Contents

ADR-008: Shadow Self on SageMaker ml.g5.2xlarge

Status

Accepted

Context

The Shadow Self is Cato's introspective verification mechanism. It uses a separate LLM (Llama-3-8B) to probe and verify the main model's responses by:

1. **Hidden state extraction**: Analyzing activation patterns for uncertainty detection
2. **Activation probing**: Training linear classifiers on hidden states to detect specific properties
3. **Consistency checking**: Comparing response patterns across different prompts

Why Not Bedrock?

AWS Bedrock provides managed LLM inference but: - **No hidden state access**: Bedrock APIs only return generated text - **No activation extraction**: Cannot get intermediate layer outputs - **Black box**: No visibility into model internals

For Shadow Self to function, we need **full access to model internals**.

Infrastructure Requirements

Requirement	Specification
Model	Llama-3-8B-Instruct (16GB weights)

Requirement	Specification
VRAM	24GB minimum (for FP16 + activations)
Hidden states	Last 8 layers extractable
Throughput	100-500 requests/second at scale
Latency	$<$ 500ms p99

Decision

Deploy Shadow Self on **SageMaker Real-Time Inference** with custom inference container:

Instance Selection

ml.g5.2xlarge chosen for: - 24GB NVIDIA A10G VRAM (fits Llama-3-8B + headroom) - 8 vCPUs, 32GB RAM - ~$1.52/hour ($1,095/month per instance) - Good balance of cost and performance

Scaling Strategy

Users	QPS	Instances	Monthly Cost
100K	10	5	$5,500
1M	100	50	$55,000
10M	500	250	$275,000

With 3-year Savings Plans: **36% discount** → ~$175,000/month at 10M users

Architecture

```
                    Shadow Self Endpoint



    SageMaker Real-Time Inference

    Instance: ml.g5.2xlarge (24GB A10G)
    Container: Custom vLLM + hidden state extraction
    Model: meta-llama/Meta-Llama-3-8B-Instruct

    Auto-scaling: 5-300 instances
    Target: 80% GPU utilization



    Inference Container Features

       output_hidden_states=True
```

```
        Configurable layer extraction [-1, -4, -8]
        Mean pooling over sequence
        Last token extraction
        Activation probing classifier heads
```

Custom Inference Container

Dockerfile

```dockerfile
FROM nvidia/cuda:12.1-runtime-ubuntu22.04

# Install Python and dependencies
RUN apt-get update && apt-get install -y \
    python3.10 \
    python3-pip \
    && rm -rf /var/lib/apt/lists/*

# Install PyTorch and vLLM
RUN pip3 install --no-cache-dir \
    torch==2.1.0 \
    transformers==4.36.0 \
    vllm==0.2.7 \
    accelerate==0.25.0 \
    safetensors==0.4.1

# Copy inference code
COPY inference.py /opt/ml/code/inference.py
COPY model_handler.py /opt/ml/code/model_handler.py

WORKDIR /opt/ml/code

# Set environment variables
ENV MODEL_PATH=/opt/ml/model
ENV CUDA_VISIBLE_DEVICES=0

# Expose port for SageMaker
EXPOSE 8080

CMD ["python3", "model_handler.py"]
```

Inference Code

```python
# inference.py - Shadow Self with hidden state extraction

import torch
from transformers import AutoModelForCausalLM, AutoTokenizer
```

```python
from typing import Dict, Any, List, Optional
from dataclasses import dataclass
import numpy as np
import json

@dataclass
class HiddenStateResult:
    """Result with hidden states for activation probing."""
    generated_text: str
    hidden_states: Dict[str, Dict[str, List[float]]]
    logits_entropy: float
    generation_probs: List[float]

class ShadowSelfModel:
    """
    Llama-3-8B with hidden state extraction for Shadow Self verification.

    Extracts:
    - Hidden states from configurable layers
    - Logit entropy for uncertainty estimation
    - Per-token generation probabilities
    """

    def __init__(self, model_path: str = "/opt/ml/model"):
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

        # Load tokenizer
        self.tokenizer = AutoTokenizer.from_pretrained(model_path)
        if self.tokenizer.pad_token is None:
            self.tokenizer.pad_token = self.tokenizer.eos_token

        # Load model with hidden state output
        self.model = AutoModelForCausalLM.from_pretrained(
            model_path,
            torch_dtype=torch.float16,
            device_map="auto",
            output_hidden_states=True,
            output_attentions=False  # Skip attention for efficiency
        )
        self.model.eval()

    @torch.no_grad()
    def generate_with_hidden_states(
        self,
        text: str,
        target_layers: List[int] = [-1, -4, -8],
        max_new_tokens: int = 256,
        temperature: float = 0.7,
```

```
    return_probs: bool = True
) -> HiddenStateResult:
    """
    Generate text and extract hidden states.

    Args:
        text: Input prompt
        target_layers: Which layers to extract (negative = from end)
        max_new_tokens: Maximum generation length
        temperature: Sampling temperature
        return_probs: Whether to return token probabilities

    Returns:
        HiddenStateResult with text, hidden states, and metadata
    """
    # Tokenize input
    inputs = self.tokenizer(
        text,
        return_tensors="pt",
        padding=True,
        truncation=True,
        max_length=2048
    ).to(self.device)

    input_length = inputs.input_ids.shape[1]

    # Generate with hidden states
    outputs = self.model.generate(
        **inputs,
        max_new_tokens=max_new_tokens,
        temperature=temperature,
        do_sample=temperature > 0,
        output_hidden_states=True,
        output_scores=return_probs,
        return_dict_in_generate=True
    )

    # Decode generated text
    generated_ids = outputs.sequences[0][input_length:]
    generated_text = self.tokenizer.decode(
        generated_ids,
        skip_special_tokens=True
    )

    # Extract hidden states
    hidden_states = {}
    if hasattr(outputs, 'hidden_states') and outputs.hidden_states:
        # outputs.hidden_states is a tuple of (num_tokens, num_layers, batch, seq, hidden)
```

```
            for layer_idx in target_layers:
                layer_key = f"layer_{layer_idx}"

                # Get hidden state for this layer at first generation step
                if len(outputs.hidden_states) > 0:
                    first_step_hidden = outputs.hidden_states[0]
                    if abs(layer_idx) <= len(first_step_hidden):
                        layer_hidden = first_step_hidden[layer_idx]

                        hidden_states[layer_key] = {
                            "mean": layer_hidden.mean(dim=1).squeeze().cpu().tolist(),
                            "last_token": layer_hidden[:, -1, :].squeeze().cpu().tolist(),
                            "norm": layer_hidden.norm(dim=-1).mean().item()
                        }

        # Calculate logit entropy
        logits_entropy = 0.0
        generation_probs = []
        if hasattr(outputs, 'scores') and outputs.scores:
            for step_logits in outputs.scores:
                probs = torch.softmax(step_logits, dim=-1)
                entropy = -(probs * torch.log(probs + 1e-10)).sum(dim=-1).mean().item()
                logits_entropy += entropy

                # Get probability of generated token
                if len(generation_probs) < len(generated_ids):
                    token_idx = generated_ids[len(generation_probs)].item()
                    token_prob = probs[0, token_idx].item()
                    generation_probs.append(token_prob)

            logits_entropy /= len(outputs.scores)

        return HiddenStateResult(
            generated_text=generated_text,
            hidden_states=hidden_states,
            logits_entropy=logits_entropy,
            generation_probs=generation_probs
        )

    def probe_uncertainty(
        self,
        hidden_states: Dict[str, Dict[str, List[float]]],
        probe_weights: Optional[np.ndarray] = None
    ) -> float:
        """
        Use trained probe to estimate uncertainty from hidden states.

        Args:
```

```
            hidden_states: Extracted hidden states
            probe_weights: Trained linear probe weights

        Returns:
            Uncertainty score [0, 1]
        """
        if probe_weights is None:
            # Default: use hidden state norm as proxy
            norms = [
                hs.get("norm", 0.0)
                for hs in hidden_states.values()
            ]
            # Lower norms often correlate with uncertainty
            avg_norm = np.mean(norms) if norms else 0.0
            return 1.0 / (1.0 + avg_norm)  # Sigmoid-like transform

        # Use trained probe
        layer_key = list(hidden_states.keys())[0]
        features = np.array(hidden_states[layer_key]["mean"])
        uncertainty = float(np.dot(features, probe_weights))
        return max(0.0, min(1.0, uncertainty))


# SageMaker handler
def model_fn(model_dir: str):
    """Load model for SageMaker."""
    return ShadowSelfModel(model_dir)

def input_fn(request_body: str, request_content_type: str):
    """Parse input for SageMaker."""
    if request_content_type == "application/json":
        return json.loads(request_body)
    raise ValueError(f"Unsupported content type: {request_content_type}")

def predict_fn(data: Dict[str, Any], model: ShadowSelfModel):
    """Run prediction for SageMaker."""
    text = data.get("inputs", "")
    params = data.get("parameters", {})

    result = model.generate_with_hidden_states(
        text=text,
        target_layers=params.get("target_layers", [-1, -4, -8]),
        max_new_tokens=params.get("max_new_tokens", 256),
        temperature=params.get("temperature", 0.7),
        return_probs=params.get("return_probs", True)
    )

    return {
```

```
        "generated_text": result.generated_text,
        "hidden_states": result.hidden_states,
        "logits_entropy": result.logits_entropy,
        "generation_probs": result.generation_probs
    }

def output_fn(prediction: Dict[str, Any], accept: str):
    """Format output for SageMaker."""
    return json.dumps(prediction)
```

TypeScript Client

```
import {
  SageMakerRuntimeClient,
  InvokeEndpointCommand
} from '@aws-sdk/client-sagemaker-runtime';

export interface HiddenStateResult {
  generatedText: string;
  hiddenStates: Record<string, {
    mean: number[];
    lastToken: number[];
    norm: number;
  }>;
  logitsEntropy: number;
  generationProbs: number[];
}

export interface ShadowSelfConfig {
  endpointName: string;
  region: string;
  targetLayers: number[];
  maxNewTokens: number;
  temperature: number;
}

export class ShadowSelfClient {
  private readonly runtime: SageMakerRuntimeClient;
  private readonly config: ShadowSelfConfig;

  constructor(config: Partial<ShadowSelfConfig> = {}) {
    this.config = {
      endpointName: config.endpointName || 'cato-shadow-self',
      region: config.region || 'us-east-1',
      targetLayers: config.targetLayers || [-1, -4, -8],
      maxNewTokens: config.maxNewTokens || 256,
      temperature: config.temperature || 0.7
    };
```

```
  this.runtime = new SageMakerRuntimeClient({
    region: this.config.region
  });
}

async invokeWithHiddenStates(
  text: string,
  options: Partial<{
    targetLayers: number[];
    maxNewTokens: number;
    temperature: number;
  }> = {}
): Promise<HiddenStateResult> {
  const payload = {
    inputs: text,
    parameters: {
      target_layers: options.targetLayers || this.config.targetLayers,
      max_new_tokens: options.maxNewTokens || this.config.maxNewTokens,
      temperature: options.temperature || this.config.temperature,
      return_probs: true
    }
  };

  const command = new InvokeEndpointCommand({
    EndpointName: this.config.endpointName,
    ContentType: 'application/json',
    Body: JSON.stringify(payload)
  });

  const response = await this.runtime.send(command);
  const result = JSON.parse(
    new TextDecoder().decode(response.Body)
  );

  return {
    generatedText: result.generated_text,
    hiddenStates: result.hidden_states,
    logitsEntropy: result.logits_entropy,
    generationProbs: result.generation_probs
  };
}

estimateUncertainty(result: HiddenStateResult): number {
  // High entropy = high uncertainty
  const entropyScore = Math.min(1.0, result.logitsEntropy / 5.0);

  // Low average probability = high uncertainty
```

```
    const avgProb = result.generationProbs.length > 0
      ? result.generationProbs.reduce((a, b) => a + b, 0) / result.generationProbs.length
      : 0.5;
    const probScore = 1.0 - avgProb;

    // Combine scores
    return (entropyScore + probScore) / 2;
  }

  async getEndpointStatus(): Promise<{
    status: string;
    instanceCount: number;
  }> {
    // Implementation would use SageMaker client to describe endpoint
    return {
      status: 'InService',
      instanceCount: 10
    };
  }
}
```

Consequences

Positive

- **Full model access**: Hidden states, activations, logits all available
- **Customizable**: Can modify inference code as needed
- **Scalable**: Auto-scaling handles traffic spikes
- **Cost-effective**: SageMaker managed infrastructure

Negative

- **High cost**: ~$130K/month at 10M users (before discounts)
- **Operational overhead**: Managing custom containers
- **Cold starts**: New instances take ~5 minutes to start
- **Model updates**: Must redeploy to update model

Terraform Configuration

```
resource "aws_sagemaker_model" "shadow_self" {
  name               = "cato-shadow-self"
  execution_role_arn = aws_iam_role.sagemaker.arn

  primary_container {
    image          = "${aws_ecr_repository.shadow_self.repository_url}:latest"
    model_data_url = "s3://${aws_s3_bucket.models.id}/llama-3-8b-instruct/"
    environment = {
      MODEL_PATH = "/opt/ml/model"
    }
```

```
  }
}

resource "aws_sagemaker_endpoint_configuration" "shadow_self" {
  name = "cato-shadow-self-config"

  production_variants {
    variant_name           = "primary"
    model_name             = aws_sagemaker_model.shadow_self.name
    instance_type          = "ml.g5.2xlarge"
    initial_instance_count = 5

    managed_instance_scaling {
      status                     = "ENABLED"
      min_instance_count         = 5
      max_instance_count         = 300
    }
  }
}

resource "aws_sagemaker_endpoint" "shadow_self" {
  name                 = "cato-shadow-self"
  endpoint_config_name = aws_sagemaker_endpoint_configuration.shadow_self.name
}

resource "aws_cloudwatch_metric_alarm" "shadow_self_latency" {
  alarm_name          = "cato-shadow-self-high-latency"
  comparison_operator = "GreaterThanThreshold"
  evaluation_periods  = 3
  metric_name         = "ModelLatency"
  namespace           = "AWS/SageMaker"
  period              = 60
  statistic           = "p99"
  threshold           = 500  # 500ms
  alarm_description   = "Shadow Self latency exceeds 500ms"

  dimensions = {
    EndpointName = aws_sagemaker_endpoint.shadow_self.name
    VariantName  = "primary"
  }
}
```

Probe Training

Train linear probes on hidden states to detect properties:

```
# train_probes.py
```

```python
import numpy as np
from sklearn.linear_model import LogisticRegression
import pickle

def train_uncertainty_probe(
    hidden_states: List[np.ndarray],
    labels: List[int]  # 0 = confident, 1 = uncertain
) -> np.ndarray:
    """
    Train linear probe to detect uncertainty from hidden states.
    """
    X = np.array(hidden_states)
    y = np.array(labels)

    probe = LogisticRegression(max_iter=1000)
    probe.fit(X, y)

    return probe.coef_[0]

# Save probe for deployment
with open('uncertainty_probe.pkl', 'wb') as f:
    pickle.dump(probe_weights, f)
```

References

- SageMaker Real-Time Inference
- Llama 3 Model Card
- Activation Probing
- vLLM