You're right to ask about the Converse API! It's a more modern and flexible way to interact with Bedrock models, especially for multi-turn conversations, and it's designed to provide a more consistent interface across different model providers.

Here's an example of how you can use the AWS SDK for Rust with the Bedrock Converse API. This will feel more like a chat interaction.

**Key Differences from InvokeModel:**

* **Message-based:** You send a list of messages (user, assistant turns) rather than a single monolithic prompt string.
* **Content Blocks:** Messages contain ContentBlocks, which can be text, images, tool requests, or tool results. For your Q&A with documents, you'll primarily use text content blocks.
* **System Prompts:** Can often be provided as a separate parameter, which is cleaner.
* **Inference Configuration:** Parameters like temperature and max tokens are grouped in an InferenceConfiguration object.
* **Structured Output:** The response is also structured, typically giving you the assistant's reply as a Message object.

**Step 1: Add/Verify Dependencies in your Cargo.toml**

These are likely the same as for InvokeModel. Ensure you have up-to-date versions.

Ini, TOML

[dependencies]  
aws-config = { version = "1.1.9", features = ["behavior-version-latest"] } # Or latest  
aws-sdk-bedrockruntime = "1.19.0" # Or latest - check crates.io for the absolute latest  
tokio = { version = "1", features = ["full"] }  
serde = { version = "1.0", features = ["derive"] } // May not be strictly needed for basic text I/O with Converse but good to have  
serde\_json = "1.0" // Ditto  
# For this example, serde and serde\_json are not directly used for request/response  
# as the SDK provides typed structs for Converse API.

**Step 2: Rust Code Example (main.rs) using Converse API**

Rust

use aws\_config::meta::region::RegionProviderChain;  
use aws\_sdk\_bedrockruntime::{Client, Error};  
use aws\_sdk\_bedrockruntime::types::{  
 ContentBlock, Message, ConversationRole, SystemContentBlock, InferenceConfiguration,  
};  
  
// Note: For Converse API, explicit request/response body structs like in InvokeModel  
// are often not needed if you're just dealing with text, as the SDK provides  
// Message and ContentBlock types. If your model expects very specific  
// 'additionalModelRequestFields' for Titan, you might need serde\_json for that part.  
  
#[tokio::main]  
async fn main() -> Result<(), Box<dyn std::error::Error>> { // Using Box<dyn Error> for broader error handling  
 // 1. Configure AWS Region  
 let region\_provider = RegionProviderChain::default\_provider().or\_else("us-east-1"); // Specify your Bedrock region  
 let config = aws\_config::load\_defaults(aws\_config::BehaviorVersion::latest()) // More modern way to load config  
 .region(region\_provider)  
 .await;  
 let client = Client::new(&config);  
  
 // 2. Define your model ID, system prompt, document, and question  
 let model\_id = "amazon.your-new-titan-model-id"; // <-- REPLACE with your specific Titan model ID that supports Converse API  
  
 let system\_prompt\_text = r#"You are an AI analytical assistant with expertise in document comprehension and insight generation. You are processing one or more large documents provided by the user. Your task is to answer questions based \*solely\* on the content of these documents, presenting your findings in a conversational, natural language format.  
  
When responding, adopt a thoughtful and analytical perspective, as if you are sharing an informed opinion or interpretation derived from the text. Avoid imperative language or direct advice. Instead, present your findings as observations or considerations suggested by the document's content.  
  
Key guidelines:  
1. Document Grounding: All information, observations, and "opinions" must be directly and exclusively traceable to the provided document(s).  
2. Conversational & Natural Language: Your response should read like a well-reasoned chat message.  
3. Analytical Tone: Frame your output as though you are offering an analysis or interpretation.  
4. Handling Ambiguity/Absence: If the information is ambiguous or not present, acknowledge this.  
5. Focus on Relevance: Ensure your response directly addresses the user's question.  
6. No Direct Advice: Share observations, not direct instructions or advice."#;  
  
 let document\_content = r#"  
 AGREEMENT OF SERVICES  
 This Agreement is made effective as of May 24, 2025, by and between Innovate Solutions Inc. ("Client"),  
 located at 123 Tech Park, Future City, and QuantumLeap AI Corp. ("Provider"), located at 789 AI Avenue, Metropolis.  
  
 1. Scope of Services: Provider will deliver advanced AI-driven data analysis services as detailed in Exhibit A.  
 The primary objective is to identify market expansion opportunities.  
  
 (Your full document content would go here... this is a snippet)  
  
 6. Governing Law: This Agreement shall be governed by and construed in accordance with the laws of the State of Innovation.  
 "#;  
  
 let user\_question = "What appears to be the main objective of the services provided by QuantumLeap AI Corp., and what is their maximum financial liability according to this document?";  
  
 // 3. Construct the user message content, including the document and the question  
 // For large documents, ensure the model's context window can handle the combined length.  
 let user\_message\_text = format!(  
 "Please analyze the following document and answer my question.\n\nDocument Content:\n{}\n\nMy Question: {}",  
 document\_content, user\_question  
 );  
  
 // 4. Prepare the messages for the Converse API  
 // For a single Q&A turn, this will typically be one user message.  
 // For multi-turn conversations, you'd append assistant and new user messages here.  
 let messages = vec![  
 Message::builder()  
 .role(ConversationRole::User)  
 .content(vec![ContentBlock::Text(user\_message\_text)])  
 .build()  
 .map\_err(|e| format!("Failed to build user message: {:?}", e))?, // Convert BuildError to String error  
 ];  
  
 // 5. Prepare the system prompt (if the model supports it this way via Converse API)  
 // Some models prefer the system prompt as the first message in the 'messages' array,  
 // others use the dedicated 'system' parameter. Check model docs.  
 let system\_prompts = vec![  
 SystemContentBlock::builder()  
 .text(system\_prompt\_text)  
 .build()  
 .map\_err(|e| format!("Failed to build system prompt: {:?}", e))?,  
 ];  
   
 // 6. Define Inference Configuration  
 let inference\_config = InferenceConfiguration::builder()  
 .max\_tokens(1024) // Max tokens in the generated response  
 .temperature(0.7)  
 .top\_p(0.9)  
 // .stop\_sequences(vec!["\n\nUser:".to\_string()]) // Optional  
 .build();  
  
 // 7. Call the Converse API  
 println!("Sending message to model {} via Converse API...", model\_id);  
 match client  
 .converse()  
 .model\_id(model\_id)  
 .set\_messages(Some(messages)) // Use set\_messages for Option<Vec<Message>>  
 .set\_system(Some(system\_prompts)) // Use set\_system for Option<Vec<SystemContentBlock>>  
 .inference\_config(inference\_config)  
 // You can also pass 'additional\_model\_request\_fields' as a Document if needed  
 // .additional\_model\_request\_fields(Document::from\_json\_str("{\"guardrailIdentifier\":\"abc\"}").unwrap())  
 .send()  
 .await  
 {  
 Ok(converse\_output) => {  
 println!("\nConverse API call successful.");  
  
 // Extract the assistant's response message  
 if let Some(output\_message) = converse\_output.output { // 'output' is Option<ConverseOutput>  
 if let Some(message) = output\_message.as\_message() { // 'as\_message()' returns Option<&Message>  
 if let Some(content\_blocks) = message.content() { // 'content()' returns Option<&[ContentBlock]>  
 for content\_block in content\_blocks {  
 if let ContentBlock::Text(text\_content) = content\_block {  
 println!("\nAI Assistant's Natural Language Response:");  
 println!("{}", text\_content.trim());  
 }  
 // You can add more handlers here for other ContentBlock types  
 // e.g., tool\_use, tool\_result, image  
 }  
 } else {  
 println!("\nAssistant's message had no content blocks.");  
 }  
 } else {  
 println!("\nConverse API output was not a message type: {:?}", output\_message);  
 }  
 } else {  
 println!("\nNo output message received from Converse API.");  
 }  
  
 // You can also inspect other parts of converse\_output like:  
 // converse\_output.stop\_reason  
 // converse\_output.usage (token counts)  
 // converse\_output.metrics  
 if let Some(usage) = converse\_output.usage {  
 println!("\nToken Usage: Input Tokens: {}, Output Tokens: {}",  
 usage.input\_tokens(),  
 usage.output\_tokens()  
 );  
 }  
 }  
 Err(e) => {  
 eprintln!("\nError calling Converse API: {}", e);  
 // The error 'e' will be of type SdkError<ConverseError>  
 // You can match on e.as\_service\_error() to get specific ConverseError types  
 }  
 }  
  
 Ok(())  
}

**Key Changes and Explanation for Converse API:**

1. **Dependencies:** aws-sdk-bedrockruntime is still the key crate. serde and serde\_json are less directly involved in constructing the main request/response objects if you're primarily dealing with text, as the SDK provides typed structs like Message, ContentBlock, and SystemContentBlock.
2. **Client:** Still aws\_sdk\_bedrockruntime::Client.
3. **converse() Method:** This is the primary method used instead of invoke\_model().
4. **Input Structure:**
   * .model\_id(): Same as before.
   * .messages(): This takes a Vec<Message>. Each Message has:
     + role(ConversationRole::User) or ConversationRole::Assistant).
     + content(vec![ContentBlock::Text(...)]): The actual content. For your use case, you'll put the combined document + question into a ContentBlock::Text for the user's message.
   * .system(): This takes a Vec<SystemContentBlock>. This is where you should put your system prompt.
   * .inference\_config(): This takes an InferenceConfiguration struct where you set max\_tokens, temperature, stop\_sequences, etc.
   * .additional\_model\_request\_fields(): If your specific Titan model requires parameters not covered by the standard InferenceConfiguration or if you need to pass specific things in the provider's native format, you can use this field. It takes an aws\_sdk\_bedrockruntime::types::Document (which can be created from a JSON string if needed).
5. **Output Structure (ConverseOutput):**
   * The main part you'll be interested in is converse\_output.output, which is an Option<aws\_sdk\_bedrockruntime::types::ConverseOutput>.
   * This output can be different types (message, metrics, trace). You typically check if it's a message using as\_message().
   * If it's a message (output\_message.as\_message()), it will be of type Message, and you can get its content() which is a slice of ContentBlocks.
   * You then iterate through the content blocks and extract the text if it's ContentBlock::Text(text\_content).
   * The ConverseOutput also includes stop\_reason(), usage() (for token counts), and metrics().

**Important Notes for Your New Titan Models:**

* **Model Support for Converse API:** Verify that your specific new Amazon Titan model fully supports the Converse API and how it expects system prompts and messages. Most modern models do.
* **System Prompt Placement:** The AWS SDK example for Converse shows the system parameter being used. This is generally the preferred way if the model supports it.
* **additional\_model\_request\_fields**: If your specific Titan model has unique inference parameters not exposed directly in InferenceConfiguration (e.g., specific guardrail configurations or other Titan-specific settings), you'd use additional\_model\_request\_fields. This field accepts an aws\_sdk\_bedrockruntime::types::Document. You'd typically construct this from a JSON string:  
  Rust  
  // Example if you needed to pass additional fields  
  // let additional\_fields\_json = r#"{"amazon-titan-specific-param": "value"}"#;  
  // let additional\_fields\_doc = aws\_sdk\_bedrockruntime::types::Document::from\_json\_str(additional\_fields\_json)  
  // .map\_err(|e| format!("Failed to create additional fields document: {}", e))?;  
  //  
  // ... in the converse call:  
  // .additional\_model\_request\_fields(additional\_fields\_doc)  
  **Consult the model's documentation for what, if anything, should go here.** For basic text generation and Q&A with standard inference parameters, you might not need it.
* **Error Handling:** The .map\_err(...) is a simple way to convert BuildError from the builders into a type that can be returned by main. Production code would use more specific error types or libraries like anyhow or thiserror. SdkError<ConverseError> provides more detailed error information from AWS.

This example should get you started with the Converse API in Rust. It's designed to be more robust for chat-like interactions and managing conversation history (though this example only shows a single turn). Remember to thoroughly check the documentation for your specific Titan model on Bedrock for any nuances in its Converse API implementation.