CET Hackathon Questionnaire

CodeReCET 2026 powered by Armada

Question 1:

Title: Real-Time YOLO11 Ops-Safety Challenge

Vertical & Domain: AI, Operational safety

GPU required: No

Business Context:
In industrial environments — such as manufacturing floors, construction sites, and warehouses — Operational Safety is paramount. Automated systems are increasingly used to detect PPE compliance (hard hats, vests), monitor "No-Go" zones, and identify hazardous machine interactions. While YOLO11 Large provides the high precision necessary to avoid false negatives in safety scenarios, deploying high-end GPUs across every corner of a massive facility is cost-prohibitive. To scale these safety "eyes," we must be able to run these complex models on standard industrial PCs or edge gateways that rely solely on 8-core CPUs.

Problem Statement:
The "High-Accuracy, High-Latency" trap prevents the deployment of YOLO11 Large in CPU-only environments. For a safety system to be "realtime" and actionable, it must process video feeds at a minimum of 10 FPS. Currently, a standard YOLO11 Large inference pass on a CPU often falls well below this threshold, leading to lagged detections that compromise worker safety.

Challenge:
Participants must optimize the YOLO11 Large inference pipeline to achieve > 10 FPS on a maximum of 8 CPU cores without significantly degrading detection accuracy.

Core Requirements & Technical Specs:

1. **Model Optimization & Inference Pipeline**

a. **Model Export & Quantization:** Convert the YOLO11 Large model from PyTorch to an optimized inference format. Apply quantization while keeping mAP@0.5 degradation within **≤ 2%** of the FP32 baseline.
b. **CPU-Optimized Runtime:** Leverage CPU-specific acceleration libraries to exploit instruction sets available on modern 8-core industrial processors.
c. **Batch & Tile Processing:** Implement sub-frame tiling or adaptive batching strategies to maximize throughput per core. The pipeline must process a single 1080p frame end-to-end (pre-process → inference → post-process) in **≤ 100ms**.

2. **Pre-Processing & Post-Processing Efficiency**
    a. **Asynchronous Decode-Infer Pipeline:** Decouple frame decoding (via FFmpeg/OpenCV) from inference using a multi-threaded producer-consumer architecture. Frame decode must not block the inference thread, ensuring zero idle cycles on compute cores.
    b. **Smart Frame Sampling:** Implement an intelligent frame-skip or keyframe-priority strategy that maintains detection continuity (e.g., process every 2nd–3rd frame with lightweight object tracking interpolation on skipped frames) to sustain ≥ 10 FPS effective detection rate.
3. **Multi-Stream & Resource Management**
    a. **Concurrent Stream Handling:** The solution must support processing of at least **4 simultaneous camera feeds** on an 8-core CPU, allocating cores efficiently via thread-pool management or process-level isolation.
    b. **Adaptive Quality Control:** Dynamically adjust input resolution (e.g., 1080p → 720p → 480p) based on real-time CPU load, ensuring the 10 FPS floor is never breached even under peak utilization.
    c. **Memory Footprint Constraint:** Total RAM consumption of the inference pipeline (model + buffers + runtime) must not exceed **4 GB** to remain viable on standard industrial PCs.
4. **Safety-Critical Reliability**
    a. **Detection Consistency:** The pipeline must guarantee zero dropped detections for objects present in ≥ 3 consecutive frames (i.e., no "flickering" detections on persistent hazards).
    b. **Alert Latency SLA:** From the moment a safety violation appears in-frame to the moment an alert signal is emitted, the total pipeline latency must be ≤ 300ms.

Question 2:

Title: Ultra-Low Latency Scalable Media Engine

Vertical & Domain: AI, Operational safety (Video Infrastructure & Edge-to-Cloud)

GPU required: No

Business Context:
Focusing on the high-performance plumbing required to deliver mission-critical video data from hazardous zones to safety monitoring hubs. Real-time safety monitoring relies on "Instant Intervention." In a smart factory, if a worker enters a restricted zone, the system must trigger an alert in milliseconds. Most standard streaming solutions suffer from "HLS Lag" (often 10–30 seconds), which is unacceptable for safety-critical applications.

Furthermore, industrial sites often have fluctuating bandwidth. We need a robust, media-agnostic engine that can take any industrial feed (RTSP/RTMP), reshape it on the fly (Transcoding/Scaling), and deliver it to both low-latency web browsers (WebRTC) and massive-scale playback devices (HLS).

Problem Statement:
The challenge is to architect a highly scalable, real-time video broadcasting platform capable of handling high-density industrial feeds with minimal end-to-end latency. The system must act as a "Universal Translator" for video, converting legacy industrial streams into modern, web-ready formats while maintaining horizontal scalability.

Challenge:
Build a containerized streaming engine that ingests RTMP/RTSP and outputs WebRTC (for <500ms latency) and HLS (for scale), featuring a dynamic API driven control plane for real-time resolution and bitrate adjustment.

Core Requirements & Technical Specs:

1. **Ingestion & Transformation:**
    - **Multi-Protocol Ingest:** Support for RTSP (legacy IP cams), RTMP.
    - **CPU-Optimized Transcoding**: Using FFmpeg/Libav with sub-GOP (Group of Pictures) processing to ensure the "Universal Translator" doesn't introduce more than 50ms of processing overhead.

- **Format Forking:** Single ingest stream simultaneously transcoded into WebRTC (UDP-based for <500ms latency) and LL-HLS (HTTP-based for massive browser/mobile scale).

2. **Horizontal Scalability & Orchestration:**
    - **Stateless Media Nodes:** Deploy as Docker containers managed by Kubernetes. Media processing pods are "disposable"—if one fails, the control plane re-routes the RTSP stream to a healthy node in <1s.
    - **SFU Fan-out:** Use a Selective Forwarding Unit (SFU) architecture for WebRTC. This allows one ingested stream to be distributed to hundreds without re-encoding it for each user.
    - **Custom HPA (Auto-scaling):** Scale pods based on Network Throughput and Active Stream Count rather than just CPU, preventing latency spikes during high-load shifts.
    - **Global State Store:** A Redis layer tracks which node is handling which camera feed, allowing the API to shift bitrates or resolutions on the fly across the entire cluster.

Question 3:

Title: Autonomous Drone based Disaster Intelligence for Damage mapping and rapid response

Vertical & Domain: AI, Operational safety

GPU required: Yes

Business Context:
In the immediate aftermath of a natural disaster—be it an earthquake, flood, or wildfire—the primary bottleneck for emergency services is Situational Awareness. Decision-makers at organizations like FEMA or the Red Cross need to know exactly where the most severe damage is located to prioritize search and-rescue teams, medical aid or plan fire fighting measures. Traditional manual assessment is slow, dangerous, and could lead to loss of lives.

Problem Statement:
The "Data Deluge" challenge during disasters means that thousands of satellite and drone images are collected but cannot be processed by human analysts fast enough to be

actionable. The goal is to build an end-to-end AI pipeline that ingests heterogeneous image data and outputs a structured, prioritized damage report, a live safety heat map that guides relief efforts with surgical precision, ensuring that the most vulnerable areas receive help first.

Challenge:
Teams should build an end-to-end pipeline for processing video feeds and outputting an impact heatmap in near real-time.

Core Requirements & Technical Specs:
A system that can take a video source - a file or stream and do the following analysis

- Detects visible signs of impact such as collapsed or heavily damaged buildings, large fire or smoke regions, flooded zones, and high debris concentration.
- Assigns an Impact Score to each grid/zone.
- Outputs a near real-time spatial heat map.
- A web-based visualization dashboard.
- Bonus point for handling multi-source imagery.
- Technical report on any assumptions and scoring logic.

Question 4:

Title: Fair-Queue GPU Load Balancer

Vertical & Domain: AI, Operational safety

GPU required: Yes

Business Context:
Deploying high-end GPUs for AI-powered inference is a significant infrastructure investment. Organizations running real-time workloads — such as multi-stream video analytics — need to extract the maximum possible throughput from a single GPU to justify cost and scale efficiently. However, as workload complexity and stream volume grow, GPUs are often underutilized due to inefficient batching, poor memory management, or unoptimized model pipelines. In safety-critical environments, both performance and stability are non-negotiable. Maximizing GPU inference throughput ensures faster detection, lower latency, higher stream density per node, and better overall return on

hardware investment — all without adding additional compute resources.

Problem Statement:
A single GPU running YOLOv11 Large inference across multiple concurrent video streams is rarely operating at its true hardware ceiling. Suboptimal batch sizes, sequential frame processing, memory bottlenecks, and pipeline inefficiencies leave significant throughput on the table. The core problem is: how do we push a single GPU to its absolute maximum sustainable inference throughput while maintaining model accuracy, system stability, and acceptable latency across all active streams?

Challenge:
Design and implement an optimized single-GPU inference pipeline for video processing with YOLOv11 Large that:

- Maximizes frames processed per second (FPS) to the hardware limit of the GPU
- Handles multiple concurrent input streams without starving any single source
- Minimizes Glass-to-Detection latency (time from frame capture to inference result)
- Maintains system stability under sustained peak load without OOM failures or crashes

Core Requirements & Technical Specs:
Participants must design and validate a solution for real-time image object detection using YOLOv11 Large, achieving a minimum throughput of 100 frame per second on a single GPU constrained to ~16 GB GDDR6 memory, fewer than 3,000 CUDA cores, and fewer than 400 Tensor cores (e.g., NVIDIA T4).

Question 5:

Title: Proximity-based hazard detection

Vertical & Domain: AI, Operational safety

GPU required: Yes

Business Context:
Heavy machinery operations in industrial, agricultural, and construction environments pose significant safety risks when workers operate in close proximity to moving or stationary equipment. Traditional safety measures rely on physical barriers, manual supervision, or wearable devices all of which are costly, restrictive, or dependent on

human compliance. There is a growing need for passive, vision-based safety systems that can monitor worker-machine proximity in real time without requiring any hardware on the worker.

Problem Statement:
Workers operating near heavy machinery such as telehandlers, excavators, and conveyor systems are at risk of injury due to insufficient spatial awareness between the operator/bystander and the machine. Current systems lack the ability to automatically detect and alert when a person enters a hazardous proximity zone around active equipment, particularly in large industrial or unstructured outdoor environments.

Challenge:
Workers operating near heavy machinery, construction vehicles or industrial hazards are at risk of injury when unsafe proximity occurs between a human and such a hazard. Build a system that can automatically process multiple video feeds and build a real-time alert system.

Core Requirements & Technical Specs:

Build a vision-based system that:

1. Takes a list of video files or multiple rtsp urls.
2. Detects humans and heavy machinery.
3. Estimates physical proximity between them in 3D space using only monocular RGB input.
4. Generates alerts when a human enters a hazardous distance threshold.
5. Log the time and feed info (video source file/rtsp url, timestamp).

Dataset Sources (Optional): Industrial datasets with humans and machines should be used.

Question 6:

Title: Visual UI Navigation Agent for Operational Platforms

Vertical & Domain: AI, Atlas AI

GPU required: Yes

ARMADA

Business Context:
Operational platforms such as Atlas are feature-rich but often complex, requiring multiple clicks, contextual awareness, and workflow familiarity. This creates usability barriers for:

- New operators
- Non-technical users
- Users with disabilities (visual, motor, cognitive)
- High-pressure operational environments where speed matters

Current systems rely heavily on manual navigation. Accessibility is often compliance-driven rather than intelligence-driven.

A Visual UI Navigation Agent can act as an intelligent layer on top of the platform — understanding screens, guiding users, executing workflows, and ensuring accessibility by design.

This enables:

- Flattens Learning Curve
- Faster incident resolution
- Improved accessibility compliance
- Higher operational efficiency

Problem Statement:
Design and implement an AI-powered Visual UI Navigation Agent that:

- Understands operational platform screens (via DOM or screenshots).
- Interprets user intent in natural language.
- Executes navigation steps autonomously or semi-autonomously.
- Enhances accessibility for users with visual, motor, or cognitive impairments.
- Maintains safe, auditable, and permission-aware actions.

The system must demonstrate reliable task execution across core workflows without compromising security, data integrity, or user control.

Challenge:

1. **Complex UI Structures**
   Deeply nested components, dynamic rendering, modal overlays.

2. **Ambiguity in Natural Language**
   User prompts like “fix this issue” require contextual reasoning.
3. **Accessibility Gaps**
   Many enterprise apps lack full semantic labeling.
4. **Hallucination Risk**
   Agent must not execute unsafe or invalid actions.
5. **Security & Permissions**
   Agent actions must respect role-based access control.
6. **Latency Constraints**
   Real-time navigation assistance must feel instant.
7. **UI Drift**
   Platform updates may break learned navigation paths.

Core Requirements & Technical Specs:

Input Modalities:

- Visual interface understanding (e.g., screenshot-based analysis using multimodal models).
- Structured UI metadata parsing (e.g., DOM or accessibility tree when available).
- Natural language command input (text-based interaction).
- Optional voice-based interaction.

Accessibility Features:

- Automated accessibility compliance checks aligned with recognized standards.
- Full keyboard-navigation capability.
- Screen-reader-friendly summaries and contextual descriptions.
- Dynamic labeling and explanation of UI elements.
- High-contrast and simplified interface overlay options.
- Voice-command support.
- Guided step-by-step assistance mode for cognitive accessibility.

Safety & Governance:

- Explicit confirmation for high-impact or irreversible actions.
- Role- and permission-aware execution controls.
- Comprehensive action logging and auditability.
- Confidence scoring prior to automated execution.

- Human-in-the-loop override and fallback mechanisms.

Desired Outcome:
A working prototype where:

- A user can type: “How to add new terminal to service line”.
- The agent navigates the UI autonomously.
- It highlights relevant components.
- Provides an accessible summary.
- Offers optional step-by-step guidance.

Demo should show:

- Standard navigation vs. agent-assisted navigation.
- Accessibility enhancement in action.
- Measurable reduction in task completion time.

Question 7:

Title: Build the Brain of a Space Data Center

Vertical & Domain: Edge, Orbital Data Center

GPU required: May need GPU

Hardware: Yes

Business Context:
Picture this — a small computer the size of a shoebox is hurtling around Earth at 28,000 km/h. It's collecting data, crunching numbers, and running AI models nonstop. But it can only phone home for a few minutes every couple of hours when it flies over a ground station. The rest of the time? Completely alone. No cloud. No support team. No retries.

Welcome to orbital edge computing — literally putting data centers in space.

This isn't science fiction. Companies are already launching compute nodes into orbit for Earth observation, climate monitoring, IoT relay, defense, and scientific research. And they all hit the same wall: space has terrible Wi-Fi.

Here's what the node has to deal with every single orbit:
90 minutes of silence — collecting data, running tasks, storing everything safely with nobody watching. Then suddenly a 10-minute window opens where it can talk to Earth. In those 10 minutes it needs to send down the most important stuff first — because once the window closes, the next one might be hours away. Oh, and at any point, a power glitch could hit. A cosmic ray could flip a bit. A rogue process could try to eat all the memory. The node has to handle all of this on its own, with zero human help.

Your job: build the software brain that makes this work.

Problem Statement:

Build a prototype software stack that turns a dumb box in space into a smart, autonomous data machine. Your system solves two connected problems:

Problem A — "What do I send first?"
When a communication window opens, time and bandwidth are limited. Your system must figure out the most valuable data to transmit first. Sending a critical anomaly alert before 500 routine temperature logs could be the difference between catching a wildfire and missing it entirely.

Problem B — "How do I survive alone?"
During the hours between windows, the node must:

- Store all collected data securely (encrypted, tamper-proof).
- Survive power failures without losing or corrupting anything.
- Keep running compute tasks safely in a sandbox.
- Have everything packed and ready to go the instant the next window opens.

Challenge:

Build a mini orbital edge node (on your laptop, a Raspberry Pi, or a cloud VM — you don't need actual rockets) that survives a simulated 4-6 hour mission with multiple orbit cycles.

Your system must handle:

What Space Throws at You	How You Simulate It

Can only talk to Earth briefly	10-min connectivity windows every 90 minutes
Slow satellite links	Bandwidth throttled to realistic speeds
Limited onboard storage	Capped disk space with no cloud backup
Random power failures	Simulated kill signals mid-operation
Buggy payloads	Intentionally broken scripts that try to crash the node

What your system needs to do:

1. Predict windows — Know when the next ground pass is coming and how long it'll last.
2. Prioritize ruthlessly — Score every piece of data by importance, freshness, and size. Critical stuff goes first. Always.
3. Compress smart — Squeeze more value into less bandwidth. Lossless for telemetry, lossy for images if needed.
4. Store securely — Encrypt everything. Verify integrity constantly. If a single bit flips, catch it.
5. Survive crashes — Power dies mid-write? When it comes back, nothing should be lost or corrupted.
6. Run tasks safely — Execute AI/analysis jobs during blackouts in a sandbox. If a task goes rogue, kill it. Don't let it take down the node.
7. Do it all alone — Zero human intervention from start to finish.

Core Requirements & Technical Specs:

- Contact-Window Prediction & Scheduling
    - Simulate future communication windows using orbital parameters (real TLE data from CelesTrak — it's free and public).
    - For each window: estimate how long it lasts and how much data you can push through.
    - Build a transmission schedule before the window opens — no wasting precious seconds deciding what to send.
    - Track how well you used each window (utilization %)
- Smart Data Queue

- Build a priority scoring engine. Every data object gets a score based on:
    - How critical is it? — Mission-flagged items always jump the queue.
    - How fresh is it? — Newer data generally matters more.
    - How big is it? — Small high-value beats large low-value.
    - How long has it been waiting? — Old data gets a priority boost so nothing rots in the queue forever.
- Pre-compress and pre-package everything so transmission starts instantly when the window opens
- Compression: LZ4 or Zstandard for logs/telemetry; optional lossy compression for imagery (JPEG/WebP quality tuning)

- The Vault (Secure Offline Storage)
    - Encrypt all stored data at rest (AES-256 or equivalent).
    - Integrity checking using hash chains or Merkle trees — if corruption happens, detect it immediately.
    - Crash-safe writes using write-ahead logging — if power dies mid-write, existing data stays intact.
    - A manifest that tracks every object: when it was created, its priority score, whether it's been sent, its checksum.
    - When storage gets full: automatically identify lowest-priority already-transmitted data and clean it up.
- Downlink Session Manager
    - Simulate a realistic communication session (throttled bandwidth, latency, possible early cutoff).
    - Send data in strict priority order.
    - Track acknowledgments — know exactly what ground received.
    - If the window closes mid-transfer, mark where you stopped and resume cleanly next time.
    - Update the vault after successful sends (mark as transmitted, eligible for cleanup).
- Compute Sandbox
    - During blackouts, run pre-loaded tasks: Python scripts, containers, or WASM modules (think: image analysis, data aggregation, anomaly detection).
    - Hard resource limits: CPU time cap, memory ceiling, disk quota.
    - Auto-kill anything that exceeds limits.
    - Encrypt and queue results for the next downlink.

- Isolation is non-negotiable — a crashed task must never affect the vault or the core system.

Desired Outcome:

A live demo showing your system surviving a simulated multi-orbit mission:

- Data continuously collected and securely vaulted during blackout periods.
- Compute tasks running in sandboxed environments between windows.
- Smart prioritization picking the highest-value data when a window opens.
- Efficient transmission with real utilization metrics.
- Clean recovery from at least one simulated power failure — zero data loss.
- A dashboard or terminal output showing: window utilization, priority decisions, vault health, task outcomes.
- Impress us: Add a visualization. Show the orbit. Show the queue draining during a contact window. Show a power failure and the vault coming back clean. Make us feel like we're watching mission control.

Dataset Sources:

- Orbital data: CelesTrak TLE data (public, free) — real satellite orbital parameters for realistic window simulation.
- Sample payloads: NASA Earthdata and ESA Copernicus open satellite imagery, synthetic telemetry logs, generated IoT sensor streams.
- Ground stations: SatNOGS network database (public) — real ground station locations for contact geometry.
- Compute tasks: We'll provide starter Python scripts for image thumbnailing, data aggregation, and anomaly flagging — or bring your own.
- Chaos scripts: Power cut simulators, process kill signals, storage quota bombs — we'll provide these or you build your own.

Why This Problem is Worth 36 Hours of Your Life

This isn't a toy problem. Space agencies, satellite startups, and defense organizations are spending millions trying to solve exactly this. If you can build a working prototype in 36 hours, you've just demonstrated a system that has real commercial value in one of the fastest-growing tech sectors on the planet — and above it.

Question 8:

Title: Power-Budget Aware Orbital Edge Job Scheduler & Mission Operations Dashboard

Vertical & Domain: Edge, Orbital Data Center

GPU required: No (optional for advanced analytics)

Hardware: Yes — SBC with battery monitoring and solar/power emulation

Business Context:
Orbital edge nodes operate under strict energy and connectivity constraints. Power generation depends on solar input and is interrupted during eclipse periods. Battery capacity is finite, and every watt-hour consumed by computation reduces energy available for communications, thermal regulation, and flight-critical subsystems. Additionally, connectivity is not continuous, so ground operators need a resilient operational view even when the node is temporarily out of contact.

Problem Statement:
Traditional schedulers assume stable power and connectivity, but orbital nodes must continuously balance compute demands against real-time and predicted energy availability. Without power-aware scheduling, high-energy workloads can deplete batteries below safe thresholds, conflict with communication windows, and starve mission-critical subsystems—making autonomous operation unsafe.

The system must therefore implement a power-budget-aware job scheduler and a mission operations dashboard that:

- Dynamically schedules workloads based on available and predicted energy (solar/eclipses)
- Prevents battery depletion below safe operating thresholds
- Pauses/resumes non-critical workloads as energy margins change
- Preserves operator visibility into energy, workload state, and alerts, even during communication blackouts

Challenge:
Build a working prototype of a mini orbital edge node that ingests real/simulated power telemetry, estimates remaining watt-hour budget, classifies jobs by power profile, schedules workloads using configurable policies, enforces safety thresholds, aligns heavy

jobs with peak solar availability, and provides an offline-capable mission operations dashboard.

Core Requirements & Technical Specs:

1. Power Telemetry Ingestion
   • Read real or simulated voltage/current data
   • Compute battery State of Charge (SoC)
   • Estimate remaining energy budget
   • Model solar input variability
   • Predict eclipse periods (simulated or orbital-based)

2. Job Classification & Power Profiling
   • Tag each job with power class (Low / Medium / High)
   • Estimate watt-hour consumption per job
   • Maintain deferred queue for jobs exceeding budget

3. Scheduling Policy Engine
   Implement pluggable scheduling strategies such as:
   • Priority Queue FIFO (baseline)
   • Earliest Deadline First (EDF)
   • Weighted Fair Queuing (WFQ)
   • Solar-Aware Scheduling (align heavy workloads with peak energy availability)
   The scheduler must:
   • Prevent battery SoC from dropping below defined thresholds
   • Dynamically pause or resume jobs based on energy state

4. Safety Governor
   • Below configurable SoC threshold (e.g., 30%), suspend non-critical jobs
   • Below critical threshold (e.g., 15%), allow only essential system processes
   • Resume deferred jobs when energy margin recovers

5. Mission Operations Dashboard
   Dashboard must include:
   • Battery SoC gauge with trend
   • Solar input vs. load consumption chart
   • Job queue view with priority & power class
   • Next predicted eclipse or high-power window indicator
   • System health metrics (CPU temp, memory, storage)

Offline Capability
• Cache latest telemetry snapshot locally
• Display data even during simulated blackout
• Queue write operations and replay on reconnection
• Indicate age of last contact

Desired Outcome (Optional):
At completion, teams should demonstrate:
• Stable operation across simulated solar and eclipse cycles
• Safe prevention of battery depletion
• Intelligent workload alignment with energy availability
• Clear operational visibility through dashboard
• Autonomous behavior during blackout periods

The final system should resemble a simplified but realistic orbital edge compute control stack.

Question 9:

Title: DTN over LoRa/Wi-Fi: Space Mesh Networking (Store-and-Forward Resilient Links)

Vertical & Domain: Edge, Networking, Orbital / Space Communications

GPU required: No

Hardware: Optional — 3 nodes (laptops/Raspberry Pi/SBCs), LoRa modules (optional), Wi-Fi router/AP (or hotspot)

Business Context:
Deep-space and near-Earth communication links are inherently intermittent. Satellites move in and out of view, ground stations have scheduled passes, and RF links degrade due to interference, obstruction, or power constraints. In these environments, conventional "always-on" IP networking assumptions fail. Operational systems must continue to move data reliably even when links drop for minutes to hours, and they must prioritize critical traffic (telemetry, commands, safety alerts) ahead of bulk payload data.

A Delay-Tolerant Networking (DTN) approach enables store-and-forward delivery: data is persisted locally, forwarded opportunistically when links are available, and retransmitted when disruptions occur — ensuring mission workflows remain robust under real-world connectivity gaps.

Problem Statement:
Build a DTN store-and-forward mesh across three nodes representing (1) satellite/source, (2) relay, and (3) orbital data center/destination. The system must tolerate scheduled and unscheduled link disruptions by buffering and retransmitting data without operator intervention, while enforcing priority classes so critical data is forwarded first.

The solution must demonstrate reliable file delivery despite multiple link breaks, with clear visibility into buffer state and delivery progress.

Challenge:
Deliver a live end-to-end demo in which files reliably arrive at the final destination across a 3-node DTN mesh under repeated link outages. The demo must include real-time observability (buffer occupancy, in-flight bundles/files, delivery acknowledgements, and latency) and show that priority traffic is delivered ahead of lower-priority transfers.

Core Requirements & Technical Specs:

Network Topology & Constraints

- Use 3 nodes: Source (Satellite), Relay, Destination (Orbital Data Center)
- Support LoRa and/or Wi-Fi links (either is acceptable; hybrid is a stretch goal)
- Links must be intentionally disrupted to simulate passes/blackouts

Link Disruption Simulation

- Scheduled link disruptions: toggle links on/off based on a defined timetable
- Disruptions must occur multiple times during the demo
- Link control can be implemented via interface down/up, firewall rules, AP isolation, or a scriptable network emulator

DTN Store-and-Forward Delivery

- Persist outbound data locally when the next hop is unavailable

- Automatically retransmit and forward when the link returns
- Provide delivery acknowledgement to confirm end-to-end arrival
- Ensure no data loss across disruptions (within configured buffer limits)

Priority Traffic Classes

- Implement at least 3 traffic classes (e.g., Critical / Standard / Bulk)
- Critical traffic must preempt lower-priority transfers when bandwidth resumes
- Scheduling must be visible and explainable (queue order and policy)

Buffer Management & Backpressure

- Implement bounded buffers per node (configurable size limits)
- Prevent overflow with graceful backpressure (reject, defer, or throttle senders)
- Expose buffer state and queue depth in real time

Observability & Operations View

- Real-time visibility into:
  - Queue depth / buffer occupancy per node
  - Files/bundles pending, forwarded, delivered, failed
  - Link status (up/down) and disruption schedule
  - End-to-end latency per file/class
- Provide a simple dashboard/UI or CLI status view suitable for live demonstration

36-Hour Deliverable:
A live demonstration showing reliable end-to-end file delivery across the 3-node DTN mesh despite repeated link breaks, including a real-time view of buffering, forwarding, and delivery status.

Success Metrics:

- File delivery ratio under active disruptions (target: 100% for demo set)
- End-to-end latency measured under disruptions (reported per priority class)
- Buffer management correctness: no overflow; graceful backpressure behavior

- Priority enforcement: critical data delivered first after link restoration
- Protocol clarity and quality of technical education (clear explanation + logs/visuals)

Desired Outcome (Optional):
Demonstrate a reusable DTN "space mesh networking" prototype that can be extended to real orbital/ground workflows, including configurable disruption schedules, priority policies, and observability suitable for mission operations.

Question 10:

Title: Offline Voice Assistant with domain knowledge for helping Field Technicians

Vertical & Domain: Edge, Edge AI

GPU required: Yes

Business Context:
Industrial edge environments — oil rigs, mining sites, remote power stations — operate in conditions where cloud connectivity is intermittent or entirely unavailable. Enterprise support operations rely heavily on vendor-supplied technical documentation datasheets, setup guides, wiring diagrams, and product labels to resolve customer queries. These documents contain richly structured yet visually complex content including multi-column layouts, nested tables, annotated images, and embedded diagrams. Field technicians must frequently consult such documents, log incidents, and escalate issues while physically occupied: wearing gloves, operating equipment, or working in confined spaces. Cloud-dependent voice assistants (Alexa, Google Assistant) are non-starters in these environments. There is a clear and growing need for AI assistants that run entirely on local hardware, with no reliance on external APIs or internet access.

Problem Statement:
Technical vendor documents contain heterogeneous content types single and multi-column text, tables, multi-header financial/spec tables, wiring diagrams, product photos, and labels that standard text extraction pipelines fail to parse accurately. When ingested naively into a RAG system, this results in broken context, missed relationships, and poor retrieval quality, directly degrading the accuracy of AI-generated support responses. Existing voice assistants (commercial or open source) assume persistent cloud connectivity for STT (speech-to-text), NLP inference, and TTS (text-to-speech). Industrial

acoustic environments introduce significant background noise (machinery, wind, generators) that degrade standard speech recognition accuracy.

Core Requirements & Technical Specs:
Unified multimodal document ingestion pipeline for high quality RAG systems, Local STT, LLM and TTS models. A working end-to-end voice pipeline — Speech → Intent → Retrieval → Speech — that runs fully offline.

Question 11:

Title: Federated Learning for Edge Anomaly Detection

Vertical & Domain: Edge, Edge AI

GPU required: Yes

Business Context:
Industrial edge deployments span multiple geographies, operators, and regulatory jurisdictions. Each site generates rich sensor telemetry — network health, power consumption, environmental readings — that is valuable for training anomaly detection models. However, raw data from these sites often cannot be centralized due to data sovereignty requirements, competitive sensitivity between operators, or sheer bandwidth constraints in remote locations. Traditional centralized ML is therefore infeasible. Federated learning (FL) offers a principled alternative: sites collaboratively train a shared model by exchanging only model updates — never raw data.

Problem Statement:
Build a federated learning pipeline that trains an anomaly detection model across multiple simulated edge sites, where each site holds a local, non-identically distributed (non-IID) partition of sensor data. The pipeline must demonstrate that a federally trained model matches or approaches centralized performance — without any site ever transmitting raw data outside its own boundary.

Challenge:
Setup an infrastructure to simulate federated learning using any of the cloud providers. Use independent storage volumes per instance. Without data sharing between the client instances, train ML models using a standard dataset (check Dataset sources below).

Core Requirements & Technical Specs:
There should be no data sharing between the client instances. Data splits created for federated learning should account for class imbalances, missing classes, skewed distributions. Have one separate client instance each for testing and validation.

Dataset Sources:
Use something like this Machine Predictive Maintenance Classification (failure type target) or some other industrial anomaly multiclass datasets ONLY.

Question 12:

Title: AI Field Troubleshooting Copilot — Remote Site First Responder

Vertical & Domain: Connectivity, Atlas AI

GPU required: Yes (Not mandatory)

Business Context:
When satellite internet or edge equipment fails at a remote site — an offshore platform, a cargo vessel, a mining camp, a construction site — the person physically present is almost never a network engineer. It's a ship captain, a rig supervisor, a site foreman, or a truck driver. They see the red light blinking but don't know what it means. They call support, wait on hold across timezones, and try to describe hardware state to an engineer verbally. The engineer asks technical questions the field person can't answer. This back-and-forth takes hours or days while the site sits without connectivity — impacting safety systems, data uploads, and business operations.

Problem Statement:
Build an AI-powered field troubleshooting copilot that a non-technical person at a remote site can interact with via chat or voice. The copilot guides them through diagnosis step-by-step using plain language, photos, and simple instructions — identifies the probable issue — and either walks them through the fix or escalates with a complete diagnostic package to remote support, cutting resolution time from hours to minutes.

Challenge:

- The person at the site and the person with expertise are never the same person — this is a fundamental access-to-expertise gap.

- Support calls are high-friction: long hold times, timezone differences, language barriers, difficulty describing hardware state verbally.
- Troubleshooting documentation exists as lengthy technical PDFs that nobody reads during a crisis.
- The same 20 issues account for 80% of field support calls but each call is handled from scratch with no institutional memory.
- Photos sent via email or chat lack context — support engineers receive blurry images and need 10 follow-up questions to understand what they're looking at.
- Many remote sites have limited or no connectivity (the connectivity being down is often the problem itself) requiring solutions that can work with minimal bandwidth.
- Mission-critical use cases (emergency response, military, offshore safety) need connectivity assurance but have no forecasting tool.

Core Requirements & Technical Specs:

- Chat-based interface accepting natural language descriptions of problems ("The internet stopped working", "The box has a red light").
- Photo input capability with visual analysis — identify device type from image, read indicator light states, detect physical issues (loose cables, wrong port connections, visible damage).
- Guided diagnostic decision tree engine for known common issues, with LLM fallback for unexpected situations.
- RAG pipeline over equipment manuals, known issue databases, and historical support resolution data.
- Step-by-step fix instructions with visual aids (diagrams, annotated photos showing which cable, which button, which port).
- Automated escalation ticket generation — packages the entire diagnostic conversation (symptoms, photos, steps attempted, device state assessment) into a structured support ticket.
- Context-aware guidance: if the site's equipment inventory is known, tailor diagnostic steps to the specific hardware deployed.

Desired Outcome:
A working prototype where a non-technical user describes a connectivity problem in plain language (and optionally uploads a photo), receives guided diagnostic questions in simple terms, gets step-by-step resolution instructions, and — if unresolved — generates a complete diagnostic package for remote support handoff.

Dataset Sources:

- Equipment documentation: publicly available Starlink, Cradlepoint, Peplink setup and troubleshooting guides
- Common issue knowledge base: curated from public forums (Reddit r/Starlink, vendor community forums), support documentation
- Sample equipment images: publicly available product photos and indicator light reference charts from vendor documentation
- Test scenarios: manually authored based on common field failure patterns documented in vendor support articles

Question 13:

Title: AI-Generated Daily Operations Briefing — "What Happened, What Matters, What To Do"

Vertical & Domain: Connectivity, Atlas AI

GPU required: Yes (Non Mandatory)

Business Context:
Every fleet operations manager, NOC analyst, and site supervisor starts their day the same way — opening 5-10 different dashboards, scanning for red indicators, checking overnight alerts, reviewing usage trends, looking at ticket queues, and mentally assembling a picture of what needs attention. This morning ritual takes 30-60 minutes and relies entirely on the operator's experience to know what to look for and what to prioritize. If they miss something buried in a dashboard, it becomes an escalation later in the day. This problem exists across every industry that operates distributed infrastructure — satellite, telecom, logistics, energy, maritime — and nobody has solved it well.

Problem Statement:
Build an AI agent that generates a personalized daily operations briefing — a concise, prioritized summary of what happened overnight, what needs attention today, and what's coming up this week. Delivered as a natural language digest that replaces the 30-minute morning dashboard ritual with a 2-minute read. The briefing should surface not just what happened, but patterns across time that point-in-time dashboards miss entirely.

Challenge:

- Operational data lives in multiple independent systems (device health, alerts, data usage, incidents, tickets) with no unified synthesis layer.
- Alert volumes are high (50-200 per day for large fleets) — most are noise, a few are critical, and telling them apart requires domain expertise.
- No prioritization exists across domains: a connectivity alert, a safety incident, and a data pool warning appear in separate systems with no relative ranking.
- Shift handovers are verbal or hastily typed — incoming operators miss critical context from the previous shift.
- Trending problems (slowly degrading performance over days) are invisible in point-in-time dashboards — they only become visible in hindsight.
- Different roles need different views: a fleet manager cares about uptime and cost, a safety officer cares about incidents, an NOC analyst cares about device health — but everyone gets the same raw dashboards.

Core Requirements & Technical Specs:

- Data ingestion from multiple operational domains: device health snapshots, alert/event logs, data usage metrics, safety incident records, open support tickets, scheduled maintenance.
- Pattern detection engine: identify repeated issues ("Terminal X has gone offline 4 times this week"), trending metrics ("Data pool usage accelerating — projected to exceed limit by Thursday"), and correlated events ("3 devices at the same site all degraded simultaneously").
- Priority ranking using LLM reasoning: rank items by business impact, not just alert severity — a warning on a mission-critical asset outranks a critical alert on a test device.
- Natural language briefing generation with structured sections: Needs Immediate Attention, Trending Issues, All Clear Summary, Upcoming This Week
- Role-based personalization: generate different briefing perspectives for different operational roles.
- Comparative context: "Your fleet uptime this week is 99.7% — up from 98.9% last week".

Desired Outcome:

A working prototype where sample multi-domain operational data is ingested, patterns are detected, items are prioritized, and a polished natural language daily briefing is generated — personalized by role and including actionable recommendations, not just status reporting.

Dataset Sources:

- Device health data: simulated fleet health snapshots (asset ID, status, location, timestamp) based on published IoT fleet management patterns.
- Alert/event logs: synthetically generated alert streams with realistic severity distributions and temporal patterns.
- Data usage metrics: simulated daily usage data for pooled data plans following typical satellite consumption curves.
- Safety incidents: sample incident records based on publicly documented industrial safety event categories (OSHA incident types).
- Support tickets: simulated ticket data with status, priority, age, and category fields based on common IT service management patterns.

Question 14:

Title: AI-Powered Anomaly Detection for Satellite Internet Terminal Networks

Vertical & Domain: Connectivity, Atlas AI

GPU required: Yes (Not mandatory)

Business Context:
Satellite internet (like Starlink) is now used to connect thousands of remote sites — ships, oil rigs, construction sites, rural offices. Companies called MSPs (Managed Service Providers) are responsible for keeping these connections healthy across hundreds or thousands of terminals spread across different geographies. The problem is that satellite internet is not like your home Wi-Fi. It changes constantly — satellites are moving overhead, weather affects signal, terminals hand off between satellites every few seconds, and network congestion varies by region and time of day. So performance numbers (speed, latency, packet loss) go up and down all the time, and that is completely normal. This makes it extremely hard for a human operator staring at a monitoring dashboard to answer a simple question: "Is this terminal actually having a problem, or is this just normal satellite behavior?"

ARMADA

Problem Statement:
Build an AI system that monitors satellite internet terminals and intelligently detects when a terminal is genuinely underperforming versus when it is behaving normally for its environment.

The system should answer two questions for every terminal at any given time:

1. Is this terminal performing worse than its own history? (local anomaly — something changed on this specific terminal).
2. Is this terminal performing worse than other terminals in the same region? (helps determine if the issue is local to one terminal or affecting the whole area).

The output should be clear, prioritized, and explainable alerts — not just "threshold crossed" but "Terminal X dropped 40% below its usual throughput, while nearby terminals are fine, suggesting a local obstruction or hardware issue."

Challenge:

- Satellite performance is inherently variable — a terminal's speed can swing 30-50% within an hour and that is normal. Traditional threshold-based alerts ("alert if speed drops below X") generate massive false positives.
- Every terminal has a different "normal" — a terminal on a moving ship behaves differently from one on a rooftop. A single global threshold does not work.
- Operators cannot manually correlate across hundreds of terminals to figure out if an issue is local or regional — they need the system to do this automatically.
- When 50 alerts fire at once, operators don't know which ones matter most. Alerts need to be ranked by actual business impact, not just severity labels.
- Existing monitoring tools show raw metrics but don't explain why something is anomalous or what to do about it.

Core Requirements & Technical Specs:

- **Per-Terminal Baselining:**
    - Learn what "normal" looks like for each individual terminal based on its historical data (last 7-30 days).

  - Account for time-of-day patterns (satellite internet often has peak/off-peak cycles).
  - Metrics to baseline: downlink throughput, uplink throughput, latency, packet drop rate, signal quality, obstruction percentage.
- **Spatial Correlation:**
  - Group terminals by geographic proximity (e.g., using H3 hexagonal grid or simple radius clustering).
  - When a terminal shows anomalous behavior, check if nearby terminals show the same pattern.
  - If yes → likely a regional/network issue (lower priority, nothing the operator can fix).
  - If no → likely a local issue (higher priority, actionable).
- **Anomaly Classification:**
  - Local anomaly: Terminal deviates from its own baseline while neighbors are fine → possible hardware issue, obstruction, misalignment.
  - Regional anomaly: Multiple terminals in the same area degrade together → possible gateway congestion, weather event, network-level issue.
  - Expected variation: Deviation is within learned normal bounds → not an anomaly, suppress alert.
- **Explainable Alerts:**
  - Each alert should include: what metric deviated, by how much, compared to what baseline, whether neighbors are affected, and a suggested cause.
  - Example output: "Terminal T-4521: Downlink throughput dropped to 12 Mbps (usual baseline: 45-60 Mbps at this hour). 8 nearby terminals are unaffected. Possible cause: local obstruction or hardware degradation. Priority: High."
- **Priority Ranking:**
  - Rank alerts by severity of deviation from baseline, duration of anomaly, whether it's local vs regional, and number of affected terminals.

Desired Outcome:

A working prototype that takes terminal performance data, learns per-terminal baselines, detects anomalies, classifies them as local vs regional, and presents a ranked alert dashboard with human-readable explanations for each alert.

Dataset Sources:

- Simulated terminal telemetry: Generate synthetic time-series data for 100-500 terminals with realistic patterns — diurnal throughput cycles, weather-correlated dips, random local failures, and regional degradation events.
- Public Starlink performance data: Ookla Speedtest Intelligence (aggregated satellite ISP performance by region), FCC Broadband Data Collection (satellite provider benchmarks).
- Weather data: OpenWeatherMap API or NOAA — cloud cover, precipitation, storms correlated with terminal locations.
- Geographic data: Random terminal coordinates clustered into 10-20 geographic regions for spatial correlation testing.

Question 15:

Title: Intelligent WAN Path Selection — Cost, Performance & Location Aware Switching.

Vertical & Domain: Connectivity, Atlas AI

GPU required: No (ML models are lightweight; optimization algorithms are CPU-based)

Business Context:
Remote sites today don't rely on a single internet connection. A typical edge site — a vessel, a truck, a construction site, a field hospital — has 2-4 WAN links available simultaneously:

Link Type	Strength	Weakness
Satellite (Starlink)	Available almost anywhere	Costs more per GB, latency varies
Cellular (4G/5G SIM)	Low latency, good in urban areas	Expensive roaming, dead zones in remote areas
Ethernet/Wired	Cheap, fast, reliable	Only available at fixed locations (ports, offices)
Wi-Fi/Mesh	Free, local	Short range, unreliable

Today, SD-WAN routers at these sites use a simple priority list to decide which link to use. or example: Priority 1: Ethernet → Priority 2: Cellular → Priority 3: Satellite.

This is static. It doesn't change based on what is actually happening. So a site might be burning expensive cellular data when a perfectly good satellite link is available, or routing a latency-sensitive video call over satellite when cellular would give better quality. When a truck drives from an urban area (great cellular) to a rural area (no cellular, satellite only), the router waits until cellular completely fails before switching — causing a connectivity gap.

Nobody is optimizing this in real time. Operators set priorities once during setup and forget about them. The result is wasted money, poor user experience, and preventable outages.

Problem Statement:

Build an intelligent WAN path selection engine that continuously evaluates all available WAN links at an edge site and makes the optimal routing decision based on three factors:

1. Cost — Which link is cheapest right now? (satellite data pool almost full = expensive; ethernet available = free).
2. Performance — Which link gives the best quality for the current workload? (video call needs low latency; file upload needs high throughput).
3. Location context — Where is the asset right now and what does history say about link quality here? (cellular great in this port, terrible 20 miles offshore).

The system should recommend or automatically execute WAN switching decisions that minimize cost while meeting performance requirements — and learn from outcomes over time.

Challenge:

- Static priorities waste money: A ship docked in port might have free ethernet available but the router is still using priority-1 satellite because nobody updated the config. Multiply this across 500 ships and the wasted spend is enormous.
- No cost awareness in routing: Current SD-WAN routers don't know that the satellite data pool is 90% consumed and every additional GB is expensive. They treat all links as equal cost.

- Performance varies by location and time: Cellular is excellent in some locations and useless in others. Satellite throughput changes based on congestion and orbital position. There's no memory of what worked well at each location.
- Different applications need different things: A safety camera stream needs guaranteed bandwidth. A firmware update can wait for the cheapest window. Email doesn't care about latency. Current routers don't differentiate.
- Switching too late or too often: Wait too long to switch = connectivity gaps. Switch too aggressively = connection flapping and dropped sessions. Finding the right balance is hard.
- No learning loop: When a switch decision works out well or poorly, nothing is recorded. The same suboptimal decisions repeat forever.

Core Requirements & Technical Specs:

- **Real-Time Link Scoring Engine:**
  - For each available WAN link, compute a composite score based on cost, performance, reliability.
  - Weighted formula: link_score = w1×cost + w2×performance + w3×reliability.
  - Weights adjust based on application type.
- **Location-Aware Intelligence:**
  - Build a location-performance map and pre-load expected quality in known areas.
  - Predict dead zones and switch proactively.
- **Application-Aware Routing:**
  - Classify traffic into real-time, bulk, and background.
  - Route different traffic types over optimal links.
- **Predictive Pre-Switching:**
  - Predict link degradation using trajectory, congestion cycles, data pool usage.
  - Switch early with overlap.
- **Decision Explainability:**
  - Log human-readable reasoning for each switch.

Desired Outcome (Optional):

A working prototype that takes a simulated fleet of multi-WAN edge sites, evaluates link options in real-time using cost + performance + location data, and makes intelligent switching recommendations with clear explanations and estimated cost savings compared to static priority routing.

Dataset Sources (Optional):

- Simulated multi-WAN telemetry: Generate synthetic data for 50-100 edge sites, each with 2-3 WAN links. Include realistic patterns — cellular signal varying with location, satellite throughput with diurnal cycles, ethernet availability at fixed locations only.
- GPS trajectories: Simulated asset movement paths (maritime routes, trucking routes) with known cellular coverage zones and port/dock locations.
- Cost models: Published satellite data plan structures (per-GB pricing tiers, pool-based pricing), cellular roaming rate cards from public carrier pricing pages.
- Cellular coverage maps: OpenCelliD (open-source cell tower database) or public carrier coverage maps for realistic signal modeling by location.
- Application traffic profiles: Simulated traffic mix — 20% real-time video, 30% bulk file transfers, 50% background telemetry — with published QoS requirements per category.