Rackspace MLOps Architect - GCP - Technical Q&A

1. Question: Design a fault-tolerant, petabyte-scale data processing pipeline for ingesting and preparing a diverse text and code corpus for training a large language model (LLM). The system must process 5TB of new data daily. Detail your component choices, data orchestration, and how you would ensure data quality and lineage.

Answer: My approach would be a distributed, event-driven architecture built entirely on the Google Cloud Platform.

- **Ingestion:** Raw data (from web scrapes, internal documents, code repositories) would land in a dedicated **Google Cloud Storage (GCS)** bucket. A GCS event notification would trigger a **Cloud Function**, which logs metadata (source, timestamp, data type) into **Firestore** and places a message into a **Pub/Sub** topic for processing.
- **Orchestration & Processing:** I'd use **Cloud Composer**, GCP's managed Apache Airflow service, to orchestrate the ETL workflows. A pool of workers running on a **Google Kubernetes Engine (GKE)** cluster or using serverless **Dataflow** would pull messages from the Pub/Sub subscription. The core processing would be done using **Apache Spark on Dataproc** for its proven scalability. The Spark jobs would:
  1. **Decode & Clean:** Process raw text, handle different character encodings, and remove boilerplate (e.g., HTML tags, markdown formatting).
  2. **Data Quality & Filtering:** Automatically run validation checks, such as language detection, toxicity scoring, and flagging personally identifiable information (PII) using the Data Loss Prevention (DLP) API. Low-quality or sensitive data would be moved to a quarantine GCS bucket.
  3. **Deduplication & Tokenization:** Perform large-scale deduplication of documents and then tokenize the entire corpus using the target model's vocabulary.
  4. **Serialization:** The processed, tokenized data would be serialized into a query-efficient format like **TFRecord** or Parquet and stored in a versioned "gold" GCS data lake, ready for training.
- **Data Lineage & Governance:** I would leverage **Dataplex** for data governance and lineage. Every transformation step orchestrated by Cloud Composer would log metadata to Dataplex, tracking the journey of each data point from raw ingestion to its use in a specific model training run. This ensures 100% reproducibility, a critical requirement for building reliable models.

Technical Explainer: GCP Data Pipeline Technologies

The proposed architecture relies on a suite of robust, scalable, and managed GCP services designed for large-scale data engineering and MLOps.

Cloud Storage and Eventing: GCS & Cloud Functions

- **Google Cloud Storage (GCS):** GCS is Google's massively scalable and durable object storage service. It's the foundational storage layer for data lakes, ML training data, and model artifacts on GCP. Its key features, like storage classes, object versioning, and tight integration with other GCP services, are critical for MLOps.
- **Cloud Functions:** Google's serverless, event-driven compute service. It's the perfect "glue" for an event-driven architecture. In our pipeline, it responds to GCS events to trigger downstream processing, providing a cost-effective and scalable way to initiate workflows without managing any server infrastructure.

Workflow Orchestration: Cloud Composer

- **Cloud Composer:** A fully managed Apache Airflow service. It allows us to author, schedule, and monitor complex data pipelines using Python. By managing the underlying infrastructure, Composer lets the team focus on workflow logic rather than Airflow administration, which is a significant operational advantage for complex ETL.

Distributed Processing: Dataproc, Dataflow, and GKE

- **Dataproc:** GCP's managed service for running Apache Spark and Hadoop clusters. It allows for the rapid provisioning of clusters, which can be scaled or even made ephemeral (created only for the duration of a job) to optimize costs. It's ideal for heavy-duty, batch-oriented data transformations.
- **Dataflow:** A serverless, unified stream and batch data processing service based on Apache Beam. For certain ETL tasks, especially those involving streaming data or requiring autoscaling based on data volume, Dataflow can be more efficient and hands-off than managing a Dataproc cluster.
- **Google Kubernetes Engine (GKE):** The gold standard for managed Kubernetes. For custom processing logic or running tools that are not native to the Hadoop ecosystem, GKE provides the flexibility to run any containerized workload at scale.

Data Governance: Dataplex

- **Dataplex:** An intelligent data fabric that provides a unified view for data governance, quality, and discovery across GCS, BigQuery, and other data stores. Its automatic metadata harvesting and data lineage tracking are invaluable for building auditable and reproducible ML workflows.

2. Question: You are tasked with designing the distributed training infrastructure for a new 100B parameter foundation model. You have

access to a large cluster of GPU and TPU nodes on GCP. Compare and contrast how you would implement this using PyTorch on GPUs versus JAX on TPUs. Justify your choice and detail how you would optimize for maximum throughput using Vertex AI.

Answer: This is a strategic decision balancing ecosystem familiarity with hardware-specific performance. I would leverage **Vertex AI Training** to manage and scale this job, regardless of the chosen framework.

- **PyTorch on A100 GPUs:**
  - **Implementation:** I'd use PyTorch XLA with **Fully Sharded Data Parallelism (FSDP)**. FSDP shards the model's parameters, gradients, and optimizer states across all GPUs, drastically reducing the memory footprint on each device. The training job would be packaged in a custom container and submitted to Vertex AI Training, specifying a machine type with multiple NVIDIA A100 or H100 GPUs (e.g., `a2-ultragpu-8g`).
  - **Pros:** The vast majority of the ML community uses PyTorch. This approach offers easier adoption, a larger talent pool, and seamless integration with the extensive PyTorch ecosystem (HuggingFace, etc.).
  - **Cons:** While performant, it may not achieve the absolute peak performance or cost-efficiency that a TPU-native solution could.
- **JAX on TPUs:**
  - **Implementation:** JAX is the native framework for TPUs and is my recommendation for a project of this scale. I would leverage `jax.pjit` to define a parallelism strategy across a TPU pod slice (e.g., `v4-256`). This involves a combination of:
    1. **Data Parallelism:** Splitting the global batch size across all TPU cores.
    2. **Model Parallelism (Sharding):** Sharding the model's weights across the TPU cores, a necessity for a 100B parameter model.
  - **Pros:** JAX, combined with the XLA compiler and Google's custom-built TPU hardware, is designed for maximum performance on large-scale models. The high-speed Inter-chip Interconnect (ICI) on TPU pods is purpose-built for the communication patterns in large model training, often leading to superior scaling efficiency and better cost-performance.
  - **Cons:** JAX has a steeper learning curve due to its functional programming paradigm.
- **Justification & Optimization with Vertex AI:** For a foundational model where training is a multi-week, multi-million dollar effort, performance and cost-efficiency are paramount. **I would strongly advocate for the JAX/TPU approach.** The potential for faster training and lower cost justifies the engineering investment.

To maximize throughput using **Vertex AI Training**, I would:

1. **Leverage Vertex AI Experiments:** To systematically track and compare different training runs (e.g., varying hyperparameters, TPU pod slice sizes).
2. **Use Mixed-Precision Training:** Employ `bfloat16` for computation, which is natively supported and highly efficient on TPUs.
3. **Optimize Input Pipeline:** Use `tf.data` to build a highly performant input pipeline that can prefetch and feed data to the TPUs at line rate, ensuring the accelerators are never idle.
4. **Profile with the TPU Profiler:** Use the integrated profiler to identify and eliminate bottlenecks in the training loop, whether in data input, computation, or host-to-device communication.

3. Question: Design a high-throughput, low-latency inference service for an Agentic AI system that relies on a fine-tuned LLM. The service must handle 1,000 requests per second (RPS) with a p99 latency of less than 150ms. Explain your choice of tools, specifically addressing model optimization and scalable serving on GCP.

Answer: Meeting these SLOs for an LLM requires aggressive model optimization and a purpose-built serving platform. My entire solution would be built on **Vertex AI Prediction**.

- **Model Optimization:** First, I would take the fine-tuned model (e.g., from the PyTorch or JAX ecosystem) and optimize it for inference on NVIDIA GPUs.
    1. **Quantization:** I would use a technique like AWQ (Activation-aware Weight Quantization) to quantize the model weights from FP16/BF16 down to INT8 or even INT4. This dramatically reduces the GPU memory footprint and increases throughput with minimal impact on accuracy for LLMs.
    2. **Optimized Kernels:** I would use a framework like **TensorRT-LLM** or **vLLM**. These frameworks replace standard HuggingFace implementations with highly optimized kernels, including FlashAttention for the attention mechanism and paged attention (in vLLM's case) to manage memory for batched requests efficiently. The output is a highly optimized engine ready for deployment.
- **Scalable Serving with Vertex AI Endpoints:** I would package the optimized model and its serving code (using a framework like vLLM) into a custom container and deploy it to a **Vertex AI Endpoint**.
    1. **Endpoint Creation:** I'd define a Vertex AI Endpoint, specifying a machine type with a suitable GPU (e.g., L4 or A100). The L4 GPU is often a cost-effective choice for inference.
    2. **Continuous Batching:** The serving framework (vLLM) would handle continuous batching. Unlike static batching, this allows new requests to be added to the currently running batch, maximizing GPU utilization and reducing latency.

3. **Autoscaling:** I would configure the Vertex AI Endpoint's autoscaler based on **target GPU utilization (e.g., 75%)**. Vertex AI will automatically provision and de-provision model replicas to meet this target, ensuring we can handle the 1,000 RPS load while minimizing cost. To meet the p99 latency SLO, I would provision for a slightly higher capacity to absorb spikes.
4. **Monitoring:** Vertex AI Endpoints automatically export metrics to **Cloud Monitoring**. I would build a dashboard to track RPS, p50/p95/p99 latency, GPU utilization, and replica count, with alerts configured to fire if we are at risk of violating our SLOs.

4. Question: Your resume mentions building a scraping infrastructure that handled 1,000+ QPS. Describe how you would build and deploy that architecture on GCP.

Answer: That system was designed for high-throughput and robustness, and its architecture translates perfectly to GCP's managed services.

- **Architecture on GCP:** It would be a distributed system orchestrated by **Google Kubernetes Engine (GKE)**.
  - **Request Queue:** A **Redis-based queue like BullMQ** would manage the URLs to be scraped. I would deploy this on a **Memorystore for Redis** instance for high availability and low-maintenance operations.
  - **Scraping Workers:** A large fleet of containerized workers, each running **Playwright** for its ability to render JavaScript-heavy sites. These workers would be deployed as a GKE Deployment. I would use a **Horizontal Pod Autoscaler (HPA)** configured to scale the number of worker pods based on the queue depth in Redis.
  - **Proxy Management:** We would integrate with a smart proxy service that manages a pool of millions of residential and datacenter IPs. Each worker would fetch a new proxy for each session to maintain a low block rate. This component would run as a separate service within the GKE cluster.
  - **Egress Control:** To manage costs and potentially route traffic through specific regions, I would configure **Cloud NAT** for the GKE cluster's egress traffic.
- **Role of RedisBloom:** The key to achieving deduplication at 1,000+ QPS was using a **RedisBloom** filter, which I would also host on the Memorystore instance.
  1. Before adding a URL to the scrape queue, a worker would check if it was `probably` in the Bloom filter.
  2. If the filter returned "no," the URL was almost certainly new, and we'd add it to the queue and the filter.
  3. If it returned "yes," the URL was likely a duplicate, and we'd discard it. This probabilistic approach allowed us to perform deduplication checks in sub-millisecond time with a tiny memory footprint compared

to a traditional set, which is essential for maintaining high throughput. Deploying this on GKE with Memorystore provides a fully scalable, managed, and robust solution for large-scale data acquisition.