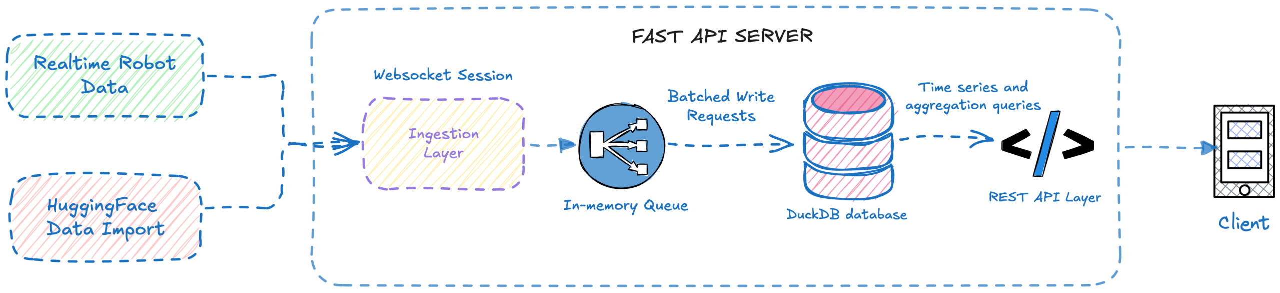
Architecture Decisions

I won't lie, this was a really interesting project to build. A lot of decisions came from semi-extensive research and prior experience.



Storage Format Decision

My database choice was DuckDB with Parquet as the underlying storage format. What influenced this:

- Type of query operations needed were analytical (time range scans and aggregations).
- Query speed needed to hit sub-200ms reads on a 60 second window.
- DuckDB handles .parquet files natively which reduced latency since we don't need to transform the shape or format of the data on the way in or out.
- It also accepts float vectors of dimension 1536 easily for embeddings.

Images are stored separately from structured telemetry data, raw bytes go to the filesystem, only the path and metadata live in the database. This keeps the query fast. The tradeoff is that Parquet isn't designed for high-frequency writes, which created real problems on the write side.

Backpressure and Buffering

DuckDB has a single write lock, so multiple robots writing concurrently serialise immediately. I handled this with an in-memory queue that batches write requests and flushes them to the database in bulk. This helped but introduced two failure modes:

- Ungraceful disconnects: anything in the queue that hasn't flushed is lost. I mitigate this with a flush on shutdown but that only helps clean exits.

- If the queue fills under sustained load the server returns an error to the client so it can handle accordingly, but honestly at that point there's not much more we can do server-side and some data loss becomes unavoidable..

AI Feature Rationale

I went with similarity search over episode summaries using OpenAI's text-embedding-3-small, storing 1536-dimensional vectors in DuckDB and querying by cosine similarity. The idea was to let operators find relevant sessions in plain language rather than writing queries directly.

Honestly I wasn't pleased with the results most queries returned low similarity scores. The weaknesses are that short generic summaries don't embed with enough signal/context, and I wasn't filtering out low-confidence results. Fixing this means applying a similarity threshold around 0.70, using richer text summaries, and potentially trying a different embedding model.

Scale

For 100 concurrent robots and 6 months of retention, DuckDB doesn't work the write lock alone rules it out. I looked at three realistic paths:

- PostgreSQL gives you concurrent writes but you lose analytical query performance.
- TimescaleDB is better through Hypertables. They bring query speeds much closer to acceptable and it scales reasonably for 100 robots.
- ClickHouse is what I'd actually want at real scale: columnar, fast reads, handles high-concurrency writes. The drawback is operational complexity, but for 5,000+ concurrent sessions it's the right call.

Images would also be stored separately in an s3 bucket, only the link would live in the database. Also, for 6 months of data you'd also need different data stores: Live data in the primary store, older data in object storage like S3 as Parquet files, with a query layer that spans both.