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.

Technical Explainer: Data Pipeline Technologies

The proposed architecture relies on a suite of robust, scalable, and battle-tested technologies designed for large-scale data engineering. Each component is chosen for a specific role, and their integration creates a powerful and resilient system.

Cloud Storage and Eventing: AWS S3 & Lambda

- **AWS S3 (Simple Storage Service):** Introduced by Amazon Web Services in 2006, S3 is the de facto standard for object storage in the cloud. Its design philosophy is centered around providing extreme durability (99.999999999%, or "11 nines") and massive scalability. Unlike a traditional file system, S3 stores data as "objects," which consist of the data itself, a unique key, and metadata. This abstraction allows it to store trillions of objects and handle millions of requests per second. For our use case, S3 serves as the landing zone for raw sensor data. Its key features, like storage classes (for cost optimization) and event notifications, are critical. The main alternative is Google Cloud Storage, which offers similar functionality. The primary drawback of S3 is the data egress cost, which can be substantial for large-scale data transfer out of AWS.
- **AWS Lambda:** Launched in 2014, Lambda pioneered the concept of "serverless" computing. It allows you to run code without provisioning or managing servers. In our pipeline, Lambda acts as the glue between S3 and the processing queue. When a new data file lands in S3, an event notification triggers a Lambda function. This function is responsible for lightweight tasks: extracting metadata, logging it, and placing a message in an SQS queue. This event-driven approach is highly efficient and cost-effective, as you only pay for the compute time consumed, measured in milliseconds. The main limitation is the 15-minute maximum execution time, making it unsuitable for long-running jobs, but perfect for this kind of trigger-based orchestration.

Workflow Orchestration: Apache Airflow

- **Apache Airflow:** Created by Airbnb and now a top-level Apache Software Foundation project, Airflow is a platform for programmatically authoring, scheduling, and monitoring workflows. Workflows are defined as Directed Acyclic Graphs (DAGs) in Python, which provides immense flexibility. Airflow's strength lies in its ability to manage complex dependencies between tasks, handle retries and failures, and provide a rich user interface for visualizing and managing pipelines. In our design, Airflow would orchestrate the entire ETL process, from pulling messages from SQS to launching Spark jobs and monitoring their completion. While powerful, Airflow can be complex to set up and manage, and its scheduler can be a performance bottleneck. Alternatives like Prefect and Dagster have emerged to address some of these challenges, offering more modern and developer-friendly interfaces.

Distributed Processing: Apache Spark & Parquet

- **Apache Spark:** Originating from UC Berkeley's AMPLab, Spark has become the leading open-source framework for large-scale data processing. Its primary innovation over the original MapReduce paradigm was its ability to perform computations in-memory, leading to significant performance gains. Spark's core abstraction is the Resilient Distributed Dataset (RDD), an immutable, distributed collection of objects. It provides a rich set of APIs for data manipulation (in Scala, Java, Python, and R) and supports SQL (Spark SQL), streaming (Structured Streaming), and machine learning (MLlib). For our pipeline, Spark's ability to scale horizontally across a Kubernetes cluster is essential for processing 15TB of data daily. Its main drawback is its complexity; tuning Spark jobs for optimal performance can be a dark art.
- **Apache Parquet:** A high-performance, columnar storage format is crucial for efficient data analysis. Parquet, created by Cloudera and Twitter, stores data in columns rather than rows. This is highly advantageous for analytical queries, as it allows the query engine to read only the necessary columns, significantly reducing I/O. Parquet also features excellent compression and encoding schemes, which further reduce storage costs and improve query performance. In our design, the final, processed data is stored in Parquet format in our "gold" S3 data lake, making it readily available for model training and analysis. The main alternative is Apache ORC, which offers similar benefits.

By combining these technologies, we create a pipeline that is not only capable of handling the massive data volume but is also fault-tolerant, reproducible, and extensible.

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.

Technical Explainer: Distributed Training Frameworks

Training a 20-billion-parameter model requires a sophisticated distributed training strategy. The choice of framework, either PyTorch or JAX, has significant implications for implementation complexity, performance, and developer velocity.

PyTorch and Fully Sharded Data Parallelism (FSDP)

- **PyTorch:** Developed by Facebook's AI Research lab (FAIR) and released

in 2016, PyTorch has become a dominant force in the deep learning landscape. Its popularity stems from its "Pythonic" feel, imperative programming style, and dynamic computational graphs. This makes it intuitive for developers and researchers to quickly prototype and debug models. The `torch.distributed` library provides the foundation for distributed training.

- **Fully Sharded Data Parallelism (FSDP):** FSDP is a powerful data parallelism strategy that goes beyond simple data parallelism (where the entire model is replicated on each GPU). With FSDP, the model's parameters, gradients, and optimizer states are sharded (distributed) across all available GPUs. During the forward and backward passes, each GPU gathers only the necessary model parameters for the layer it is currently computing, and discards them immediately afterward. This "just-in-time" parameter fetching dramatically reduces the peak memory usage on each GPU, allowing you to train much larger models. FSDP is a more user-friendly approach to model sharding compared to manual tensor parallelism, as it handles the communication and sharding logic automatically. However, this abstraction can sometimes come at the cost of performance and control compared to more explicit frameworks like JAX.

JAX and 3D Parallelism

- **JAX:** Developed by Google Research, JAX is a library for high-performance machine learning research. It combines a NumPy-like API with a powerful JIT (Just-In-Time) compiler called XLA (Accelerated Linear Algebra). JAX's functional programming paradigm and composable transformations (`grad`, `jit`, `vmap`, `pmap`) allow for the creation of highly optimized and efficient code.

- **3D Parallelism:** For massive models, a single parallelism strategy is often insufficient. JAX's flexibility allows for the combination of three distinct parallelism techniques:

  1. **Data Parallelism:** The most common form, where the training data is split across multiple devices, and each device has a full copy of the model.
  2. **Tensor Parallelism (or Intra-layer Model Parallelism):** The weights of a single large layer (e.g., a transformer block) are sharded across multiple devices. This is essential when a single layer is too large to fit on one GPU.
  3. **Pipeline Parallelism (or Inter-layer Model Parallelism):** The model is split into stages, and each stage is placed on a different set of devices. Data flows through the pipeline, with each stage processing a different micro-batch simultaneously. This helps to keep all GPUs busy and improves overall throughput.

  Implementing this 3D parallelism in JAX using `pjit` provides fine-grained control over how the model and data are distributed, often leading to

superior performance compared to more automated solutions. However, it requires a deeper understanding of distributed systems and a more significant engineering effort.

Optimization and Communication

- **Mixed-Precision Training:** This technique uses a combination of lower-precision (e.g., `bfloat16` or `float16`) and higher-precision (`float32`) data types during training. It significantly reduces memory usage and can leverage specialized hardware like NVIDIA's Tensor Cores for faster computation.
- **NVIDIA NCCL (NVIDIA Collective Communications Library):** NCCL is a library that provides highly optimized implementations of collective communication operations (e.g., `all_reduce`, `all_gather`) for NVIDIA GPUs. It is the backbone of multi-GPU and multi-node training, ensuring that the communication between GPUs is as efficient as possible.

In summary, the choice between PyTorch/FSDP and JAX/3D-parallelism is a trade-off between ease of use and raw performance. For a foundational model where training is a long and expensive process, the upfront investment in JAX can pay significant dividends in terms of faster iteration and reduced training costs.

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.

Technical Explainer: Inference Optimization and Serving

Achieving high-throughput, low-latency inference is a critical challenge in production machine learning. This requires a two-pronged approach: optimizing the model itself and building a scalable serving infrastructure.

Model Optimization: ONNX and TensorRT

- **ONNX (Open Neural Network Exchange):** Before we can optimize a model, we need a standard way to represent it. ONNX, created by Microsoft and Facebook, provides an open format for deep learning models. It acts as an intermediary representation, allowing you to train a model in one framework (like PyTorch) and then deploy it for inference using another (like TensorRT). This decoupling of training and serving is a key principle of MLOps.
- **NVIDIA TensorRT:** TensorRT is a high-performance deep learning inference optimizer and runtime from NVIDIA. It takes a trained model (in ONNX or other formats) and applies a series of aggressive optimizations to prepare it for deployment on NVIDIA GPUs. Key optimizations include:
    - **Graph Fusion:** TensorRT analyzes the model's computational graph and fuses multiple layers into a single, highly optimized kernel. This reduces the overhead of launching multiple CUDA kernels and improves GPU utilization.

- **Precision Calibration (Quantization):** One of the most effective optimizations is quantization, which involves converting the model's weights and activations from 32-bit floating-point (FP32) to lower-precision formats like 16-bit floating-point (FP16) or 8-bit integer (INT8). This reduces the model's memory footprint and can significantly speed up computation, especially on GPUs with specialized hardware like Tensor Cores.
- **Kernel Auto-Tuning:** TensorRT profiles a variety of pre-built and auto-generated kernels for each layer and selects the fastest one for the target GPU architecture.

The result of these optimizations is a "TensorRT engine," a self-contained, optimized model that can deliver significantly lower latency and higher throughput than the original framework model.

Scalable Serving: Ray Serve

- **Ray and Ray Serve:** Ray is an open-source framework for building distributed applications. Ray Serve is a scalable model serving library built on top of Ray. It is designed to handle the complexities of production model serving, including high-throughput, low-latency requirements, and the need for dynamic scaling.
- **Key Features of Ray Serve for this use case:**
  - **Deployment Graph:** Ray Serve allows you to define a "deployment graph," which is a pipeline of models and business logic. In our design, we have a simple graph with a router and a model deployment.
  - **Dynamic Batching:** This is a critical feature for GPU-based inference. GPUs are most efficient when processing data in large batches. Ray Serve can automatically collect individual requests as they arrive and batch them together before sending them to the model. This maximizes GPU throughput and reduces the cost per request.
  - **Autoscaling:** Ray Serve can automatically scale the number of model replicas up or down based on the incoming traffic. This ensures that you have enough capacity to handle peak loads while minimizing costs during idle periods. It integrates with Kubernetes' Horizontal Pod Autoscaler (HPA) for seamless scaling.
  - **Framework-Agnostic:** Ray Serve can serve models from any machine learning framework, making it a flexible and future-proof choice.

By combining the model-level optimizations of TensorRT with the infrastructure-level scalability of Ray Serve, we can build an inference service that meets the demanding requirements of a real-time autonomous driving system.

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.

Technical Explainer: High-Throughput Web Scraping

Building a web scraping system that can handle over 1,000 queries per second (QPS) is a significant engineering challenge. It requires a distributed architecture, sophisticated tools for browser automation, and clever techniques for managing state and avoiding detection.

Orchestration and Automation

- **Kubernetes:** At this scale, a container orchestration system is essential. Kubernetes, the de facto standard, allows us to manage a large fleet of containerized scraping workers. It handles deployment, scaling, and self-healing, ensuring that our scraping infrastructure is always available and can adapt to changing workloads.

- **Playwright:** Modern websites are heavily reliant on JavaScript to render content. Simple HTTP clients like `requests` or `curl` are often insufficient, as they only retrieve the initial HTML and do not execute the necessary JavaScript to load the full page content. Playwright, a browser automation library from Microsoft, solves this problem by controlling a real browser (Chromium, Firefox, or WebKit). It can render JavaScript, handle complex user interactions, and provide access to the full DOM, making it an ideal tool for scraping modern web applications.

Queueing and Deduplication

- **BullMQ:** A robust queueing system is the backbone of any distributed scraping architecture. BullMQ, a Redis-based queue for Node.js, is a great choice for this task. It provides features like job persistence (so we don't lose URLs if a worker fails), delayed jobs, and atomic operations, all of which are essential for building a reliable system.
- **RedisBloom and Probabilistic Data Structures:** One of the biggest challenges in large-scale scraping is deduplication: how do you avoid scraping the same URL over and over again? A naive approach would be to store every scraped URL in a database and check for existence before adding a new URL to the queue. However, at 1,000+ QPS, this would create a massive bottleneck. The database would struggle to keep up with the high volume of reads and writes.

  This is where probabilistic data structures like Bloom filters come in. A Bloom filter is a space-efficient data structure that is used to test whether an element is a member of a set. It can produce false positives (it might say an element is in the set when it's not), but it will never produce false negatives (if it says an element is not in the set, it's definitely not).

  RedisBloom, a Redis module, provides a highly optimized implementation of a Bloom filter. By using RedisBloom, we can perform deduplication checks in sub-millisecond time with a very small memory footprint. The trade-off is a small, tunable false-positive rate, which is perfectly acceptable for this use case (it's okay if we occasionally miss a URL, as long as we don't overwhelm our system with duplicates). This probabilistic approach is the key to achieving deduplication at scale.

By combining these technologies, we can build a scraping system that is not only fast and scalable but also robust and resilient enough to handle the challenges of the modern web.