Zoox Principal ML Infrastructure Engineer - Technical Interview Q&A

1. Question: Design a fault-tolerant, petabyte-scale data processing pipeline for ingesting and preparing multimodal sensor data (Lidar, Camera, Radar) for autonomous vehicle model training. The system must process 15TB 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.

- **Ingestion:** Data from our vehicles would land in a dedicated S3 bucket. An S3 event notification would trigger an AWS Lambda function, which logs metadata (vehicle ID, timestamp, sensor type) into a DynamoDB table and places a message into an SQS queue for processing.
- **Orchestration & Processing:** I'd use **Airflow**, as I did at SpaceKnow, to orchestrate the ETL workflows. A pool of workers running on a Kubernetes cluster would pull messages from SQS. The core processing would be done using **Apache Spark** for its proven scalability in handling terabyte-scale data, as demonstrated by my experience processing 14TB/day at The Climate Corporation. The Spark jobs would:
    1. **Decode & Synchronize:** Unpack raw sensor data and time-synchronize streams from different sensors (Lidar, Camera, Radar) to create coherent "scenes."
    2. **Feature Extraction:** Run initial feature extraction. For example, using PyTorch models on GPU worker nodes to perform object detection on images or ground plane segmentation on Lidar point clouds.
    3. **Data Quality Checks:** Automatically run validation checks, such as ensuring data completeness, sensor calibration verification, and flagging anomalous sensor readings. Bad data would be moved to a quarantine location for manual review.
    4. **Serialization:** The processed, synchronized scenes would be serialized into a query-efficient format like **Parquet** and stored in a versioned "gold" S3 data lake.
- **Data Lineage & Governance:** I would integrate a data lineage tool to track the journey of each data point from raw ingestion to its use in a specific model training run. This would involve logging the exact code versions (git commits), Spark job configurations, and data versions used for each transformation. This ensures 100% reproducibility, a core focus of my past work.

2. Question: You are tasked with designing the distributed training infrastructure for a foundational perception model at Zoox. The model has 20 billion parameters. You have a cluster of 64 nodes, each with 8 NVIDIA A100 80GB GPUs. Compare and contrast how you

would implement this using PyTorch (FSDP) versus JAX. Justify your choice and detail how you would optimize for maximum throughput.

Answer: This is a classic trade-off between ecosystem maturity and raw performance.

- **PyTorch (FSDP):**
  - **Implementation:** I would use PyTorch's Fully Sharded Data Parallelism (FSDP). FSDP shards the model's parameters, gradients, and optimizer states across all GPUs. During the forward/backward pass, an `all_gather` operation collects the necessary full layer parameters on each GPU just before they are needed, and they are discarded immediately after. This dramatically reduces the peak GPU memory footprint.
  - **Pros:** Easier to adopt for teams already familiar with PyTorch. It integrates seamlessly with the existing PyTorch ecosystem (e.g., `DataLoader`, `torch.distributed`).
  - **Cons:** Can sometimes be a "black box," offering less fine-grained control than JAX.
- **JAX:**
  - **Implementation:** JAX offers more explicit control. I would leverage `jax.pjit` (or the older `pmap`) to define the parallelism strategy. For a model of this size, a 3D parallelism approach would be optimal:
    1. **Data Parallelism:** The global batch size is split across all 512 GPUs (64 nodes * 8 GPUs).
    2. **Tensor Parallelism:** Within each node, the parameters of large layers (like attention blocks) are sharded across the 8 GPUs.
    3. **Pipeline Parallelism:** The model layers are grouped into stages, and each stage is assigned to a set of nodes. This keeps all GPUs busy by working on different micro-batches simultaneously.
  - **Pros:** JAX's XLA compiler can perform aggressive optimizations, often leading to higher raw throughput. The explicit nature of `pjit` gives precise control over how the model and data are sharded.
  - **Cons:** Steeper learning curve due to its functional programming paradigm and the need to manage state explicitly.
- **Justification & Optimization:** For a foundational model where training is a multi-week effort and performance is paramount, **I would choose JAX.** The upfront engineering investment is justified by the potential for faster iteration. To maximize throughput, I would:
  1. **Use Mixed-Precision Training:** Employ `bfloat16` for computation and `float32` for master weights to cut memory usage nearly in half and leverage the A100's Tensor Cores.
  2. **Optimize Communication:** Use NVIDIA's NCCL for `all_reduce` and `all_gather` operations, ensuring the network fabric is not a bottleneck.
  3. **Tune Batch Size:** Profile the training job to find the maximum batch size that fits in GPU memory to improve utilization. My goal

would be to exceed 90% GPU utilization and achieve a throughput of several thousand tokens per second per GPU.

3. Question: Design a high-throughput, low-latency inference service for a deployed object detection model. The service must handle 5,000 requests per second (RPS) with a p99 latency of less than 30ms. Explain your choice of tools, specifically addressing model optimization with TensorRT and scalable serving with a framework like Ray Serve.

Answer: Meeting these strict SLOs requires optimization at every level.

- **Model Optimization with TensorRT:** First, I would take the trained PyTorch model and convert it to the ONNX format, which I have experience with. Then, I'd use **NVIDIA TensorRT** to perform several key optimizations:
  1. **Graph Fusion:** Fuse multiple layers (e.g., convolution, bias, and ReLU) into a single kernel, reducing kernel launch overhead.
  2. **Precision Calibration:** Quantize the model weights from FP32 to INT8. This provides a significant speedup (up to 3-4x) with minimal accuracy loss. I would use a representative calibration dataset to determine the optimal quantization thresholds.
  3. **Kernel Auto-Tuning:** TensorRT profiles and selects the fastest available kernels for the target GPU (e.g., A100). The output is a highly optimized TensorRT engine, which I've seen provide a **5-10x reduction in latency** compared to a naive framework deployment.
- **Scalable Serving with Ray Serve:** I would use **Ray Serve** to build the serving infrastructure on a Kubernetes cluster.
  1. **Deployment Graph:** I'd define a Ray Serve deployment graph. The entry point would be a lightweight "router" deployment that accepts incoming requests.
  2. **Dynamic Batching:** The router would perform dynamic batching, collecting individual requests for a few milliseconds (e.g., 5-10ms) and batching them together to feed to the model. This is crucial for maximizing GPU throughput.
  3. **Model Replicas:** The router would forward these batches to a pool of "model" deployments, each running the optimized TensorRT engine on a dedicated GPU. Ray Serve would manage these replicas.
  4. **Autoscaling:** I would configure Ray Serve's autoscaler to monitor the queue depth and latency. If the p99 latency approaches our 30ms SLO, it would automatically scale up the number of model replicas to handle the load. For a 5,000 RPS target, assuming a batch size of 32 and a 20ms processing time per batch on one GPU, we'd need approximately `5000 / (32 / 0.020) = ~3.125` GPUs. I would provision at least 4-5 replicas for redundancy and to handle spikes.
- **Monitoring:** I would expose Prometheus metrics from Ray Serve for RPS, p50/p95/p99 latency, and batch size, and use NVIDIA DCGM to monitor

GPU utilization and memory on each serving node.

4. Question: Your resume mentions building a scraping infrastructure that handled 1,000+ QPS. Describe the architecture in detail. How did you handle IP rotation, session management, and JavaScript-heavy websites? How did RedisBloom fit into this?

Answer: This system was designed for high-throughput and robustness.

- **Architecture:** It was a distributed system orchestrated by Kubernetes.
  - **Request Queue:** A **BullMQ** instance (a Redis-based queue I've used) managed the queue of URLs to be scraped. This provided job persistence and retry logic.
  - **Scraping Workers:** A large fleet of containerized workers, each running **Playwright**. Playwright was chosen over simple HTTP clients because it can render JavaScript and handle complex web applications, which was essential for our targets.
  - **Proxy Management:** We didn't just rotate proxies; we used a smart proxy service that managed a pool of millions of residential and datacenter IPs. Each Playwright worker would request a new proxy from this service for each session, ensuring a low block rate.
  - **Session Management:** For sites requiring logins, we had a separate system for managing user sessions (cookies, local storage) which could be injected into Playwright's browser contexts.
- **Role of RedisBloom:** At 1,000+ QPS, you can't simply store every scraped URL in a traditional database and check for existence before scraping. The latency would be too high. I used a **RedisBloom** filter, which is a probabilistic data structure.
  1. Before adding a URL to the scrape queue, we 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 allowed us to perform deduplication checks in sub-millisecond time with a very low, tunable false-positive rate (e.g., <0.01%) while using significantly less memory than a set-based approach. This was the key to achieving deduplication at that scale.