Holistic AI

Bedrock Proxy API

Hackathon Participant Guide

🚀 Quick Start

API Endpoint

```
https://ctwa92wg1b.execute-api.us-east-1.amazonaws.com/prod/invoke
```

Authentication

Each team will receive:

- **Team ID**: Your team identifier
- **API Token**: Your API key

Basic Request Format

```
curl -X POST https://ctwa92wg1b.execute-api.us-east-1.amazonaws.com/prod/invoke \
  -H "Content-Type: application/json" \
  -H "X-Team-ID: YOUR_TEAM_ID" \
  -H "X-API-Token: YOUR_API_TOKEN" \
  -d '{
    "team_id": "YOUR_TEAM_ID",
    "model": "us.anthropic.claude-3-5-sonnet-20241022-v2:0",
    "messages": [
      {"role": "user", "content": "Hello, how are you?"}
    ],
    "max_tokens": 1024
  }'
```

Available Models

Anthropic Claude Series

`us.anthropic.claude-3-5-sonnet-20241022-v2:0` Recommended

`us.anthropic.claude-3-5-haiku-20241022-v1:0` Fast

`us.anthropic.claude-3-opus-20240229-v1:0` Most Powerful

`us.anthropic.claude-3-sonnet-20240229-v1:0`

`us.anthropic.claude-3-haiku-20240307-v1:0` Fastest

`us.anthropic.claude-opus-4-20250514-v1:0`

`us.anthropic.claude-sonnet-4-20250514-v1:0`

`us.anthropic.claude-sonnet-4-5-20250929-v1:0`

`us.anthropic.claude-haiku-4-5-20251001-v1:0`

Meta Llama Series

`us.meta.llama3-2-90b-instruct-v1:0` Large

`us.meta.llama3-2-11b-instruct-v1:0` Balanced

`us.meta.llama3-2-3b-instruct-v1:0` Lightweight

`us.meta.llama3-2-1b-instruct-v1:0` Ultra-light

`us.meta.llama3-1-70b-instruct-v1:0`

`us.meta.llama3-1-8b-instruct-v1:0`

`us.meta.llama3-3-70b-instruct-v1:0`

`us.meta.llama4-scout-17b-instruct-v1:0`

`us.meta.llama4-maverick-17b-instruct-v1:0`

Amazon Nova Series

`us.amazon.nova-premier-v1:0` Most Powerful

`us.amazon.nova-pro-v1:0` Recommended

`us.amazon.nova-lite-v1:0` Fast

`us.amazon.nova-micro-v1:0` Ultra-fast

Mistral Series

`us.mistral.pixtral-large-2502-v1:0` Large

`mistral.mistral-large-2402-v1:0`

`mistral.mistral-small-2402-v1:0` Fast

`mistral.mistral-7b-instruct-v0:2`

`mistral.mixtral-8x7b-instruct-v0:1`

DeepSeek Series

`us.deepseek.r1-v1:0` Latest

<> Usage Examples

1. Basic Conversation

```
import requests

API_ENDPOINT = "https://ctwa92wg1b.execute-api.us-east-1.amazonaws.com/prod/invoke"
TEAM_ID = "your_team_id"
API_TOKEN = "your_api_token"

response = requests.post(
    API_ENDPOINT,
    headers={{
        "Content-Type": "application/json",
        "X-Team-ID": TEAM_ID,
        "X-API-Token": API_TOKEN
    }},
    json={{
```

```
        "team_id": TEAM_ID,
        "model": "us.anthropic.claude-3-5-sonnet-20241022-v2:0",
        "messages": [
            {{"role": "user", "content": "Hello, how are you?"}}
        ],
        "max_tokens": 1024
    }}
)

result = response.json()
print(result["content"][0]["text"])
```

2. Multi-turn Conversation

```
conversation = [
    {{"role": "user", "content": "My name is Alice"}},
    {{"role": "assistant", "content": "Nice to meet you, Alice!"}},
    {{"role": "user", "content": "What's my name?"}}
]

response = requests.post(
    API_ENDPOINT,
    headers={{
        "Content-Type": "application/json",
        "X-Team-ID": TEAM_ID,
        "X-API-Token": API_TOKEN
    }},
    json={{
        "team_id": TEAM_ID,
        "model": "us.anthropic.claude-3-5-sonnet-20241022-v2:0",
        "messages": conversation,
        "max_tokens": 1024
    }}
)
```

3. Agent Functionality (Tool Calling)

> **Note:** Tool calling is natively supported by Claude series, DeepSeek, and Amazon Nova models. Llama and Mistral models use prompt format and do not support these parameters at the API level, though similar functionality may be achievable through prompt engineering.

```
tools = [
    {{
        "name": "get_weather",
        "description": "Get the current weather for a location",
        "input_schema": {{
            "type": "object",
            "properties": {{
                "location": {{
                    "type": "string",
                    "description": "The city and state"
                }}
            }},
            "required": ["location"]
        }}
    }}
]

response = requests.post(
    API_ENDPOINT,
    headers={{
        "Content-Type": "application/json",
        "X-Team-ID": TEAM_ID,
        "X-API-Token": API_TOKEN
    }},
    json={{
        "team_id": TEAM_ID,
        "model": "us.anthropic.claude-3-5-sonnet-20241022-v2:0",
        "messages": [
            {{"role": "user", "content": "What's the weather in San Francisco?"}}
        ],
        "tools": tools,
        "tool_choice": "auto",
        "max_tokens": 1024
    }}
)
```

4. Structured Output (JSON Schema)

> **Note:** Structured output uses Tool Use (Function Calling) under the hood. Not all models support this feature. If you receive an error indicating that the model doesn't support Tool Use, you may need to use a different model or alternative approaches (see below).

Get structured JSON responses that match your schema:

```
response = requests.post(
    API_ENDPOINT,
    headers={{
        "Content-Type": "application/json",
        "X-Team-ID": TEAM_ID,
        "X-API-Token": API_TOKEN
    }},
    json={{
        "team_id": TEAM_ID,
        "model": "us.anthropic.claude-3-5-sonnet-20241022-v2:0",
        "messages": [
            {{"role": "user", "content": "Extract person info: John is 30 years old and
        ],
        "max_tokens": 1024,
        "response_format": {{
            "type": "json_schema",
            "json_schema": {{
                "name": "person_info",
                "strict": True,
                "schema": {{
                    "type": "object",
                    "properties": {{
                        "name": {{"type": "string"}},
                        "age": {{"type": "integer"}},
                        "city": {{"type": "string"}}
                    }},
                    "required": ["name", "age", "city"]
                }}
            }}
        }}
    }}
)

# Parse structured response
result = response.json()
person_json = json.loads(result["content"][0]["text"])
print(person_json)  # {{"name": "John", "age": 30, "city": "New York"}}
```

> **Note:** If a model doesn't support structured output, you'll receive an error message. In that case, you can:
>
> - **Try a different model** - Some models have better support for Tool Use
> - **Use Prompt Engineering** - Explicitly request JSON format in your prompt
> - **Client-side Parsing** - Let the model return natural JSON and parse/validate it in your application code

5. Using Pydantic Models (Python)

If you're using Python with Pydantic, you can easily convert your models to JSON Schema:

```
from pydantic import BaseModel, Field

class Person(BaseModel):
    """Person information"""
    name: str = Field(description="Person's full name")
    age: int = Field(ge=0, le=150, description="Person's age")
    city: str = Field(description="City where person lives")

# Get JSON schema from Pydantic model
json_schema = Person.model_json_schema()

# Use in API request
response = requests.post(
    API_ENDPOINT,
    headers={{
        "Content-Type": "application/json",
        "X-Team-ID": TEAM_ID,
        "X-API-Token": API_TOKEN
    }},
    json={{
        "team_id": TEAM_ID,
        "model": "us.anthropic.claude-3-5-sonnet-20241022-v2:0",
        "messages": [
            {{"role": "user", "content": "Extract person info: John is 30 and lives in
        ],
        "max_tokens": 1024,
        "response_format": {{
            "type": "json_schema",
            "json_schema": {{
                "name": "person_info",
                "strict": True,
                "schema": json_schema
            }}
        }}
    }}
)

# Parse and validate with Pydantic
result = response.json()
person_json = json.loads(result["content"][0]["text"])
person = Person.model_validate(person_json)  # Automatic validation
print(f"Name: {{person.name}}, Age: {{person.age}}, City: {{person.city}}")
```

> **Tip:** The API also supports directly passing Pydantic model's JSON schema (without the `type` field), and it will automatically wrap it in the correct format.

$ Quota and Limits

> **Budget Limit:** Each team has an independent budget limit (default $50). Request count, token count, and max tokens per request are unlimited as long as you stay within budget.

Check Remaining Quota

The API response includes a `metadata.remaining_quota` field:

```
{{
  "content": [...],
  "metadata": {{
    "remaining_quota": {{
      "llm_cost": 10.5,
      "gpu_cost": 5.2,
      "total_cost": 15.7,
      "budget_limit": 50.0,
      "remaining_budget": 34.3,
      "budget_usage_percent": 31.4
    }}
  }}
}}
```

GPU Access

Teams can request GPU access through AWS SageMaker Notebook Instances for training models, running inference, or other compute-intensive tasks. GPU access is managed by hackathon administrators.

> **Request GPU Access:** Contact hackathon organizers to request a GPU notebook instance. Administrators will create and manage instances for your team.

Available GPU Instance Types

Instance Type	GPU	VRAM	CPU	RAM	Cost/Hour	Best For
`ml.t3.medium`	None	-	2	4GB	$0.05	Testing, no GPU needed
`ml.t3.xlarge`	None	-	4	16GB	$0.20	CPU-only workloads
`ml.g4dn.xlarge`	1x T4	16GB	4	16GB	$0.74	Inference, light training
`ml.g5.xlarge`	1x A10G	24GB	4	16GB	$1.41	Best price/performance
`ml.p3.2xlarge`	1x V100	16GB	8	61GB	$3.83	High-end training

Using GPU Notebooks

Once a notebook instance is created for your team, you will receive a Jupyter notebook URL. The notebook includes:

- Python 3.11 environment
- Pre-installed ML libraries
- AWS SDK (boto3) for Bedrock access
- Full internet access for package installation

Cost Management

GPU costs are tracked separately from LLM API costs and count toward your team's total budget. Important notes:

- **Stop when not in use:** Instances incur costs while running. Stop them when you're done working.
- **Monitor usage:** Check your budget regularly via the API's `remaining_quota` field.
- **Start with smaller instances:** Use `ml.t3.medium` for testing, upgrade to GPU instances only when needed.

Accessing Bedrock from Notebooks

Notebook instances have direct access to AWS Bedrock. You can use boto3 to call Bedrock models:

```
import boto3
import json

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

response = bedrock_runtime.invoke_model(
    modelId='us.anthropic.claude-3-5-sonnet-20241022-v2:0',
    body=json.dumps({{
        'anthropic_version': 'bedrock-2023-05-31',
        'max_tokens': 1024,
        'messages': [{{
            'role': 'user',
            'content': 'Hello!'
        }}]
    }})
)

result = json.loads(response['body'].read())
print(result)
```

> **Note:** GPU access is subject to availability and budget limits. Contact organizers early if you need GPU resources for your project.

Error Handling

Error Code	Description	Solution
401	Unauthorized	Check team_id and api_token
403	Model not allowed	Verify model ID
429	Too Many Requests	Budget exhausted, check remaining_quota
400	Bad Request	Check request format

Best Practices

1. **Choose the Right Model** - Use Haiku for simple tasks, Sonnet/Opus for complex tasks, Claude series for agent applications
2. **Manage Conversation History** - Keep it concise and clean up irrelevant history
3. **Optimize Token Usage** - Use max_tokens to limit output length
4. **Error Handling** - Always check response status codes and implement retry logic
5. **Monitor Budget** - Regularly check remaining_quota and alert users when approaching limit

---

Last Updated: 2025-11-10 | API Version: v1

If you have questions, please contact Hackathon organizers