- **System Layers**
- **Data Flow (ingestion → processing → storage → optimization → response)**
- **Technology Choices (why each)**
- **Integration points (Rust ↔ Python ↔ SurrealDB ↔ APIs)**
- **Prototype execution flow (step-by-step)**

🚆 End-to-End Workflow: Railway Intelligent Decision-Support System

1. System Architecture Layers

1. **Data Sources (Indian Railways context)**
   - Real-time data: Train running status, delays, location, timetable, platform availability, signalling.
   - APIs:
     - RailwayAPI (3rd-party, Indian Railways) – live train running status, PNR, station schedule.
     - [IRCTC unofficial APIs] (real-time train positions via scraping).
     - [NTES APIs (if accessible via trial, Indian Railways official feed)].
   - Synthetic data: fallback when APIs are not open → simulate train movements, delays, signalling disruptions.
2. **Data Engineering (Rust)**
   - **Ingestion Pipelines**: Rust async services (Tokio, Reqwest) to pull from APIs/simulators.
   - **Transform & Normalize**: map raw feeds → unified schema (train_id, section_id, priority, delay, location, ETA, etc.).
   - **Stream Processing**: use `Kafka` or `NATS` for real-time feed, optional (or keep lightweight using `tokio::mpsc`).
   - **Storage**: Persist in **SurrealDB** (graph + time-series like queries for section conflicts).
3. **Backend (Rust)**
   - Expose APIs to:
     - Query current train states.
     - Trigger optimization request for a corridor.
     - Serve controller dashboards (via frontend).
   - Adaptor for **SurrealDB** (Rust driver).
   - RPC/bridge with Python models (via gRPC or PyO3 bindings).
4. **Optimization & AI/ML (Python)**

- **OR-Tools (constraint programming)**:
    - Model precedence rules, track sections, crossing constraints, priorities.
    - Solve for minimal delay & maximal throughput.
- **AI/ML**:
    - Predict train delays based on historical data/weather.
    - Reinforcement learning for “what-if” re-optimization.

5. **Controller Dashboard (UI)**
    - Displays:
        - Suggested precedence & crossing schedules.
        - Conflict resolution explanation.
        - What-if scenario comparisons.
        - KPIs (punctuality, throughput, avg delay).
    - Exposed via Rust backend APIs → React/Next.js frontend.

---

2. Data Flow (Step-by-Step)

Step 1 – Ingestion

- Rust async jobs query real-time APIs every X seconds.
- Data is validated & normalized into a **common schema**.
- Push into **SurrealDB** (historical + live tables).

Step 2 – Processing

- Rust event pipeline listens to new updates (train enters section, delay reported).
- Backend triggers **Python OR-Tools service** for re-optimization when:
    - Delay > threshold.
    - Section conflict (two trains scheduled for same crossing).
    - Controller requests “re-simulate”.

Step 3 – Optimization

- Python service fetches required slice of data from SurrealDB.
- Runs constraint solver (precedence + crossing optimization).
- Produces new feasible schedule → sends back via gRPC/REST to Rust backend.

Step 4 – Response to Controllers

- Backend persists optimization result into SurrealDB (audit + retrieval).
- Controller dashboard shows recommendation + “why” (e.g., Freight train delayed 15min, Passenger train gets priority).

- What-if analysis: user tweaks → re-submit to optimizer.

---

3. Technology Choices & Justification

Layer	Tech	Why
Data Ingestion	Rust (Tokio + Reqwest)	Fast, async, memory safe for real-time APIs
Stream Processing (opt)	Kafka/NATS or just `tokio::mpsc`	Lightweight but scalable
Storage	SurrealDB	Graph + time-series + SQL-like queries → perfect for train networks
Backend APIs	Rust (Axum/Actix)	High-perf, integrates with SurrealDB
OR/AI	Python (OR-Tools + scikit-learn/PyTorch)	Rich ecosystem for optimization & ML
Integration	gRPC or PyO3 bindings	Clear contract between Rust backend and Python solver
Dashboard	React/Next.js	Simple, interactive visualization for controllers

---

4. Prototype Execution Flow

1. Start **data ingestion service** (Rust) → pull live train API for chosen corridor (say Delhi–Howrah).
2. Store all train movement events into SurrealDB.
3. Backend API (`/optimize?section=XYZ`) calls Python service with current corridor state.
4. Python solver returns **optimized precedence/crossing schedule**.
5. Backend persists schedule + streams update to UI.
6. Controller can click **"what if"** → triggers solver with modified constraints.

---

5. Real-World Operations Knowledge Mapping

- **Precedence Rules**:
  - Mail/Express > Passenger > Freight.
  - Emergency trains override.
- **Signalling**:

- Absolute Block: only one train in a block section.
- Automatic Signalling: multiple trains if spaced by signals.
- **Timetable Practices**:
  - Sectional running time fixed.
  - Padding time for delays.
- **Regulatory Compliance**:
  - Always safety-first, no two trains in same block.
  - AI outputs must explain reasoning → audit trail.

---

6. End-to-End Prototype Overview

```
Error parsing Mermaid diagram!

Parse error on line 7:
...ontroller Dashboard (React)]  F -->|Use
-----------------------^
Expecting 'SQE', 'DOUBLECIRCLEEND', 'PE', '-)', 'STADIUMEND',
'SUBROUTINEEND', 'PIPE', 'CYLINDEREND', 'DIAMOND_STOP', 'TAGEND', 'TRAPEND',
'INVTRAPEND', 'UNICODE_TEXT', 'TEXT', 'TAGSTART', got 'PS'
```

✅ This workflow is **full-proof for hackathon**:

- Uses **real-time APIs where possible** (fallback synthetic).
- Rust ensures **high-performance backend & data pipelines**.
- Python gives **optimization power**.
- SurrealDB acts as a **unified brain** for historical + real-time train state.
- Supports **auditability, controller overrides, re-optimization** → which matches IR standards.

Would you like me to **expand this into a "Hackathon Implementation Roadmap" (week-by-week)** so you know what to build first and what to keep lightweight for demo vs. later production-level?