinder under-training on smaller datasets. Hence, for smaller samples (<1K) we recommend that you duplicate the data to make the "Effective epoch" higher. For example, if the dataset is duplicated to 2x times, training 5 epochs would be effectively mean 10 epochs on the original data. For larger samples (up to 5k) we recommend 2 epochs, for sample sizes greater than 5k we recommend using 1 epoch for faster convergence.
- Avoid a Large Warm up Number for Small Sample: The learning rate will gradually increase
 to the set value during warm up. Therefore, you should avoid a large warm up number for a
 small training sample because your learning rate might never reach the set value during the
 training process. We recommend setting the warmup steps by dividing the dataset size by 640
 for Amazon Nova Micro, 160 for Nova 2 Lite and 320 for rounding the number.
- **Bigger learning rate for smaller models:** Amazon Nova Micro may benefit from a larger learning rate due to the effective batch size used on the back end.
- **Quality over Quantity**: The quality of the training data is more important than the quantity. Begin with a small, high-quality dataset for initial fine-tuning and performance evaluation, then iterate and expand based on the results.
- **Data Refinement:** For certain use cases, cleaning and improving the training data using Amazon Nova models might be beneficial. This refined data can then be used to fine-tune smaller models effectively.
- **Diversify and Augment:** You can improve the model performance by increasing the variation and diversity in your customization dataset. Your fine-tuning data and evaluation data should be consistent with the actual traffic distribution that the model will see.
- **Distillation:** Nova 2 Lite and can be used to generate training data for fine-tuning Amazon Nova Micro models. This method can be very effective if the larger models are already highly capable at the target task.

When to Distill or Fine Tune?

We recommend that you use distillation when

• You do not have labeled data and the larger models in the family (aka, Teacher models) are highly capable on the target task.

• Larger models are better than smaller model on the target task but you need the latency and cost profile of smaller model with the accuracy of larger models.

We recommend that you use custom fine-tuning when

- You do not see good performance, even on a larger model and there is a intelligence gap in the model.
- Your use case is in a very narrow domain and not general enough for the model to know about it.

Hyperparameters for Understanding models

There are 3 hyperparameters that you can adjust when fine-tuning an Understanding model.

Hyperpara meter	Туре	Minimum	Maximum	Default
Epochs	integer	1	5	2
Learning rate	float	1.00E-06	1.00E-04	1.00E-05
Learning rate warmup steps	integer	0	20	10

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.

On-demand inference 194

- 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 <u>Amazon Bedrock customization API</u> 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	RPM per Custom Model	TPM per Custom Model
Model	Deployment	Deployment
Nova 2 Lite	2,000	4,000,000

To learn more about the quotas available for Amazon Nova, see Quotas for Amazon Nova.

On-demand inference 195

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.

With Amazon SageMaker Al

Developers sometimes require greater control and flexibility when customizing Amazon Nova models. With SageMaker AI, you can leverage larger labeled datasets, perform deeper customization (such as preference optimization or full rank fine-tuning) and access enhanced capabilities for hyperparameter optimization and active train-loss curve analysis. You can perform these advanced customization features on Amazon Nova models in the SageMaker AI platform.

With Amazon Nova customization using SageMaker AI, you can launch SageMaker training jobs or launch jobs on SageMaker AI Hypepod.

Topics

- SageMaker Al Training Jobs
- SageMaker AI HyperPod training
- Evaluate your custom training jobs

SageMaker AI Training Jobs

Customizing Amazon Nova models with Amazon SageMaker Training Jobs follows a structured workflow designed to simplify the complex process of fine-tuning large language models. This end-to-end workflow encompasses model training, evaluation and deployment for inference. For more information, see Customizing Amazon Nova models in the Amazon SageMaker AI Developer Guide.

With Amazon SageMaker AI, you can fine-tune existing pre-trained foundation models, such as Amazon Nova, without training your own models from scratch. The following sections detail the fine-tuning options in SageMaker AI when working with Amazon Nova foundation models.

Topics

Full-rank fine-tuning

With Amazon SageMaker AI 196

- · Low-rank adapter fine-tuning
- · Direct preference optimization
- Distillation

Full-rank fine-tuning

Full-rank fine-tuning modifies all foundation model parameters to optimize its performance for specific tasks or domains. This comprehensive approach updates the entire model architecture, enabling deeper adaptations than adapter-based methods. For more information, see Fine-tune foundation models.

How full-rank fine-tuning works

During full-rank fine-tuning, the model learns by updating all its parameters using your training data. This process of full-rank fine-tuning:

- Allows the model to develop specialized knowledge for your domain.
- Enables significant changes to the model's underlying representations.
- Requires more computational resources compared to adapter-based methods but can achieve better task-specific performance.

When to choose full-rank fine-tuning

We recommend using full-rank fine-tuning in the following scenarios:

- When LoRA PEFT fine-tuning doesn't achieve the desired performance levels.
- For specialized domains that require deep expertise (such as medical, legal, or technical fields).
- When you have large, high-quality datasets for your use case.
- When accuracy requirements outweigh computational cost considerations.
- For applications that require significant deviation from the base model's behavior.

Low-rank adapter fine-tuning

The most effective and cost-efficient method to enhance base model performance is through low-rank adapter parameter-efficient fine-tuning (LoRA PEFT). The underlying principle of LoRA PEFT

SageMaker Al Training Jobs 197

is that only a small number of additional weights requires updating to adapt it to new tasks or domains.

LoRA PEFT efficiently fine-tunes foundation models by introducing low-rank, trainable weight matrices into specific model layers, reducing the number of trainable parameters while maintaining model quality. A LoRA PEFT adapter augments the base foundation model by incorporating lightweight adapter layers that modify the model's weights during inference, while keeping the original model parameters intact. This approach is also considered one of the most cost-effective fine-tuning techniques. For more information, see Fine-tune models with adapter inference components.

When to choose LoRA PEFT

We recommend using LoRA PEFT in the following scenarios:

- You should generally start with LoRA PEFT over other fine-tuning methods because it's a fast training procedure.
- LoRA PEFT is effective in cases where the base model's performance is already satisfactory. In this case, the goal of LoRA PEFT is to enhance its capabilities across multiple related tasks, such as text summarization or language translation. LoRA PEFT's regularization properties also help prevent overfitting and mitigate the risks of the model "forgetting" the source domain. This ensures the model remains versatile and adaptable to various applications.
- You can use LoRA PEFT for instruction fine-tuning scenarios with relatively small datasets. LoRA
 PEFT performs better with smaller, task-specific datasets than broader, larger datasets.
- For large, labeled datasets that exceed the Amazon Bedrock customization data limits, you can use LoRA PEFT on SageMaker AI to generate better results.
- If you've already achieved promising results through Amazon Bedrock fine-tuning, LoRA PEFT on SageMaker AI can help further optimize the model hyperparameters.

Direct preference optimization

Direct preference optimization (DPO) is an efficient fine-tuning method for foundation models that uses paired comparison data to align model outputs with human preferences. This approach provides the direct optimization of model behavior based on human feedback about which responses are more desirable.

Why DPO matters

SageMaker Al Training Jobs 198

Foundation models trained on large-scale data often generate outputs that might be factually correct but fail to align with specific user needs, organizational values, or safety requirements. DPO addresses this gap by allowing you to do the following:

- Fine-tune models toward desired behavior patterns.
- Reduce unwanted outputs or harmful responses.
- Align model responses with brand voice and communication guidelines.
- Improve response quality based on domain expert feedback.

How DPO works

DPO uses paired examples where human evaluators indicate which of two possible responses is preferred. The model learns to maximize the likelihood of generating preferred responses while minimizing undesired ones. You can implement DPO by using either of the following techniques:

- Full-rank DPO: Updates all model parameters to optimize for preferred responses.
- LoRA-based DPO: Uses lightweight adapters to learn preference alignments, requiring fewer computational resources.

When to choose DPO

We recommend using DPO in the following scenarios:

- Optimizing for subjective outputs that require alignment with specific human preferences.
- Adjusting the model's tone, style, or content characteristics to match desired response patterns.
- Making targeted improvements to an existing model based on user feedback and error analysis.
- Maintaining consistent output quality across different use cases.
- Implementing safety guardrails through preferred response patterns.
- Training with reward-free reinforcement learning.
- Using only preference data instead of graded or labeled data.
- Improving the model in nuanced alignment tasks, such as helpfulness, harmlessness, or honesty.

DPO is effective for iteratively refining model behavior through carefully curated preference datasets that demonstrate desired versus undesired outputs. The method's flexibility in supporting

SageMaker Al Training Jobs 199

full-rank and LoRA-based approaches lets you choose the most suitable implementation based on your computational resources and specific requirements.

Distillation

Model distillation is a method that transfers knowledge from large, advanced models to smaller, efficient ones. With Amazon Nova models, a larger "teacher" model (like or Amazon Nova Premier) passes its capabilities to a smaller "student" model (like Nova 2 Lite or Amazon Nova Micro). This creates a customized model that maintains high performance while using fewer resources.

For information on how to complete this using SageMaker AI Training Jobs, see <u>Amazon Nova</u> distillation.

SageMaker AI HyperPod training

You can customize Amazon Nova models using <u>Amazon Nova recipes</u> and train them on SageMaker AI HyperPod. A recipe is a YAML configuration file that provides details to SageMaker AI on how to run your model customization job.

SageMaker AI HyperPod offers high-performance computing with optimized GPU instances and Amazon FSx for Lustre storage, robust monitoring through integration with tools like TensorBoard, flexible checkpoint management for iterative improvement, seamless deployment to Amazon Bedrock for inference and efficient scalable multi-node distributed training-all working together to provide organizations with a secure, performant and flexible environment to tailor Amazon Nova models to their specific business requirements.

Amazon Nova customization on SageMaker AI HyperPod stores model artifacts including model checkpoints in a service-managed Amazon S3 bucket. Artifacts in the service-managed bucket are encrypted with SageMaker-managed KMS keys. Service-managed Amazon S3 buckets don't currently support data encryption using customer managed keys. You can use this checkpoint location for evaluation jobs or Amazon Bedrock inference.

This section provides details about the Amazon Nova model parameters that you can tune with SageMaker AI HyperPod, when you might want to tune them and how they might affect model performance. The parameters are presented by training technique. For information on how to submit a job, see Running a SageMaker training job.

Continued pre-training (CPT)

Continued Pre-Training (CPT) is a technique that extends a pre-trained language model's capabilities by training it on new domain-specific data while preserving its general language

understanding. Unlike fine-tuning, CPT uses the same unsupervised objectives as the original pre-training (such as masked or causal language modeling) and doesn't modify the model's architecture.

CPT is particularly valuable when you have large amounts of unlabeled domain-specific data (like medical or financial text) and want to improve the model's performance in specialized areas without losing its general capabilities. This approach enhances zero-shot and few-shot performance in targeted domains without requiring extensive task-specific fine-tuning.

For detailed instructions about using CPT with Amazon Nova model customization, see the Continued Pre-Training (CPT) section from SageMaker user guide.

Supervised fine-tuning (Full FT, PEFT)

Supervised fine-tuning (SFT) is the process of providing a collection of prompt-response pairs to a foundation model to improve the performance of a pre-trained foundation model on a specific task. The labeled examples are formatted as prompt-response pairs and phrased as instructions. This fine-tuning process modifies the weights of the model.

You should use SFT when you have domain-specific data that requires providing specific prompt-response pairs for optimal results. Both full-rank SFT and parameter-efficient SFT are available.

For detailed instructions about using SFT with Amazon Nova model customization, see the Supervised Fine-Tuning (Full FT, PEFT) section from SageMakeruser guide.

Direct preference optimization (DPO)

DPO is an advanced technique that fine-tunes models based on human preferences rather than fixed labels. It uses paired examples where humans have indicated which response is better for a given prompt. The model learns to generate outputs that align with these preferences, helping to improve response quality, reduce harmful outputs and better align with human values. DPO is particularly valuable for refining model behavior after initial SFT.

Both full-rank DPO and low-rank adapter (LoRA) DPO are available.

For detailed instructions about using DPO with Amazon Nova model customization, see the <u>Direct Preference Optimization (DPO)</u> section from SageMakeruser guide.

Proximal policy optimization (PPO)

Proximal policy optimization (PPO) is the process of using several machine learning models to train and score a model. The PPO process involves five key components:

- Actor train model (or policy model): A supervised fine-tuning (SFT) model that gets fine-tuned and updated every epoch. The updates are made by sampling prompts, generating completions and updating weights using a clipped-surrogate objective. This limits the per-token log-profitability change so that each policy step is *proximal* to the previous one, preserving training stability.
- Actor generation model: A model that generates prompt completions or responses to be judged by the reward model and critic model. The weights of this model are updated from the actor train or policy model each epoch.
- **Reward model**: A model with fixed (frozen) weights that's used to score the actor generation model, providing feedback on response quality.
- **Critic model**: A model with trainable (unfrozen) weights that's used to score the actor generation model. This score is often viewed as an estimate of the total reward the actor receives when generating the remaining tokens in a sequence.
- Anchor model: An SFT model with frozen weights that is used to calculate the Kullback-Leibler (KL) divergence between the actor train model and the original base model. The anchor model ensures that the updates to the actor model are not too drastic compared to the base model. Drastic changes can lead to instability or performance degradation.

Together, these components create a sophisticated reinforcement learning system that can optimize language model outputs based on defined reward criteria while maintaining stable training dynamics.

For detailed instructions about using PPO with Amazon Nova model customization, see the Proximal Policy Optimization (PPO) section from SageMakeruser guide.

Iterative training

Iterative training allows you to develop more sophisticated training pipelines to tune Amazon Nova models. By chaining training modules, you're able to layer training techniques to customize your models exactly to your needs.

To begin, you start by training Amazon Nova using one of the techniques described in SageMaker Al HyperPod training. In the output S3 location defined during training, locate the manifest.json file. This file contains the value checkpoint_s3_bucket that indicates where the output model is defined. You can utilize this output location as the model_name_or_path value in future training runs.

For detailed instructions about using iterative training with Amazon Nova model customization, see the Iterative training section from SageMakeruser guide.

Evaluate your custom training jobs

An evaluation recipe is a YAML configuration file that defines how your Amazon Nova model evaluation job is executed. With this recipe, you can assess the performance of a base or trained model against common benchmarks or your own custom datasets. Metrics can be stored in Amazon S3 or TensorBoard. The evaluation provides quantitative metrics that help you assess model performance across various tasks to determine if further customization is needed.

Model evaluation is an offline process, where models are tested against fixed benchmarks with predefined answers. They are not assessed in real-time or against live user interactions. For real-time evaluations, you can evaluate the model after it is deployed to Amazon Bedrock by calling the Amazon Bedrock runtime APIs.

For detailed instructions about evaluating you trained Amazon Nova models, see the <u>Evaluating</u> your trained model section from SageMaker user guide.

Code and Troubleshooting

This chapter provides code examples and troubleshooting guidance to help you work effectively with Amazon Nova models.

Topics

- Code library
- Troubleshooting

Code library

This section provides code examples for common Amazon Nova operations using the Converse API.

Basic request

Send a basic text request to Amazon Nova models.

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
 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,
        }
    },
)
# 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 images and text together by embedding image data directly in the request.

Non-streaming

```
"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-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-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 images and text together by referencing images stored in S3.

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",
                            # "bucket0wner": "<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-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",
                            # "bucket0wner": "<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-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?"
        ],
    }
1
# 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

# Create the Bedrock Runtime client
bedrock = boto3.client(
    "bedrock-runtime",
    region_name="us-east-1",
    config=Config(read_timeout=3600),
)

# Define the tool configuration
tool_config = {
```

```
"tools": [
        {
            "toolSpec": {
                "name": "get_weather",
                "description": "Get the current weather for a location",
                "inputSchema": {
                     "json": {
                         "type": "object",
                         "properties": {
                             "location": {
                                 "type": "string",
                                 "description": "The city and state, e.g. San
 Francisco, CA",
                             }
                         },
                         "required": ["location"],
                    }
                },
            }
        }
    ]
}
messages = [
    {
        "role": "user",
        "content": [{"text": "What's the weather like in Seattle?"}],
    }
]
# Invoke the model
response = bedrock.converse(
    modelId="us.amazon.nova-2-lite-v1:0", messages=messages, toolConfig=tool_config
)
# Extract tool use from response
content_list = response["output"]["message"]["content"]
for content in content_list:
    if "toolUse" in content:
        tool_use = content["toolUse"]
        print(f"Tool: {tool_use['name']}")
        print(f"Input: {tool_use['input']}")
```

Streaming

```
import boto3
from botocore.config import Config
import json
# Create the Bedrock Runtime client
bedrock = boto3.client(
    "bedrock-runtime",
    region_name="us-east-1",
    config=Config(connect_timeout=3600, read_timeout=3600),
)
# Define the tool configuration
tool_config = {
    "tools": [
        {
            "toolSpec": {
                "name": "get_weather",
                "description": "Get the current weather for a location",
                 "inputSchema": {
                     "json": {
                         "type": "object",
                         "properties": {
                             "location": {
                                 "type": "string",
                                 "description": "The city and state, e.g. San
 Francisco, CA",
                             }
                         },
                         "required": ["location"],
                     }
                },
            }
        }
    ]
}
messages = [
    {
        "role": "user",
        "content": [{"text": "What's the weather like in Seattle?"}],
]
```

```
# 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_tool_use = None
for event in response["stream"]:
    if "contentBlockStart" in event:
        start = event["contentBlockStart"]["start"]
        if "toolUse" in start:
            current_tool_use = start["toolUse"]
            print(f"Tool: {current_tool_use['name']}")
    elif "contentBlockDelta" in event:
        delta = event["contentBlockDelta"]["delta"]
        if "toolUse" in delta:
            tool_input = json.loads(delta["toolUse"]["input"])
            print(f"Input: {tool_input}")
```

InvokeModel API

Use the InvokeModel API for direct model invocation with JSON request/response format.

Non-streaming

```
"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

```
"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

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

Troubleshooting 226

Regional availability issues

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

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(
        connect_timeout=3600, # 60 minutes
        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.

Regional availability issues 227 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

Reasoning mode errors 228

```
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:

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)
        }
}
```

Request and response schema 233

```
}
       },
         "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
               "bucket0wner": "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/assistant) and content
- inferenceConfig: Controls model output behavior (temperature, tokens, etc.)
- toolConfig: Tool specifications for function calling



Note

When using the Converse API, the topK and reasoningConfig parameters must be placed in additional Model Request Fields instead of inference Config.

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
          }
        },
```

Complete response structure 238

```
{
          "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:

Complete response structure 239

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	
AWS SDK for C++	AWS SDK for C++ code examples	
AWS CLI	AWS CLI code examples	
AWS SDK for Go	AWS SDK for Go code examples	
AWS SDK for Java	AWS SDK for Java code examples	
AWS SDK for JavaScript	AWS SDK for JavaScript code examples	
AWS SDK for Kotlin	AWS SDK for Kotlin code examples	
AWS SDK for .NET	AWS SDK for .NET code examples	
AWS SDK for PHP	AWS SDK for PHP code examples	
AWS Tools for PowerShell	AWS Tools for PowerShell code examples	
AWS SDK for Python (Boto3)	AWS SDK for Python (Boto3) code examples	
AWS SDK for Ruby	AWS SDK for Ruby code examples	
AWS SDK for Rust	AWS SDK for Rust code examples	
AWS SDK for SAP ABAP	AWS SDK for SAP ABAP code examples	
AWS SDK for Swift	AWS SDK for Swift code examples	

SDK reference 241

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.

Key benefits 242

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 <u>AWS Acceptable Use Policy</u> and mitigate undesired outcomes while providing a delightful customer experience. Our approach to Responsible AI (RAI) is structured around our <u>core dimensions of responsible AI</u>, 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 Al system

Topics

- Guidelines
- Recommendations
- Amazon Nova Lite and Pro Customizable Content Moderation Settings

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.

Guidelines 244

- 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

Recommendations 245

Bedrock models, including Amazon Nova. See Section <u>50.3</u> of the AWS Service Terms and the AWS <u>Data Privacy FAQ</u> for more information. For service-specific privacy information, see the Privacy and Security section of the <u>Amazon Bedrock FAQs</u> 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 <u>here</u>.

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 <u>Service Terms</u>). 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.

Amazon Nova Lite and Pro Customizable Content Moderation Settings

Content generation for Amazon Nova models is moderated by multiple <u>responsible AI (RAI)</u> <u>controls</u>. Two key controls are the alignment of the core model to RAI pillars and a runtime control – called an output model – that moderates core model responses to ensure alignment with these pillars.

Amazon Nova Lite and Pro support customizable content moderation settings for customers with approved business use cases. These settings allow content generation through three available combinations: the security pillar only, a combined setting for safety, sensitive content and fairness, or all pillars together.

These pillars encompass:

- Safety Covering dangerous activities, weapons and controlled substances.
- **Sensitive content** Including profanity, nudity and bullying.
- Fairness Considerations around bias and culture.
- **Security** Concerns involving malware and malicious content.

Amazon Nova customizable content moderation settings allow you to adjust safeguards relevant to your business requirements. In all cases, Amazon Nova enforces essential, non-configurable controls to ensure responsible use of AI, such as controls to prevent harm to children and preserve privacy. Please see Responsible use for additional details on Amazon Nova safeguards.

Nova customizable content moderation settings are available for the Lite and Pro models using the method in <u>Deploy a custom model for on-demand inference</u> in the Bedrock User Guide, in the useast-1 (N. Virginia) region.

To access customizable content moderation settings, contact your AWS Account Manager.

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 guotas in the AWS General Reference.

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 <u>shared responsibility model</u> 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 <u>AWS</u>
 <u>Compliance Programs</u>. 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 Bedrock resources with AWS CloudFormation

Because Amazon Bedrock 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 <u>Amazon Bedrock</u> <u>agents</u> or <u>Amazon Bedrock knowledge bases</u>, see the <u>Amazon Bedrock resource type reference</u> in the <u>AWS CloudFormation User Guide</u>.

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
Initial release of Amazon Nova 2.0	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