How to Use Amazon Bedrock to Deploy Foundation Models in Production (Complete Guide)

22 April 2026 - 7 min. read

Amazon Bedrock | Foundation Models | Generative AI | Large Language Models | Machine Learning

Introduction

Generative AI, Machine Learning, Large Language Models, Foundation Models.

If you work in the Cloud, these terms are buzzing everywhere: from tech conferences to corporate meetings, and in every newsletter landing in your inbox. But how many times, after the initial enthusiasm, have you faced the most uncomfortable question: "Okay, but how do we bring this to production?"

That is exactly where many generative AI projects stall. Between choosing models, configuring infrastructure, optimizing costs, and meeting security constraints, the gap between the proof-of-concept and production can change significantly and become a blocker to the project's go-live.

Amazon Bedrock was created precisely to solve these problems. It is not just another service promising miracles, but a serverless platform that makes the most advanced Foundation Models (FM) accessible through managed APIs, allowing you to focus on business value rather than infrastructure. In this guide, we will show you how to use Amazon Bedrock in real-world scenarios, from deployment to cost optimization, covering architectural patterns like RAG and Agents. Because doing a "hello world"

with ChatGPT is one thing, but building scalable and efficient production-ready systems is another.

What is Amazon Bedrock and why should you consider it?

Amazon Bedrock is a fully managed service that offers access to Foundation Models from several leading AI companies through a single API. Think of it as an AI model marketplace where you can choose the one best suited for your use-case without worrying about training, hosting, or scaling.

The available Foundation Models

Bedrock provides proprietary and open-source models, including:

- **Anthropic** (Claude Opus, Sonnet, Haiku)
- **Meta** (Llama)
- **Amazon** (Nova Lite, Nova Pro, Titan Embeddings)
- **AI21 Labs** (Jurassic-2)
- **DeepSeek**
- **MoonShot Kimi**

The variety is not random: every model has specific characteristics in terms of performance, cost, capabilities, and speed. Choosing the right one makes the difference between a sustainable project and an AWS bill spiraling out of control.

Why Bedrock instead of other solutions?

"But can't I just use OpenAI's APIs directly or host an open-source model on EC2?" we are often asked. Sure you can. But it is important to consider these aspects:

- **Native AWS integration:** Bedrock integrates perfectly with the rest of the AWS ecosystem. IAM policies, VPC endpoints, CloudWatch metrics, AWS PrivateLink – everything works out of the box. If your infrastructure is already on AWS, the integration overhead is minimal.
- **Simplified management:** No servers to maintain, no models to update manually, no worries about availability zones or failover. It is serverless in the true sense of the word.

- **Data residency and compliance:** Your data does not leave your AWS account. For regulated sectors like finance or healthcare, this can be a non-negotiable requirement.
- **Transparent pricing:** Pay-per-use based on processed tokens. No initial commitments, no hidden infrastructure costs.

But be careful: as always in the cloud, "serverless" does not mean it does everything by itself; it is crucial to architect intelligently to optimize costs and performance.

Basic Architecture: How Bedrock works

Let's understand how Bedrock works under the hood.

The Invocation Model

Bedrock offers mainly two usage modes:

- **On-demand inference:** You make an API call, you receive a response. Simple as that. You pay for what you use, ideal for variable loads.
- **Provisioned throughput:** You reserve dedicated capacity to have predictable latency and optimized costs on high volumes. Think of it as Reserved Instances, but for AI.

The choice between the two depends, as usual, on your use-case. If, for example, you have a chatbot with unpredictable spikes, on-demand is the right choice. If you process thousands of documents per day with a constant volume, provisioned throughput can save you up to 50%.

Key Architectural Components

A typical Bedrock implementation includes:

- **Model invocation:** Direct calls to the Foundation Model.
- **Knowledge Bases:** Vector repositories to implement RAG.
- **Agents:** Autonomous orchestrators that can use tools and external APIs.
- **Guardrails:** Policies to filter problematic content or PII.

Custom models: Fine-tuning base models (when on-demand is not enough).

Deploy

Theory is done. Now let's see how to bring Bedrock to production using Infrastructure as Code.

AWS Prerequisites

Before starting, ensure you have:

- An AWS account with sufficient permissions.
- AWS CLI configured.
- Enabled the desired models in the Bedrock console (some require an explicit request).

Important: Not all models are available in all regions. Always check regional availability before designing the architecture.

Once deployed, we can use our foundational model for our workloads; a typical example is RAG.

Integrating Bedrock with an existing application

You might be dealing with an existing application that doesn't use native Amazon Bedrock APIs. It's not a problem at all: during re:Invent 2025, **Project Mantle** was announced. It is a distributed inference engine designed to offer OpenAI-compatible API endpoints. The idea is extremely simple: it acts as a drop-in replacement, allowing developers to migrate existing applications (built using the OpenAI API set) to Bedrock, simply by changing the API endpoint and generating a new key from the AWS console. This way, you can use the models present on Bedrock without modifying the application code, reducing porting time for applications to zero.

In this page, you can find everything you need to start.

Costs and optimization: making the numbers add up

Let's talk about money. Because a chatbot that costs €10,000 a month to serve 1,000 users is not sustainable.

Bedrock Pricing Model

Bedrock uses a pay-per-use model based on:

- **Input tokens:** Text you send to the model (prompt + context).

- **Output tokens:** Text generated by the model.
- **Storage:** (for Knowledge Bases): Cost of OpenSearch Serverless or other vector DB.
- **Provisioned throughput:** (optional): Reserved capacity.

Pricing varies from model to model, here an example as of January 2026

- **Claude Haiku**: $1/1M input tokens, $5/1M output tokens
- **Claude Sonnet**: $3/1M input tokens, $15/1M output tokens
- **Claude Opus**: $5/1M input tokens, $25/1M output tokens

Cost optimization strategies

1. **Intelligent model selection** Don't always use the largest model. Create a tiered strategy:

```
`# model_selector.py
def select_model_for_task(task_type, complexity, context_length):
""" Select the model related to tasks """
if task_type == 'classification' or task_type == 'extraction':
    return 'anthropic.claude-4-haiku-v1:0'
elif task_type == 'summarization':
    if context_length < 10000:
        return 'anthropic.claude-4-v1:0'
    else:
        return 'anthropic.claude-4-v1:0'

elif task_type == 'reasoning' or complexity == 'high':
    if context_length > 50000:
        return 'anthropic.claude-4-opus-v1:0'
    else:
        return 'anthropic.claude-4-sonnet-v1:0'

elif task_type == 'code_generation':
    return 'anthropic.claude-4-sonnet-v1:0'
```

```
else:
    return 'anthropic.claude-4-haiku-v1:0'
```

2. Monitoring and cost alerts Implement monitoring from day one:

```
# cost_monitoring.py
import boto3
from datetime import datetime, timedelta

cloudwatch = boto3.client('cloudwatch')


def put_cost_metric(model_id, tokens_used, cost):
    """
    Publishes cost metrics to CloudWatch
    """
    cloudwatch.put_metric_data(
        Namespace='Bedrock/Usage',
        MetricData=[
            {
                'MetricName': 'TokensUsed',
                'Value': tokens_used,
                'Unit': 'Count',
                'Timestamp': datetime.utcnow(),
                'Dimensions': [
                    {'Name': 'ModelId', 'Value': model_id}
                ]
            },
            {
                'MetricName': 'EstimatedCost',
                'Value': cost,
                'Unit': 'None',
                'Timestamp': datetime.utcnow(),
                'Dimensions': [
                    {'Name': 'ModelId', 'Value': model_id}
                ]
            }
```

```
        ]
    )
```

At this point, with this metric, you can create a CloudWatch alarm on token usage to detect anomalies and avoid unexpected costs.

Security and Governance: protecting data

Bedrock handles potentially sensitive information. Security is not optional.

Guardrails: the automatic filter

Bedrock Guardrails allows you to automatically filter problematic content, both in user inputs and model responses, regardless of the model used.

It works across six policy categories: content filters, denied topics, word filters, sensitive information filters, contextual grounding check, and Automated Reasoning checks. Each one is independently configurable, so you can build exactly the level of protection you need.

Content filters cover six predefined categories of harmful content: Hate, Insults, Sexual, Violence, Misconduct, and Prompt Attack. For each one you can adjust the filter strength, and it's not a binary choice: you can tune it based on the application's context.

For sensitive data, you can choose from a predefined list of PII types or define custom patterns using regular expressions. Particularly useful in regulated industries like finance or healthcare.

Conclusion: from MVP to production

We have covered a lot of ground: deployment infrastructure-as-code, RAG for knowledge retrieval, Agents for task automation, cost optimization, security best practices.

But here is the truth: **bringing Foundation Models to production is not a technological project, it is a business project.** Technology is the means, not the end.

Before writing the first line of code, ask yourselves:

- What is the measurable business value? "Having a chatbot" is not an answer.

- Which success metric will we use? Cost per interaction? Customer satisfaction? Time-to-resolution?
- Do we have the right data? A RAG without good documentation is as useless as a car without gas.
- Are costs sustainable at scale? €100/month for a POC is acceptable. €10,000/month for 1000 users probably isn't.

Amazon Bedrock makes the technology accessible. But building successful AI systems still requires planning, intelligent architecture, and continuous optimization. The good news? Now you have the tools to do it.

In the next articles, we will delve into specific use cases: how to build a multi-language RAG system, how to optimize an Agent to reduce hallucinations, how to implement A/B testing between different models.

Stay tuned on **#Proud2beCloud!**

Damiano Giorgi

Ex on-prem systems engineer, lazy and prone to automating boring tasks. In constant search of technological innovations and new exciting things to experience. And that's why I love Cloud Computing! At this moment, the only "hardware" I regularly dedicate myself to is that my bass; if you can't find me in the office or in the band room try at the pub or at some airport, then!
