SLNG Forward Deployed Speech Engineer - Technical Assessment

Duration: 2 hours
Format: Hybrid (Technical Implementation + Customer Scenario)
Focus Areas: Real-time speech processing, customer integration, problem-solving

---

Assessment Overview

This assessment simulates real-world scenarios you'll face as a Forward Deployed Engineer at SLNG, combining technical implementation with customer interaction skills. You'll work on speech API integration, debug production issues, and demonstrate customer communication abilities.

Part 1: Production Debugging Challenge (45 minutes)

You need to build a generic AI model inference router that can handle requests to multiple AI services with different protocols, implement retry logic, and manage concurrent requests efficiently.

Deliverable: Complete Python implementation

Build a Python class `AIModelRouter` that manages requests to multiple AI inference endpoints with the following requirements:

Core Functionality:

1. **Multiple Provider Support**: Handle HTTP REST and WebSocket connections
2. **Load Balancing**: Distribute requests across multiple instances of the same model
3. **Retry Logic**: Implement exponential backoff for failed requests
4. **Circuit Breaker**: Disable failing endpoints temporarily
5. **Concurrent Processing**: Handle multiple simultaneous requests efficiently
6. **Response Caching**: Cache responses for identical requests (with TTL)

Task 1: AI Model Router Implementation

This repository contains a solution for the SLNG technical task: a generic AI model inference router.

The router is implemented in Python using `asyncio` for asynchronous operations, `pydantic` for data validation, and `loguru` for logging.

Setup

This project uses `uv` for package management.

1. **Install dependencies:**

```
uv sync
```

Running the Demonstration

The `main.py` script starts mock AI provider servers and demonstrates the router's features, including basic routing, retries, and circuit breaking.

```
uv run python main.py
```

Testing

Unit tests are provided in `tests/test_model_router.py`. Run them using `pytest`:

```
uv run pytest
```

The project is formatted and linted with Ruff.

Part 2: Production Debugging Challenge (45 minutes)

Scenario

A customer's voice agent deployed through SLNG's AI gateway is experiencing intermittent latency spikes during peak hours. Users report delays of 3-5 seconds in AI model responses, but the issue only occurs 15% of the time.

Debugging Materials Provided:

```
Production Logs (Sample):
2025-09-18 14:23:15 [INFO] WebSocket connection established - session_id:
abc123
2025-09-18 14:23:17 [INFO] Audio stream started - routing to STT model
2025-09-18 14:23:18 [WARN] Model queue full - STT model instance
overloaded
2025-09-18 14:23:21 [ERROR] Model inference timeout - STT model
unresponsive
2025-09-18 14:23:21 [INFO] Fallback triggered - routing to backup STT
instance
2025-09-18 14:23:24 [INFO] Response sent - total_latency: 4.2s

SLNG Gateway Metrics:
- Model instances: 8/12 STT, 4/6 Speaker-ID, 6/8 TTS
- Average model inference time: 180ms STT, 120ms Speaker-ID
- Model queue depth: 15 requests (STT), 3 requests (Speaker-ID)
- Gateway CPU usage: 65% average, 90% peak
- Active WebSocket connections: 150 average, 400 peak
```

Frame the Problem

A client is complaining about long latencies (3-5 seconds) for 15% of their requests during peak hours. The client provided a log trace that corroborates this issue. The next step is to examine our Gateway metrics to identify the source of the problem.

Our debugging process is as follows:

1. Analyze the root cause by examining the provided logs and metrics.
2. Diagnose the problem and formulate a strategy to reproduce it.
3. Implement and test immediate solutions, then communicate the resolution to the client.
4. Propose long-term architectural changes to prevent this issue from recurring.

Analysis

The first problematic metric is for the **STT Model**: only `8/12 STT` instances are active. Four instances are idle while the active ones are overloaded (indicated by `Model queue full` in the logs and a queue depth of 15). This suggests a primary bottleneck in the gateway's ability to distribute requests to all available workers, pointing to a potential problem with the load balancer or scaling system.

Furthermore, it's critical that the **Gateway CPU usage** is hitting 90% during peak hours. This CPU pressure, combined with 400 active WebSocket connections, likely means the gateway is overwhelmed and is the underlying *reason* for the model under-utilization.

From these metrics, we can hypothesize that the `Model queue full` and `inference timeout` errors are correlated. Once a request is enqueued behind 15 other requests, its wait time plus its own inference time exceeds the timeout threshold, triggering the error and fallback mechanism.

Further Analysis

To confirm the root cause, we need to collect more granular, real-time data from the gateway and model instances.

- **Real-time Metrics:** Instead of a static snapshot, we need to observe real-time metrics. We should start by inspecting the load balancer's behavior to confirm if it is distributing traffic correctly across all STT instances.
- **Instance-Level Monitoring:** We need to check the CPU utilization of each model instance. This will help us understand if there's a scaling problem, especially since 4 instances are idle while others are overloaded. We should also track the number of requests processed per second by each instance.
- **Gateway Profiling:** To determine if the high CPU utilization is due to a software bug or simply high traffic, we should profile the gateway's performance during a latency spike. This will pinpoint the exact source of the high CPU usage.
- **Load Balancer Logs:** We should examine the load balancer logs to understand how it chooses endpoints and measure its own routing latency.

How can we implement this monitoring?

I would use the industry-standard tools: **Prometheus** for collecting the high-frequency, time-series data we need, and **Grafana** to visualize all key metrics on a single, unified dashboard.

How can we reproduce the issue?

- **Staging Environment:** Ideally, we would reproduce the issue in a controlled staging environment that mirrors our production setup. We can use a load testing tool to simulate realistic peak-hour traffic.
- **Live Environment (with caution):** If a representative staging environment is unavailable, we can attempt to reproduce the issue in the live environment. This would involve carefully using our service as a client would to experience the same problems. Once fixes are implemented, we can use the same method to verify the resolution. Tools designed for generating high-throughput WebSocket traffic would be essential here.

Proposed Solutions

We will divide the solutions into two categories: immediate fixes and long-term architectural changes.

Immediate Solutions

1. **Fully Utilize Existing Model Instances:** The most impactful immediate fix is to resolve the under-utilization of STT instances (`8/12`). We must modify the load balancer configuration to ensure traffic is distributed properly across all 12 available instances.
2. **Adjust Queue and Timeout Configuration:** The current settings lead to frequent timeouts. As a temporary measure, we can slightly increase the `inference_timeout` (e.g., from 3s to 5s) to reduce the number of failed requests while we address the underlying load distribution problem.

Long-Term Solutions

To solve the root cause and ensure future scalability, more fundamental changes are required.

1. **Optimize Load Balancer Strategy:** The core of the issue appears to be the load balancer's inability to distribute load effectively. We should experiment with different balancing strategies. A good candidate would be an algorithm that **routes requests to the instance with the shortest queue**.
2. **Implement Autoscaling for Model Instances:** The `8/12` metric implies a static pool of workers. We should implement a dynamic autoscaling solution. Using a **Horizontal Pod Autoscaler** in a Kubernetes environment, configured to scale based on CPU/memory utilization or custom metrics like queue depth, would allow the system to adapt to fluctuating demand automatically. If not already in use, migrating to Kubernetes should be strongly considered.

Prevention Strategies

1. **Proactive Monitoring and Alerting:** The issue was reported by a customer, which indicates a gap in our monitoring. We must implement a proactive monitoring and alerting system. Alerts should be configured to trigger on anomalies like high queue depth, increased inference latency, or unbalanced worker utilization, allowing us to address problems before they impact users.

Part 3: Customer Communication Simulation (30 minutes)

Scenario

You're on a video call with the CTO of a contact center software company. They're frustrated because their SLNG integration is causing dropped calls during high-volume periods.

Customer Email (Background):

```
Subject: URGENT - Production Issues with SLNG Integration

Hi SLNG Team,

We're experiencing critical issues with our SLNG AI gateway integration in
production.
During peak hours (2-4 PM CET), approximately 8% of calls are being
dropped
when voice transcription starts. This is impacting our SLA with clients.

Our engineering team suspects it's related to AI model scaling and request
routing,
but we're not sure how to properly configure SLNG's model deployment
parameters.

We need this resolved ASAP - we have an angry enterprise client
threatening
to cancel their contract.

Best regards,
Sarah Chen, CTO
VoiceFlow Systems
```

Reply

```
Hi Sarah,

Thank you for reaching out. I've received your email and want to assure
you that we understand the urgency and severity of this issue. Dropped
calls directly impacting your client's SLA is a critical situation, and we
are treating it with the highest priority.

Based on your description, the pattern of dropped calls during peak hours
when voice transcription starts points directly to a potential bottleneck
in AI model scaling or request routing. In peak hours with high volume a
model missconfiguration in how the gateway provisions resources or how it
queues requests to the appropriate models can cause trouble as you
mentioned.

To resolve this as quickly as possible, here is our proposed plan of
action:

1. **Immediate Call**: I'm available to jump on a call with your
engineering team immediately. This will allow us to review your current
SLNG configuration, specifically your model deployment parameters and
request routing logic, in real-time.
1. **Data Collection**: Please have your team prepare the following data
for the call:
```

```
    - A full copy of your current SLNG configuration.
    - Any relevant application logs and performance metrics.
    - The specific error codes associated with the dropped calls.

I've already started to look into your SLNG models' metrics to gain a deep
understanding of your specific case. We will then work with your team on
the call to analyze the data and pinpoint the exact root cause, whether
it's a model misconfiguration, an inefficient routing rule, or a resource
constraint. Please let me know what time works best for you and your team
to connect.

Kind regards,
Jan Leyva
```