SLO Recommendation Engine - Take Home Exercise

Overview

You are tasked with designing an AI-assisted system that analyzes metrics and dependencies across interconnected microservices and recommends appropriate SLOs/SLAs for each service — while accounting for upstream/downstream dependencies, datastores, and infrastructure components.

The system should expose an API that integrates into an internal developer platform (similar to Spotify's Backstage) used by developers to manage their microservices, deployments, and operational data.

You will provide a clear architectural diagram and accompanying design notes describing your approach.

Optionally, you may include a working demo hosted in a GitHub repository with setup instructions to run it locally.

Illustrative Example (For Context Only)

To ground your thinking, consider a simplified e-commerce system:

Services:
- `api-gateway` → depends on [`auth-service`, `product-service`, `checkout-service`]
- `auth-service` → depends on [`user-db`]
- `checkout-service` → depends on [`payment-service`, `inventory-service`]
- `payment-service` → depends on [`external-payment-api`]

Sample Metrics (last 30 days):
- `api-gateway`: p99 latency = 250ms, error rate = 0.1%
- `auth-service`: p99 latency = 50ms, error rate = 0.05%
- `checkout-service`: p99 latency = 800ms, error rate = 0.5%

Current SLOs:
- `api-gateway`: 99.9% availability, p95 < 200ms
- `checkout-service`: 99.5% availability, p95 < 1000ms

Design Question: How would your system recommend updated SLOs that account for:
- `checkout-service` depending on unreliable `external-payment-api`?

- The cascading impact on `api-gateway`?

You don't need to solve this specific example - use it as a reference for the types of problems your design should address.

Expected Deliverables

1. Architectural Diagram

Provide a high-level system diagram that illustrates:

- Core components and their interactions
- Data flow between ingestion, reasoning, evaluation, and visualization layers
- Interfaces (APIs, data sources, knowledge retrieval systems)
- Integration points with the internal developer platform

Your diagram should clearly show how data moves through the system — from metrics ingestion to SLO recommendation generation, validation, and delivery to developers.

2. Accompanying Design Notes

Provide concise documentation describing your design decisions and reasoning. Please include the following sections:

Architecture Overview

- Key system components:
  - Data ingestion and processing
  - Dependency modeling and graph construction
  - Recommendation engine
  - Knowledge/RAG layer
  - Evaluation, feedback, and governance
  - Observability and auditing
- Data flow and interfaces (APIs, message buses, or pipelines)
- Integration with the internal developer platform (e.g., service catalog APIs, plugins, or data sync jobs)

Dependency Modeling

- How service dependencies are represented and analyzed (graph schema, critical path detection, fan-in/fan-out, blast radius)
- How dependency structure influences SLO recommendations
- How your system handles circular, missing, or partial dependency information

SLO Recommendation Approach

- Inputs considered (historical performance, criticality, error budgets, dependencies, incidents, and resource utilization)
- Your methodology:
  - Rule-based
  - Statistical/optimization
  - ML or LLM-assisted reasoning
  - Hybrid (and why)
- How recommendations adapt to upstream and downstream constraints (propagation of latency/error)
- How trade-offs between feasibility and ambition are balanced

Knowledge & Reasoning Layer

If your design includes retrieval-augmented generation (RAG) or knowledge systems:

- Knowledge sources: What would you retrieve from? (e.g., SRE runbooks, SLO templates, incident history)
- Retrieval strategy: How do you ensure relevance and accuracy?
- Grounding: How do you validate and cite sources?
- Guardrails: How do you prevent hallucinations or incorrect recommendations?

Note: You may use rule-based heuristics, statistical methods, or AI/LLM approaches. Choose what makes sense and explain your reasoning.

Evaluation & Safety

Recommendation Quality:
- How recommendation quality is measured (backtesting, budget burn rate, incident correlation)
- Consistency checks across dependent services
- Feasibility validation against historical performance

Safety & Guardrails:
- How the system handles adversarial inputs or edge cases
- PII/sensitive data protection in metrics and logs
- Validation of AI-generated outputs (if using LLMs)
- Fallback behavior when confidence is low
- Rate limiting and abuse prevention for API endpoints

Explainability:
- What every recommendation must disclose (reasoning, data sources, confidence level)
- Audit trail for compliance and debugging
- Feedback mechanism for developers to report issues

Continuous Improvement:
- Feedback loop for learning from accepted/rejected recommendations
- A/B testing or shadow mode strategies
- Model drift detection and retraining approach

Scalability & Operations

- How the system scales to hundreds or thousands of services across multiple regions or tenants
- Data freshness guarantees and cost considerations
- Versioning, drift detection, and model retraining strategies
- Operational resilience and observability of the recommendation service
- API availability, performance, and rate-limiting considerations

Assumptions & Trade-offs (Critical Section)

We want to understand your reasoning process. Please explicitly document:

Assumptions:
- Available data sources (metrics format, granularity, retention)
- Scale characteristics (number of services, request volumes, regions)
- Developer platform capabilities (APIs, data model, auth)
- Team structure and operational maturity

Trade-offs & Design Decisions:
- Why you chose your recommendation methodology
- Accuracy vs. latency vs. cost considerations
- Level of automation vs. human oversight
- Where you'd invest engineering effort first vs. later

Out of Scope (but acknowledged):
- What you'd add with more time/resources
- Known limitations of your approach

3. Optional: Working Demo (GitHub Repository)

If you choose to build a minimal working demo:

- Host it in a public GitHub repository
- Include the following:
  - A README.md with clear setup and run instructions
  - All assets required to run locally (e.g., Docker Compose, lightweight config, or mock data)
  - Example API endpoints demonstrating:
    - Dependency ingestion
    - SLO recommendation retrieval
    - Integration callback with the internal developer platform (e.g., service catalog plugin or API update)

A simple REST or GraphQL API is sufficient. Focus on clarity, reasoning, and usability over completeness.

Expected API Contract (Illustrative Example)

Your API design should support workflows like:

```bash
 Ingest service dependency graph
POST /api/v1/services/dependencies

 Request SLO recommendations for a service
GET /api/v1/services/{service-id}/slo-recommendations

 Accept or modify a recommendation
POST /api/v1/services/{service-id}/slos

 Check dependency impact of proposed SLO change
POST /api/v1/slos/impact-analysis
```

Use this as inspiration, not prescription. Design the API that makes sense for your architecture.

Evaluation Criteria

Category	Weight	What Great Looks Like
Architecture & Clarity	20%	Logical, modular design with clear diagrams and data flow

Systems Thinking	20%	Deep understanding of dependencies, scale, and operational reality
Applied AI Judgment	20%	Sensible use of ML/LLMs; grounded reasoning and explainability
Integration & API Design	15%	Thoughtful developer experience and platform integration
Safety & Reliability	15%	Realistic approach to guardrails, failure modes, and data protection
Communication	10%	Clear documentation, explicit assumptions, thoughtful trade-offs

Guidelines

- Making Assumptions: This exercise intentionally leaves some details unspecified to simulate real-world ambiguity. Strong submissions will:
    - Identify key unknowns
    - Make reasonable assumptions and state them explicitly
    - Discuss how different assumptions would change your design

- Data & References:

    - No data or sample files are provided — make reasonable assumptions and state them explicitly
    - You may reference public datasets (e.g., Google Cloud Trace, DeathStarBench, CNCF Online Boutique) if needed for illustrative examples
    - Focus on clarity, realism, and architectural depth rather than implementation detail

- Submission Format: Your submission may include:
    - A single architectural diagram (required)
    - Accompanying design notes (required)
    - Optional GitHub link to a demo implementation (optional but encouraged)

- Submission: Please submit your deliverables via:
    - Email with attached documents and/or GitHub repository link
    - OR provide a GitHub repository URL containing all materials

Questions? Feel free to reach out if you need clarification on any aspect of this exercise. We look forward to seeing your design approach and reasoning!