THILAGAN INIYAVAN [CB.SC.U4CSE23750]
DHRUSHEEK RISHI MENON [CB.SC.U4CSE23716]
PREETHI KANNAN [CB.SC.U4CSE23738]
SREEPATHY JOSHY [CB.SC.U4CSE23748]
PRANAV KISHAN T Y [CB.SC.U4CSE23437]
KARTHIKEYAN P G [CB.SC.U4CSE23539]

Review 2

EDGE COMPUTING-BASED REAL-TIME SCHEDULING FOR DIGITAL TWIN FLEXIBLE JOB SHOP WITH VARIABLE TIME WINDOW

The team

Preethi Kannan	Developed Simulation
Thilagan Iniyavan	Physical workshop design - machines , architecture diagram
Sreepathy Vadakath Joshy	Physical workshop design - jobs ,UI
Dhrusheek Rishi Menon	MQTT messaging and integration,data visualization
Pranav Kishan T Y	Genetic Algorithm
Karthikeyan P G	Fuzzy Systems

Work We Have Done

1. Designed and implemented the Machine simulations
2. Designed and Implemented Job Simulations
3. Prototype Dynamic Job rescheduling
4. Digital Twin Integrated

CHALLENGES WE FACED!

CHALLENGE 1

Modeling multi-step jobs with machine class constraints

Each job isn't just a single task – it has subtasks that require specific machine to complete the job
e.g A->B->C

CHALLENGE 2

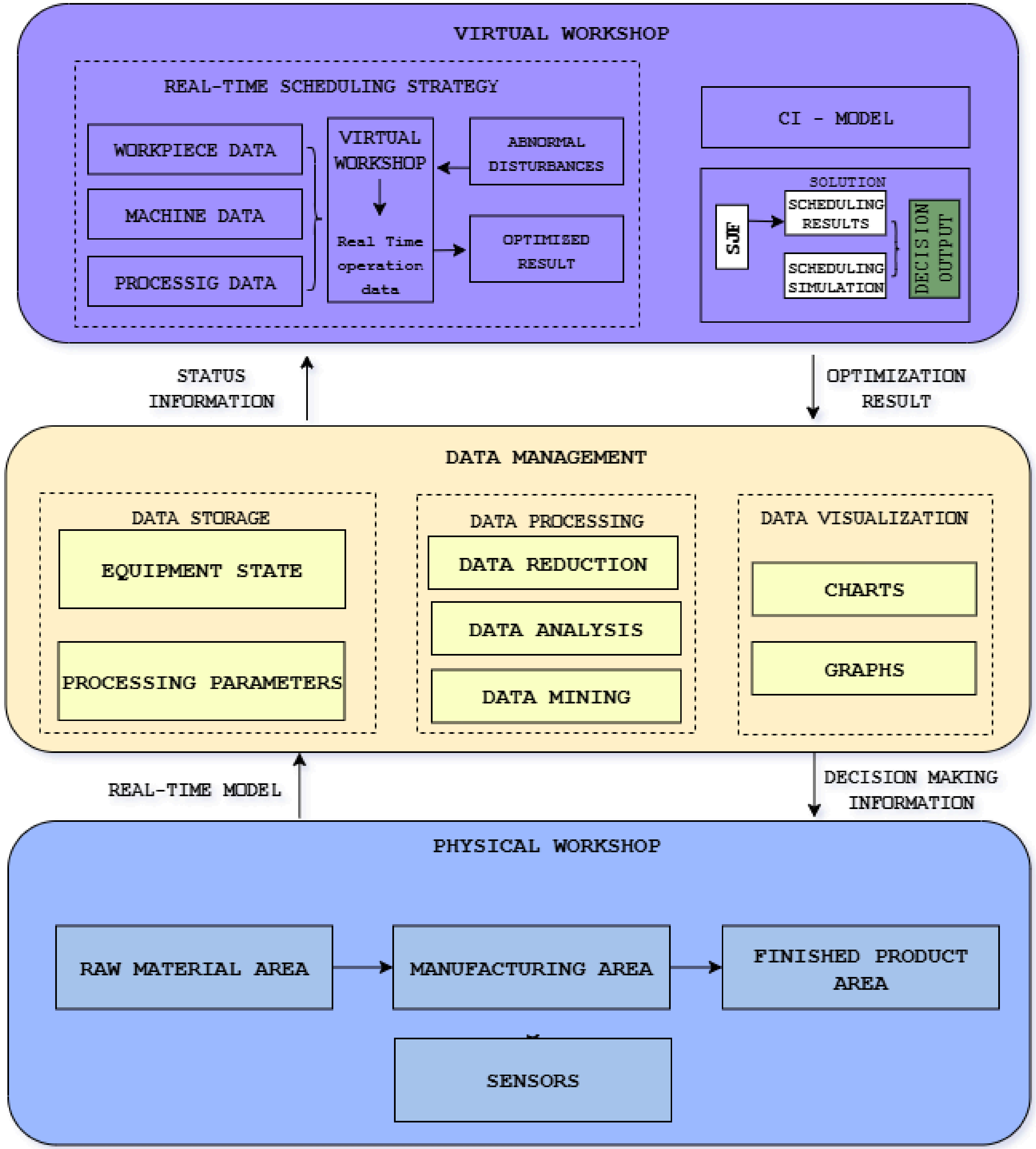
Failure handling and rescheduling realism

Once failed, a machine goes into a timed repair state, but the job it was processing must not be lost.

CHALLENGE 3

Synchronizing MQTT telemetry with job events

- job/status → continuous telemetry per machine (temperature, vibration, job id).
- jobshop/status → discrete events (STARTED, STEP_DONE, FAILED, COMPLETED).



VIRTUAL WORKSHOP
REAL-TIME SCHEDULING STRATEGY
WORKPIECE DATA
MACHINE DATA
PROCESSIG DATA
VIRTUAL WORKSHOP
Real Time operation data
ABNORMAL DISTURBANCES
OPTIMIZED RESULT
CI - MODEL
SOLUTION
SJF
SCHEDULING RESULTS
SCHEDULING SIMULATION
DECISION OUTPUT
STATUS INFORMATION
OPTIMIZATION RESULT
DATA MANAGEMENT
DATA STORAGE
EQUIPMENT STATE
PROCESSING PARAMETERS
DATA PROCESSING
DATA REDUCTION
DATA ANALYSIS
DATA MINING
DATA VISUALIZATION
CHARTS
GRAPHS
REAL-TIME MODEL
DECISION MAKING INFORMATION
PHYSICAL WORKSHOP
RAW MATERIAL AREA
MANUFACTURING AREA
FINISHED PRODUCT AREA
SENSORS

Job Parameters

```
class Job:
    """
    Multi-step job. Each step requires a specific machine CLASS (A/B/C/D).
    Steps are sequential; each has its own remaining ticks.
    """
    job_id: str
    intensity: str
    temp_inc: float
    vib_inc: float

    # steps: list of (required_class, remaining_ticks)
    steps: List[Tuple[str, int]] = field(default_factory=list)
    current_step: int = 0  # index into steps
```

```
    "light":    {"temp_inc": 3.0, "vib_inc": 0.8},
    "moderate": {"temp_inc": 4.5, "vib_inc": 1.2},
    "heavy":    {"temp_inc": 6.5, "vib_inc": 1.8},
    "stress":   {"temp_inc": 8.0, "vib_inc": 2.4},
}

# Route patterns: each entry is a SEQUENCE of machine CLASSES required
ROUTE_PATTERNS = [
    ["A", "B"],
    ["A", "B", "C"],
    ["C", "A"],
    ["B", "D"],
    ["A", "C"],
    ["B", "C"],
    ["A", "A", "B"],
```

Current Parameters:

- job_id: Unique identifier for the job
- intensity: Load level → "light", "moderate", "heavy", "stress"
- temp_inc, vib_inc: Temperature and vibration increments per tick
- steps: List of (machine class, remaining ticks) e.g., [("A", 5), ("B", 3)]
- current_step: Tracks current step in the job workflow

Developing Parameters:

- Power Consumption rate: Rate at which power consumed by the machine when doing a job.
- Temperature Reduction percentage : Percentage of temperature reduced when entering a machine.

Machine Parameters

```python
class Machine:
    """
    Digital Twin Enabled Machine Class for Flexible Job Shop Simulation.

    Attributes:
        class_name (str): Category/type of the machine (e.g., 'A', 'B', 'C').
        machine_id (str): Unique identifier for the machine (e.g., 'A_1').
        temp_base (float): Baseline temperature when idle or after repair.
        temp_threshold (float): Temperature limit for fault detection.
        vib_base (float): Baseline vibration level.
        vib_threshold (float): Vibration limit for fault detection.
        repair_time (int): Number of timesteps required to repair after fault.

    State Variables:
        temperature (float): Current temperature of the machine.
        vibration (float): Current vibration level.
        operational (bool): Operational status (True = working, False = faulty).
        repair_timer (int): Current repair countdown timer.
    """
```

```python
# Simulate random fluctuation/noise in temp and vibration increments
temp_change = job_temp_increment + random.uniform(-1.5, 1.5)
vib_change = job_vib_increment + random.uniform(-1, 1)

self.temperature += temp_change
self.vibration += vib_change

# Check if machine state exceeds fault thresholds
if self.temperature >= self.temp_threshold:
    self.operational = False
elif self.vibration >= self.vib_threshold:
    self.operational = False
```

The Simulation currently does not include accidents or rush orders , instead we manually assign operational flag as False randomly.

The logic for change in temperature , vibrations and the cool down time has to be refined.

Simulation Flow

```python
    # Seed jobs
    for _ in range(seed_jobs):
        self.enqueue_new_job()

# --- MQTT helpers ---
def _on_connect(self, client, userdata, flags, rc):
    print("[MQTT] Connected" if rc == 0 else f"[MQTT] Failed rc={rc}")

def _publish_jobshop_event(self, event_type: str, payload: dict):
    msg = {"type": event_type, **payload}
    self.client.publish(TOPIC_JOBSHOP, json.dumps(msg))

def _publish_job_status(self, machine: Machine):
    self.client.publish(TOPIC_JOB_STATUS, machine.status_json(self.t))

# --- Job flow helpers ---
def enqueue_new_job(self):
    job = Job.make_random()
    heappush(self.class_queues[job.required_class],
             (job.remaining_ticks_on_step, job.job_id, job))

def _enqueue_current_step_front(self, job: Job):
    heappush(self.class_queues[job.required_class],
             (job.remaining_ticks_on_step, job.job_id, job))

def _enqueue_next_step(self, job: Job):
    if not job.done:
        heappush(self.class_queues[job.required_class],
                 (job.remaining_ticks_on_step, job.job_id, job))
```

hard-coded thresholds

SJF

STEP_DONE

- One stage of the job finished

COMPLETED

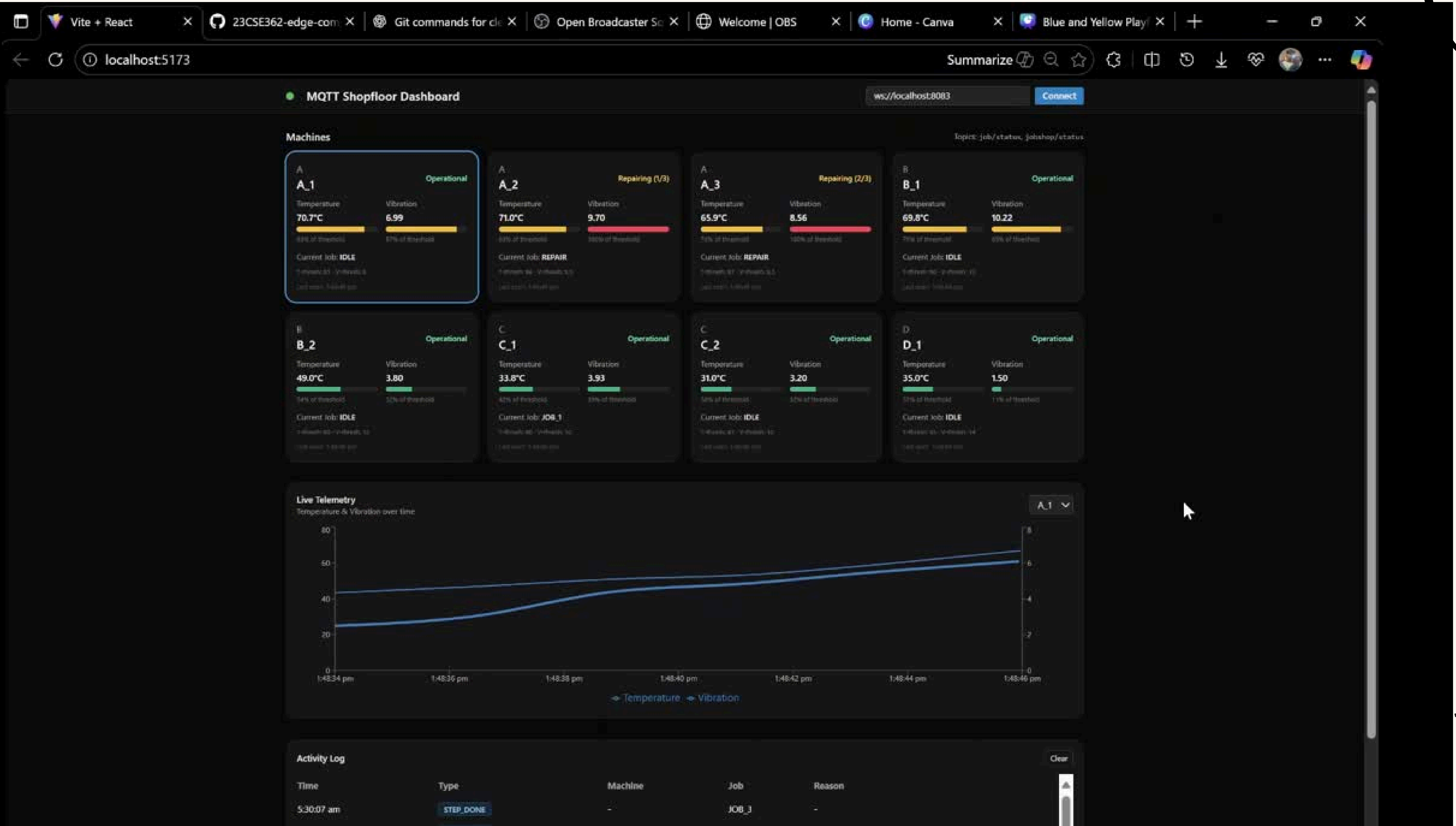
- Entire job finished.

FAILED

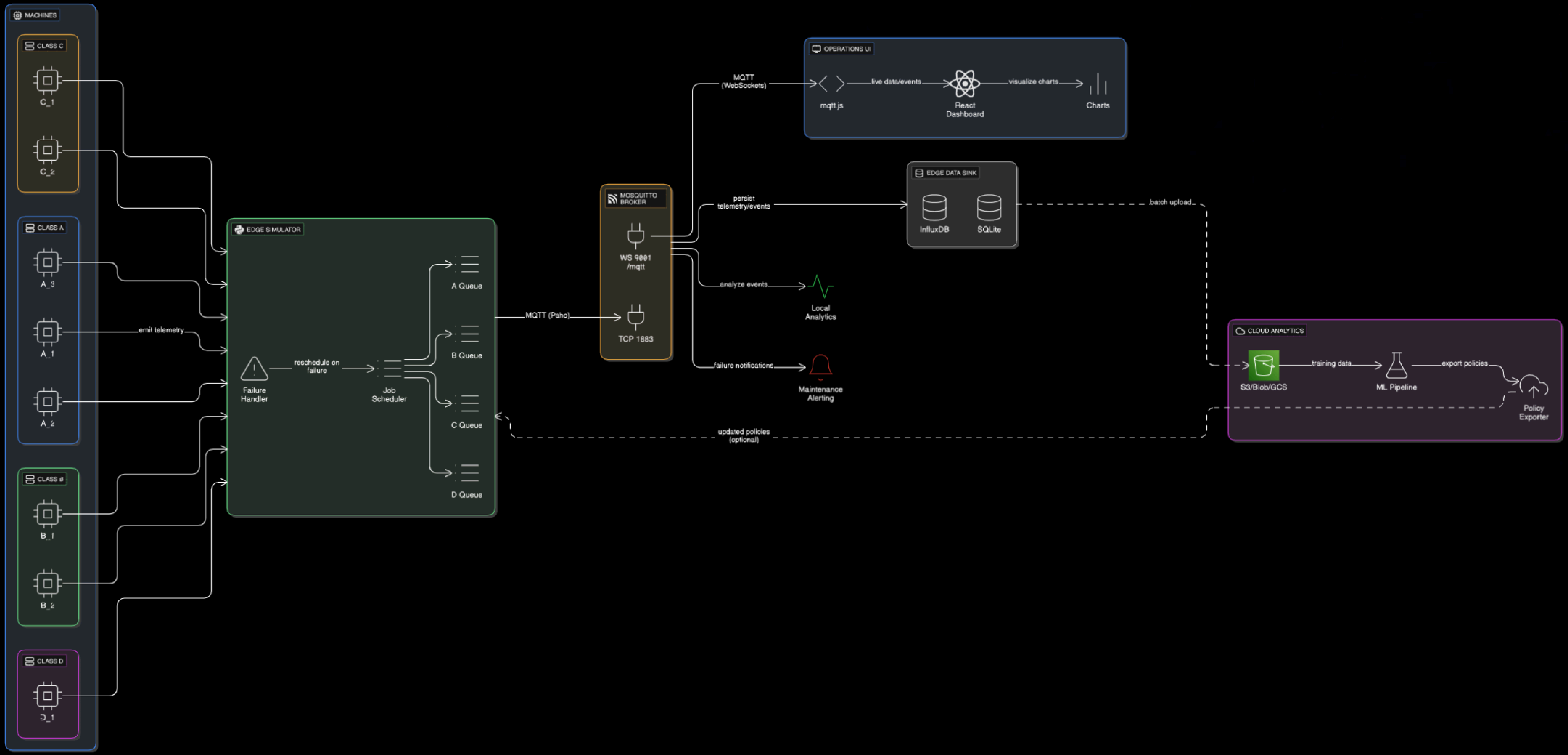
- Re-queue it at the front of its current class queue.

Working Of the Dashboard

React + Vite + MQTT



Working Architechture



```
[13:49:48] Topic: job/status
{
  "class_name": "C",
  "current_job": "REPAIR",
  "machine_id": "C_1",
  "status": "Repairing (3/4)",
  "temp_threshold": 80,
  "temperature": 53.0,
  "timestamp": 38,
  "vib_threshold": 10.0,
  "vibration": 10.36
}
```


MQTT Message Structure

mosquitto-websockets.conf

```
listener 1883
protocol mqtt

listener 8083
protocol websockets

allow_anonymous true
log_type all

```

How is the Frontend Picking up the messages??

Prediction of Machine Failure

Genetic Algorithm (GA)

- Problem: How to tune fuzzy thresholds (low/medium/high)?
- GA process:
  a. Generate population of fuzzy thresholds (chromosomes).
  b. Run simulation → compute fitness.
  c. Select best → crossover → mutate → evolve.
- Fitness: Reward when fuzzy outputs high risk near failure and low risk when safe.
- GA finds optimal fuzzy breakpoints → more accurate and adaptive.

Code Snippet (Fitness Function)

```
if is_about_to_fail:
    TARGET_RISK = 85.0
    error = abs(risk_score - TARGET_RISK)
    self.fitness_score += (100 - error)
else:
    self.fitness_score += (100 - risk_score) / 10
```

Fuzzy Failure Predictor

- Inputs: Temperature and Vibration fuzzified into sets (Low, Normal, Hot / Low, Medium, High).
- GA dynamically optimizes the exact boundaries and peaks of these fuzzy sets to maximize prediction accuracy.
- Example: at 85°C, temp can be 40% normal and 60% hot.
- Rules: We wrote 4 intuitive rules,

```
rules = [
    ctrl.Rule(temp['hot'] | vib['high'], risk['critical']),
    ctrl.Rule(temp['normal'] & vib['medium'], risk['medium']),
    ctrl.Rule(temp['normal'] & vib['low'], risk['low']),
    ctrl.Rule(temp['hot'] & vib['low'], risk['medium'])
]
```

- Output: Rules fire partially, overlap, and create a fuzzy “risk curve.”
- Defuzzification: We use the centroid method → find the balance point of that curve → gives a crisp risk % like 72%.

Results & Takeaway:

- Baseline Fuzzy: Works but depends on manually chosen thresholds.
- GA-Optimized Fuzzy: Learns the best thresholds → higher reliability.
- Performance: Improved precision/recall in predictive alerts.
- Strengths of CI Approach:
  - Human-interpretable (Fuzzy Rules) + Self-tuning (GA).
  - Lightweight → runs at the machine edge, sends MQTT alerts.
  - Handles uncertainty smoothly, interpolates unseen cases.

THANK YOU!