Design Document: Dynamic Protobuf-based Task Orchestration with Redis Streams

1. Problem Statement

We need to build a system where an orchestrator coordinates multiple tasks running across different machines.

- The output of one task should be shared with another.
- Output format is structured, strongly typed, but not known at compile time.
- System must support dynamic schemas, cross-machine communication, and scalable message passing.

2. Requirements

Functional:

- Producers should define message schemas dynamically at runtime.
- Schemas should be shareable across machines.
- Consumers should be able to fetch schemas dynamically and deserialize messages.
- Support for complex data types: nested messages, repeated fields, maps, map-of-lists.

Non-Functional:

- Scalability: Multiple producers/consumers across distributed setup.
- Reliability: Messages should not be lost, schema consistency guaranteed.
- Observability: Logs, metrics, and schema version tracking.
- Developer Experience: Easy-to-use library for schema registration, serialization, deserialization.

3. High-Level Architecture

Producer → Redis (Schemas + Stream) → Consumer

Producer Flow:

1. Dynamically define schema (FileDescriptorProto)

2. Auto-version schema (hash/version)
3. Validate schema

4. Register schema in local DescriptorPool

5. Generate dynamic message class via MessageFactory

6. Serialize message to binary

7. Push schema to Redis hash (if new)

8. Push message to Redis Stream (payload + schema_id)

Redis:

- Stream: messages (payload, schema_id)
- Hash: schemas (schema_id → FileDescriptorProto bytes)
- Supports replication, trimming, persistence

Consumer Flow:

4. Read message from Redis Stream
5. Extract schema_id
6. Check local DescriptorPool cache

- Exists: use cached descriptor
- Not exists: fetch FileDescriptorProto from Redis hash → load into pool

4. Create dynamic message class via MessageFactory

5. Deserialize binary payload → dynamic message instance

6. Process message / optionally produce downstream messages

7. Metrics/logging for observability

8. Error handling: missing/corrupt schema → dead-letter queue

4. Data Model

Schemas:

- Stored in Redis Hash: `schemas`
- Key: schema_id (hash of FileDescriptorProto bytes or version string)
- Value: FileDescriptorProto serialized

Messages:

- Stored in Redis Stream: `messages`
- Fields:
- schema_id: string
- payload: serialized Protobuf binary

5. Example

Schema (Person):

message Person {

string name = 1;

int32 age = 2;

}

Stored in Redis as:

"schemas": {

"person_v1": <FileDescriptorProto bytes>

}

Message sent to stream:

"messages": [

{

"schema_id": "person_v1",

"payload": <binary>

}

]

6. Alternatives Considered

- Kafka + Confluent Schema Registry

Pros: Battle-tested, integrates well with Protobuf/Avro/JSON.

Cons: Extra infra dependency, more complex than Redis for smaller setups, not in memory

- JSON instead of Protobuf

Pros: Human-readable, no schema sharing needed.

Cons: Verbose, slower, no strong typing.

7. Improvements / Extensions

- Schema Versioning: Auto-generate schema_id using hash of FileDescriptorProto.
- Caching: Consumers cache dynamic descriptors locally to avoid repeated Redis fetches.
- Consumer Groups: Scale out consumers with Redis consumer groups.
- Observability: Schema logs, metrics for processing latency, dead-letter queue.
- Compression: Add gzip/snappy for very large payloads.
- Typed Code Generation: Optionally generate Python dataclass or TypedDict for improved IDE support.

8. Open Questions

- Should schema registry be Redis-only, or do we add a dedicated DB (e.g., PostgreSQL for audit trail)?
- Should we allow backward compatibility checks before schema updates?
- How to handle stream trimming (time vs size based)?