Operation Duloc

What Are You Doing In My Swamp?

University of Florida

Our Proposal

- Minimize surfacing, maximize mission time: event-driven, role-coordinated AUV patrol with secure cloud relay.
    - We built ROS2 multi-sub coordination + AWS IoT bridge + edge-first comms strategy.
- Protect long undersea assets with fewer surface windows and faster incident reporting.
- Challenges faced: Low-bandwidth, intermittent comms; expensive surfacing; need continuous pipeline coverage.
- Scope: 3 subs (Mid-tier relay, SubA, SubB) simulated on ESP32/ROS2; scalable to N subs.
    - Success criteria (MVP): Reliable plan handoffs, event uplink within surface windows, downlink tasking applied onboard.
- How is this a novel approach?
    - Time-sliced role rotation (MID/SUB1/SUB2) to compress surface windows; plans & events exchanged peer-to-peer and only summarized to cloud
- Use Case: Low-bandwidth, Low-range, limited up-time connecting to cloud, and continuous monitoring

Web App UI

- Frontend built with Angular (Typescript, HTML, CSS) and styled with Tailwind CSS
- SSE connection to backend
- Collaborative dashboard with 8 other hackathon teams!
- View on www.swamp-portal.com

Web Server Design

Production:

- Spring serves both API and static content
- Compiled Angular HTML files are moved into Spring's static directory through Dockerfile

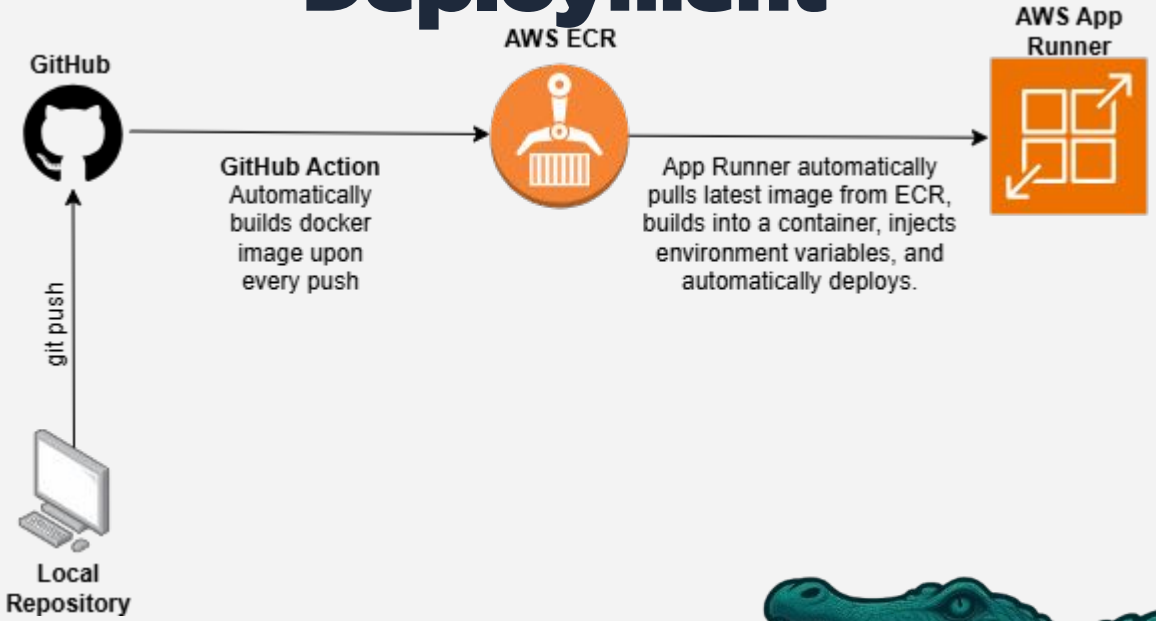
Development:

- Spring serves API on localhost:8080
- Nginx serves Angular on localhost:4200

All:

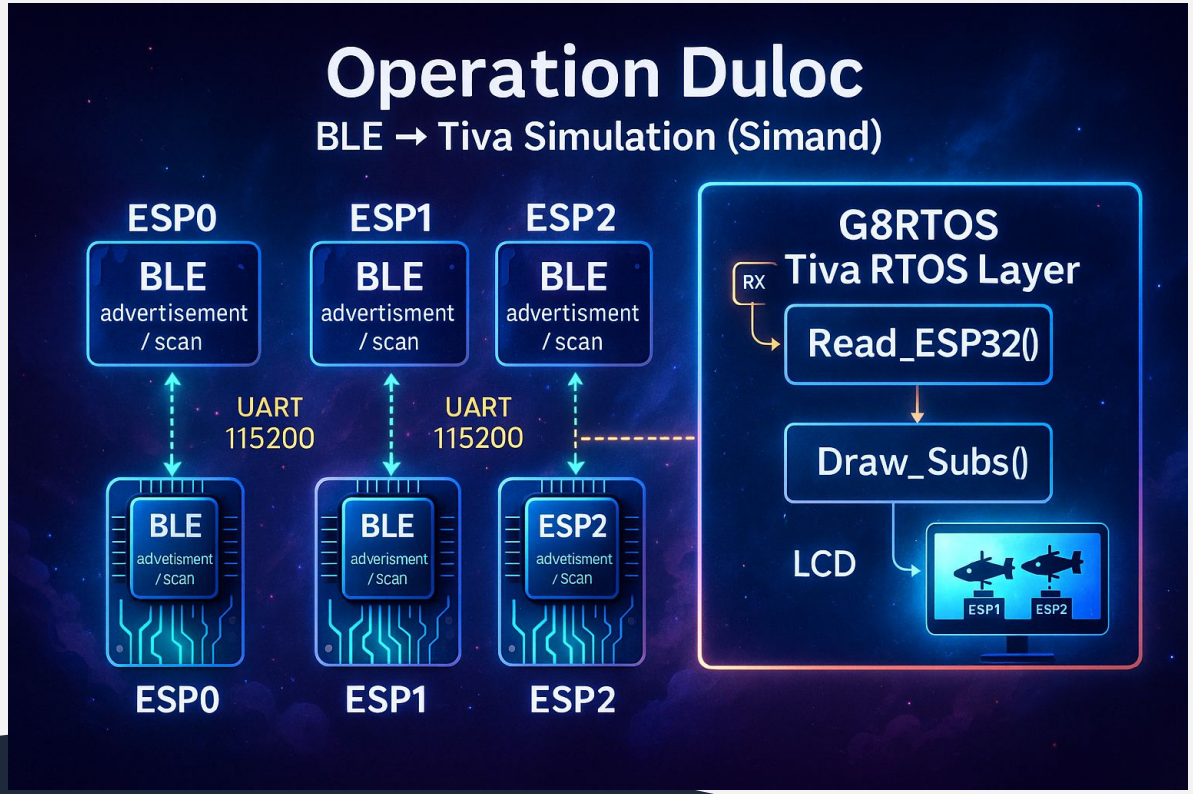
- SSE is used to stream real-time events to all subscribed clients without need for polling (faster and uses much fewer server resources)
- Plan, Report, and Event packets are aggregated to send a single JSON containing all relevant information
- Time-Sensitive data is held in a buffer that removes "expired" data
- The server keeps the aggregated data state
- Deployed on https://www.swamp-portal.com/

Web Server Continuous Deployment



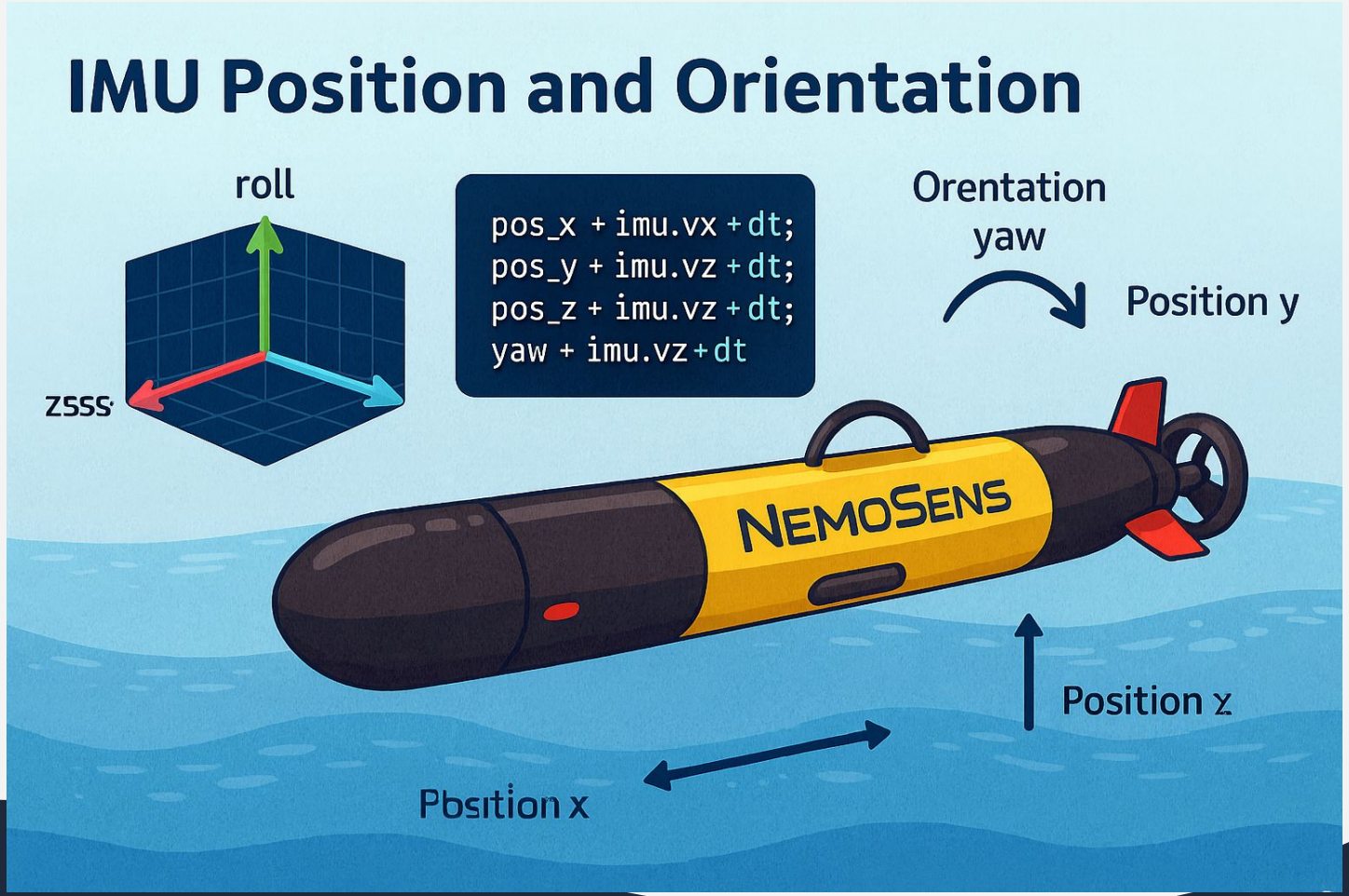
Submarine Firmware

Firmware System Architecture

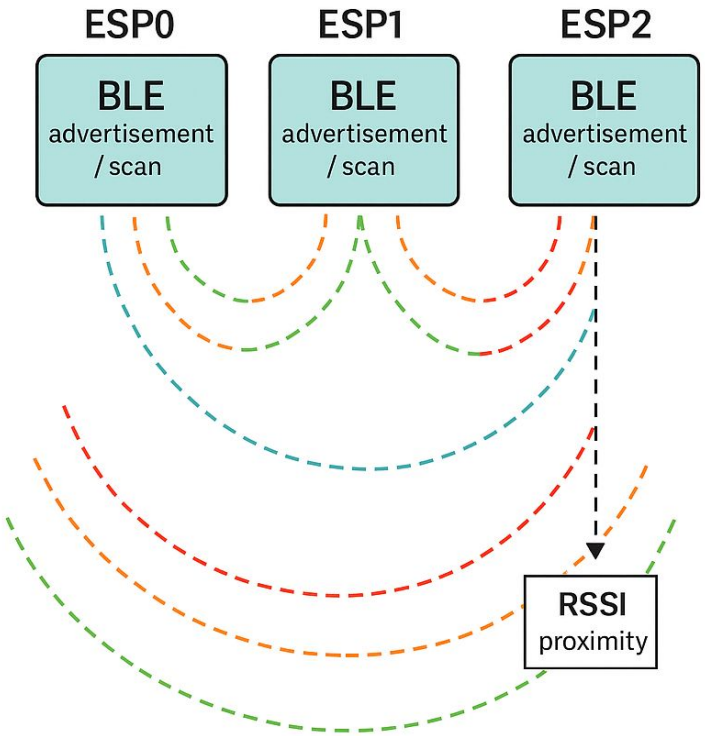


- ESP32 boards each running
  - BLE advertisement/scan
  - UART output (event signaling)
- TIVA Boards
  - Processing IMU data and proximity data
- BeagleBone Black
  - Connected via UART/SPI from TIVA
  - Running ROS node for global swarm data integration

IMU Position and Orientation



Optical Transceiver Emulation via BLE



- Proximity detection simulated using BLE
- RSSI is a proxy for optical range (≥ -60 dBm)
- UART signals sent to Tiva on connect or disconnect

Event	UART signal
ESP0 in range	0
ESP1 in range	1
ESP2 in range	2
ESP0 lost	A
ESP1 lost	B
ESP2 lost	C

ROS2 AUTONOMOUS SUB NODES

- ROS 2 mesh of nodes across vehicle and gateways
- Time-sliced roles (MID_TIER, SUB1, SUB2) with rotating leadership
- Surface relay phases to merge, compress, and uplink knowledge.
  - Plans + movement history unified, deduped, and sanity-checked
  - Identity-centric data model that survives role rotation
  - Deterministic state machine for predictable fleet behavior

ROS2 AUTONOMOUS: Cont.

Innovative:

- Works on small edge hardware, scales with the cloud
- Identity-centric data model keeps history clean across role rotation, vs. role-bound logs that fragment as fleets grow.
- With deterministic time slots, we bound each node's messaging rate, avoiding the $O(N^2)$ message growth typical of ad-hoc peer-to-peer chatter.
- Clean uplink contracts (small, versioned JSON) decouple vehicles from cloud apps; you can add analytics/products without firmware churn. Modular design.
  Edge→Cloud symmetry: same topics, same containers, same tests across dev laptops, simulators, and hardware.

Scalable:

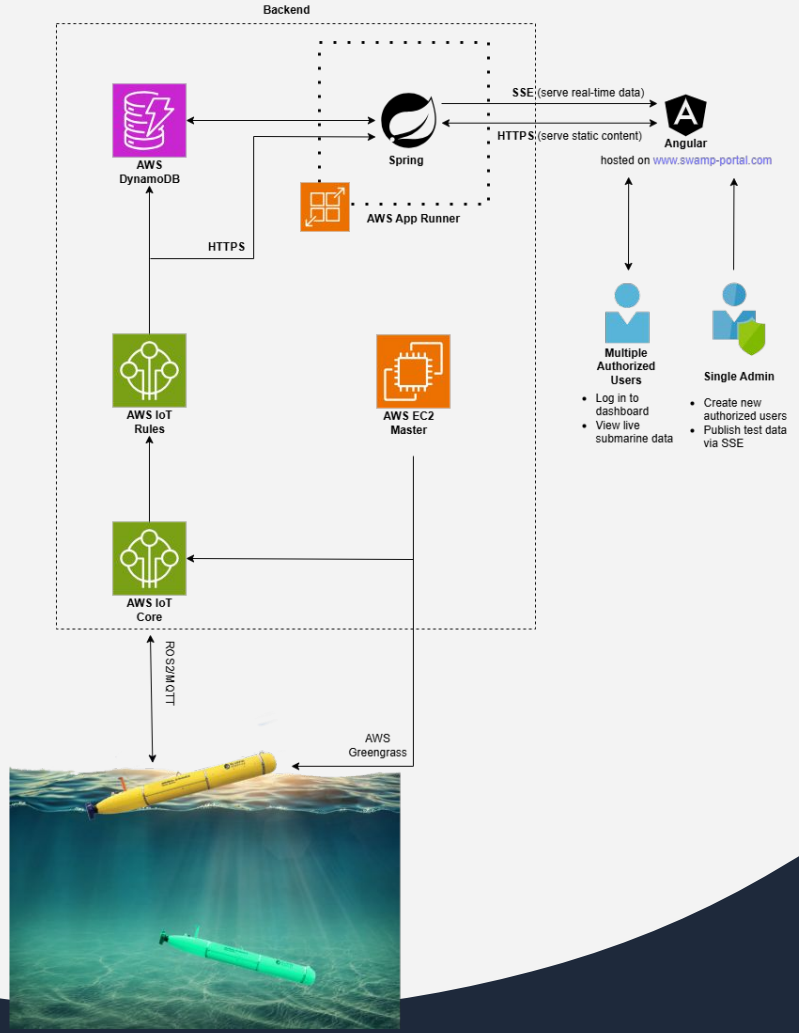
- **MQTT** Bridge Fanout: Reports/Events go to the **AWS IoT Core** -> Rules -> **Dynamo**
  - Other data consumers could be easily configured through IoT
- Tool Chain scalability:
  - Containerized builds (Docker) to produce multi-arch images (arm64/amd64) for Pi, Jetson, and sim hosts.
  - Over the air updates - Use of **AWS IoT Greengrass** for staging updates and version control.

AWS Services

AWS Architecture Connections

Masterlist AWS used:

- Greengrass: Fleet Management
- IoT Core: Network Messaging
- DynamoDB: Persistence
- IoT Rules: Message Piping
- EC2: Scheduling Master Paths
- ECR: Deployment docker Images
- App Runner: Server bring up
- CloudWatch Error logging



DynamoDB

Serverless, elastic data plane (Amazon DynamoDB): absorbs high-frequency IoT telemetry via **AWS IoT Rules** (no ingestion code), auto-scales with mission load and device count.

Flexible, schema-on-read: no rigid tables—supports evolving ROS/control payloads (snapshot, downlink, events); IoT Rules parse fields (timestamps, IDs, metrics) into queryable attributes.

Reliable persistence + analytics path: per-mission composite keys (e.g., report_id + sort key) for ordered logs; **TTL** prunes time-sensitive data; continuous **S3 backup/replication** enables historical analysis and DR.

Real-time + historical access: same tables power live dashboards and retrospective review; PoC validates write throughput/latency and alignment with the proposed design.

Security & interfaces: fine-grained IAM on tables/streams, clear API boundaries (IoT → Rules → DynamoDB → S3), partner-friendly integration using AWS standards.

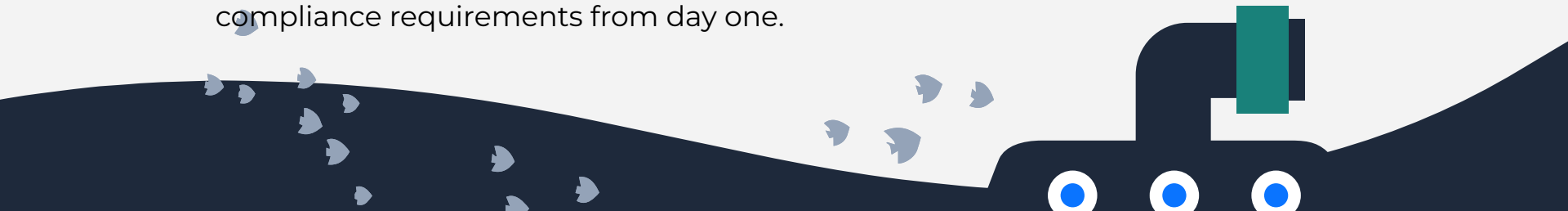
IoT Core

- **Standard MQTT pub/sub via AWS IoT Core:** low-latency, bidirectional comms between edge devices and Mission Control; logical channels like uuv/relay/snapshot, uuv/relay/downlink, uuv/relay/events keep streams clean and discoverable.

- **Centralized, managed device hub:** all ROS UUVs and control nodes connect through IoT Core's secure broker—decoupling hardware from backend consumers (EC2, web app, DynamoDB) and enabling horizontal scale without re-plumbing.

- **Rules-driven routing (no custom listeners):** IoT Rules fire automatically on new messages to persist to DynamoDB, fan-out to SSE, or archive to S3—simple, declarative dataflows that match our PoC design.

- **Security baked in:** TLS mutual auth with X.509 certs, per-device identity in IoT Registry, and fine-grained IAM policies on topics and actions—meeting scalability and compliance requirements from day one.



IoT Rules

3 Major Publishers

EC2 Swamp Control → uuv/relay/downlink (new plans are sent out from server to sub)
ROS Subsystem → uuv/relay/snapshot (historical data is sent from sub IoT core server)
→ uuv/relay/event (any significant events that occurred are streamed separately)

Dataflow Destinations

1. Webserver (via HTTP rule forwarding) for display
2. DynamoDB for logging

Overall

- Minimal Custom Backend
- Centralized (in AWS) data logging
- Real-time dashboard updates

Benefits

- **Decoupled design:** devices don't need to know about databases or APIs
- **Low-latency routing:** actions occur server-side almost instantly
- **Unified process layer:** consistent rule logic across devices (better for scalability)
- **Scalable:** works for thousands of devices publishing to a topic in parallel
- **Support Error Logging:** Errors on action call are redirected to CloudWatch event logs

EC2

Acts as Mission Coordinator

- Runs "Swamp_control_v3" instance that subscribes to snapshot/event stream and publishes new plans to the ROS UUVs
- Manages state and command scheduling
- Aggregates and interprets incoming telemetry
- Dispatches operational plans to IoT devices
- Runs automatic logic and route planning
- Communicates over MQTT5

System Integration

- **Clear system boundaries and data flow:** ROS UUVs and control nodes publish/subscribe over MQTT to AWS IoT Core, IoT Rules route messages to DynamoDB/S3, the Spring backend serves snapshots and a live SSE stream, and the Angular UI consumes SSE/REST for real-time and historical views.

- **Well-defined, documented interfaces:** MQTT topic map (e.g., uuv/relay/snapshot, uuv/relay/downlink, uuv/relay/events), versioned JSON envelopes for Plan/Report/Event, and REST/SSE contracts published via OpenAPI with sample requests and expected responses.

- **Modular, swappable components:** Edge, Broker/Rules, Storage, Backend, and UI are independently deployable; each layer can be upgraded or replaced (e.g., different UI or storage) without breaking the others.

- **Easy partner integration on open standards:** Partners can subscribe read-only via MQTT or SSE and query via REST using **MQTT 3.1.1, HTTP/SSE, and REST+JSON**, with per-team namespaces and filters plus a quickstart (cert provisioning, topic map, Postman collection, IaC samples).

- **Strong security end-to-end:** Devices authenticate to IoT Core with TLS mutual auth (X.509), users/apps authenticate to the portal and enforce RBAC at the backend, and IAM least-privilege policies guard DynamoDB, S3, and Rules actions.

Cost and Performance

- Low-Cost, Scalable Design: Modular hardware and firmware minimize per-unit cost—additional subs only require flashing the same code.
- Efficient Communication: AWS Greengrass and ROS 2 ensure reliable data transfer without high-bandwidth infrastructure costs.
- Performance Tradeoffs: Prioritize real-time data and fault tolerance over minor latency to balance network load and uptime.
- Optimization Strategies: Edge processing on subs reduces cloud computation and bandwidth usage.
- Cost-Effectiveness: Uses off-the-shelf components, open-source frameworks, and scalable cloud services for minimal expansion cost.

Lessons/Improvements

- ROS code node routine made for arbitrary number of subs.
- Certain ratio of relay subs to patrol subs for most efficient communication. More Patrol subs to be added to ours.

#1 Lesson - Draw the system diagram first thing. Establish interfaces!

Technical Debt and Cost

- Web server holds a state and it is impossible to run multiple instances. Fix would include moving state into a database, however we were time restrained.
    - Running the only instance in a server was faster as there were not database calls.
- Not well-defined interfaces/endpoints not well established.
    - API Keys switch up messed up endpoint referenced in code.
- Poorly defined communication protocols.
    - Our JSON formats were very finicky, took fine tuning and revisions.

Designed system with cost-effective subs and reducing surfacing time lengthens mission time/ productive time, less wasted time for surfacing/charging.

Constraints

- Degree of Emulation is constrained to limited hardware which cannot mimic conditions of the real life situation.
    - Using Bluetooth not optical communication
- Assumptions:
    - Assume surface communication.
    - Assume subs will make deadlines/ waypoints at certain times.
    - Assume sub senses magnetic field accurately for monitoring.

Big Wins

- Real hardware emulation of the distributed system
- Built AWS IoT Greengrass for 32-bit ARM
- Ported ROS2 Humble to 32-bit ARM
- Emulated low bandwidth communication protocol and
- Coordinated sending/receiving reports and information using Iot Core and MQTT5.
- Full AWS stack!
- Autonomously interacting node exchanging information in time efficient communication scheme
- Firmware is fully functional, scalable, and deployed to all available devices right now

Scalability

- Modular design supports any number of submarines.
- AWS integration and the UI/backend scale to track large fleets with continuous monitoring.
- Interfaces handle high data loads over long durations.
  Firmware is fully scalable—add new devices by simply uploading the code.

powered by aws

Thank You!