VITALS

Visually Intelligent Tactical Agent for Locating Survivors

Team Members

Justin Gamboa Computer Science	Brendan Rodriguez Computer Engineering	Alex Gershfeld Computer Science
Giorgio Torregrosa Computer Science	Stephen An Computer Engineering	David Jackson Computer Science

Sponsors

Joseph Rivera Lockheed Martin	Rick Leinecker University of Central Florida

Table of Contents

Overview

The following document serves as a comprehensive overview for the VITALS 2025-2026 senior design project sponsored by Lockheed Martin. Our team provides insight on the state of the project as established by the previous senior design team, research for foundational understanding of the tools and technologies used in this project, a design plan moving forward into the next stage of development, and an outline for our team structure and motivations.

Since this design document was created before our team's implementation began, certain features detailed herein may manifest in different forms than originally planned, however, the core functionality and operational objectives will remain consistent through development. For example, our graphical user interface may see several iterations due to feedback received while testing. Any such deviations will be well documented along with records justifying the changes made. Therefore, we intend for this document to serve as the “what” and “why”, while the “how” will be continuously optimized throughout development.

Executive Summary

VITALS stands for **V**isually **I**ntelligent **T**actical **A**ssistant for **L**ocating **S**urvivors. We intend to further the production of an all encompassing hardware and software solution to aid search and rescue operations in locating survivors using drone technology and leveraging AI to enhance speed and expand the reach of individual operators. Lockheed Martin is investing heavily into new drone systems and this project offers an opportunity to advance the field by providing a framework for this perceived software solution. Our software solution can best be explained by dissecting the acronym VITALS into its core components.

Visually Intelligent

Our software will be able to leverage advancements in object detection and multi modal models like LLaVA to identify points of interest in drone footage that will aid our system in search and rescue. We will run detection on key frames and produce captions with LLaVA to help the operator find poi’s without having to monitor every camera from all drones. Our system will alert the operator of confident detections and ask them to make a decision on how to handle

the new information. Additionally, our system will be utilizing a multi-agent framework that includes Ollama to allow near autonomous mission execution based on operator input. We will be able to define points of interest, create job queues for drones, adapt to mission updates, leverage open source data bases like openstreetmap to inform path planning for efficient searching, and send MAVLink commands to drones without needing input from the operator other than a mission statement like “investigate the house”.

Tactical Assistant

Our system will be installed via a docker container on a full hardware solution that allows for tactical field deployment at a moment's notice. All that will be required from the operator is activation of the drones and activation of the ground control station which will load Mission Planner, a fully furnished ground control software, and VITALS, our proprietary multi-agent software which relies on telemetry data from Mission Planner. The operator will establish a connection between the drones and mission planner then use VITALS so define mission objectives and context. A full description of this hardware solution can be found later in this document.

Locating Survivors

Our all encompassing solution to autonomous SAR is intended for use in locating survivors and assisting medical teams in reaching those in need. We understand how drone swarm technology can be repurposed for other means but our system is designed and prompted to be of service to humans and help those who need it. We prompt our agents with this knowledge to ensure mission commands are issued with this intent.

Conclusion

Current project status is ready for implementation of a new multi-agent framework. The code base as it stands is a proof of concept with GUI, path planning, vision, dispatching, and LLM modules ready for further development and documentation. There is a lot of work to be done and we expect to hand off this project with live demonstrations of drone flight from LLM prompts by the beginning of April.

The Problem

Search and rescue operations are critical, time sensitive events that usually rely on ground and air teams working to locate survivors. These methods are inherently resource intensive and prone to human error. During the minute to minute execution of rescue tasks, personnel may unintentionally take actions which cost valuable time. Human operators experience cognitive fatigue during extended operations leading to mistakes and risking lives. Drones provide a unique advantage for aerial surveillance over large difficult to access terrain. However, there is an inherent bottleneck in commanding a swarm of drones for search and rescue operations; one drone needs one operator meaning that a swarm of drones would require several personnel available on scene. This challenge presents an opportunity to leverage artificially intelligent agents for autonomously managing a swarm of drones over a large search area while only requiring one operator providing mission context statements to an intelligent system. VITALS is designed to meet this need.

Mission Statement

To design, develop, and integrate software and hardware Search & Rescue (SAR) solutions utilizing Unmanned Aerial Vehicles (UAV) together with an energy efficient generative AI computer which serves as a SAR Ground Control Station for the Lockheed Martin Aerospace and Defense Company.

The software solution will be deployed onto an NVIDIA Orin Jetson Nano offering unparalleled performance for AI and robotics tasks. Its energy efficient design delivers long lasting performance for portable and embedded systems. Our agentic software solution leverages the autonomous operation capabilities of Mission Planner to identify victims from drone swarm cameras using computer vision and controlling the drone swarm flight paths with waypoints sent via MAVLink packets through Mission Planner. The system will be applied in real world scenarios for search and rescue and disaster response missions, where drones can assist in locating survivors and providing real time data.

Project History

The previous VITALS team, led by James Trent, was able to lay out design plans for our software solution. They accomplished several milestones which include:

- Developing a YOLO model with roboflow for object detection.
- Initializing infrastructure to use Ollama as a reasoning agent.
- Developing a GUI with an LLM chatbox for user commands.
- Implementing a grid system and path planning algorithm.
- Initializing LangGraph infrastructure for agent communication
- Established a TCP connection with Mission Planner for simulation and testing

Upon review of the code base we have determined which aspects of these milestones were implemented and what is lacking or needing further development. All of this and our plan for future implementation are detailed in this document. Additionally, there was no documentation of how to repeat the simulation process with Mission Planner, only a brief video demonstration that did not explain in detail how to replicate the demo. Our team has created a demo video which details the steps to launching VITALS with Mission Planner and a written step-by-step process found in this document to Running VITALS.

The VITALS software repository has many unnecessary files which saturates the code base and hinders development. We have confirmed this by contacting members of the previous team who told us that files outside of the app directory were “useless”. The graphical user interface, retrieval of openstreetmap data, selection of way points, and considerable amount of working software aspects of the project are impressive nonetheless and our computer science team has dedicated a sprint to dissecting the repo and producing internal documentation for a conceptual understanding of the current codebase. VITALS has not yet been tested on real drones; the previous team only achieved a limited simulated mission. Our goal is to continue the work laid out by James Trent’s team and achieve real world flight via a hardware solution. The software research section of this document details our overview of what they accomplished.

Minimum Viable Product

Outlined below is a detailed description of our minimum viable product for this semester. The success criteria section explains the exact expectations for the project as a whole while the component sections describe our expectations for hardware and software.

Success Criteria

The team has been in constant communication with our sponsor from Lockheed Martin, Joseph Rivera. With his help we have defined clear success criteria for our minimum viable product. The VITALS software must be deployed on our hardware solution and demonstrate the ability to send MAVLink commands to a physical drone via chat bot communications. Here are the minimum definitions of integration for the VITALS software.

1. Hardware Integration: The Jetson Nano-based ground control station must successfully establish stable communication with a target drone and send MAVLink commands.
2. Software Integration: The VITALS software must generate valid MAVLink mission commands from user inputs and successfully upload these commands to the drone's flight controller via Mission Planner.

This MVP definition intentionally focuses on proving the end to end integration of hardware, software, and communication protocols rather than requiring fully autonomous AI decision making. While the long term vision includes sophisticated multi-agent reasoning and autonomous mission adaptation, the MVP establishes the fundamental infrastructure upon which these advanced capabilities can be iteratively developed. Success at this stage validates the technical feasibility of the architecture and provides a working platform for incremental enhancement toward full autonomy in subsequent development phases.

Hardware Component

We aim to create a ground control station that will be housed in a portable, weather resistant laptop style enclosure designed for field operations in challenging environments. At its core, the system features an NVIDIA Jetson Orin as the primary compute module, selected for its balance of AI inference capabilities, power efficiency and thermal management suitable for embedded applications. The station includes an integrated monitor for real time mission

visualization and telemetry monitoring, along with a built-in keyboard for operator input and mission parameter adjustment. To ensure operational independence from fixed infrastructures the system incorporates a rechargeable battery supply capable of sustaining multi hour field operations, eliminating reliance on external power sources during remote SAR deployments. The entire package is designed to be transportable by a single operator and deployable within minutes of arriving at an incident site.

Software Component

The software component represents the intelligent core of VITALS, integrating multiple subsystems into a cohesive autonomous mission architecture. At the foundational level, the system leverages Mission Planner as the established ground control interface, providing telemetry visualization, flight parameter configurations, and manual override capabilities that SAR operators are already familiar with. Building upon this foundation, VITALS introduces an AI driven mission generation layer that accepts high level mission objectives and translates them into executable MAVLink commands. The translation process is powered by a proposed multi agent architecture where Agent C (Coordinator and Reasoner) synthesizes mission intents, Agent D (Mapping and Planning) converts these intents into optimized flight paths, and Agent A (Vision and Perception) provides real time object detection feedback to adapt missions dynamically, both uploading autonomous mission waypoints and receiving live telemetry for adaptive replanning.

Operator Perspective

The following is a synthetic story developed to help readers conceptually understand the use of VITALS and how it can provide value to search and rescue operations. No actors in this story are real and this story does not represent a possible testing scenario for development. Our software team is not focused on accuracy of detections per the MVP and we do not intend to sell this system to any clients.

Lost Hiker Story

A 64-year-old solo hiker has been reported missing after failing to return from a day hike in the Osceola National Forest. He was last seen wearing a blue jacket and red hat near the Florida national scenic trail turkey run trailhead at 12 am. His family contacted authorities to help locate him worried that something may have happened. At 2 pm the local sheriff's office activates search and rescue operations to the trailhead in an effort to locate the lost hiker. Luckily for this sheriff's office, the city invested money into purchasing the VITALS ground control station and 3 drones mounted with hd cameras and pixhawk auto pilot flight controllers. Deputy Donovan Baggett has dedicated time to learning how to operate with these systems and is deployed at the site to initiate autonomous search and rescue.

Within minutes of arriving, Deputy Baggett is able to open the VITALS laptop case and open the mission planner along with the VITALS software. He establishes a connection with the drones via mission planner and begins laying out the mission objective. First the deputy defines a search area using VITALS draw polygon feature on the interactive map. That polygon is stored and used in tandem with the openstreetmap database to create a grid with key features like buildings and water identified to aid a drone path planning algorithm. Deputy Baggett selects “Start Mission” and the drones begin to take off following predetermined flight paths from the path planning agent. As the drones begin sweeping the defined search area, they transmit live video back to VITALS. A yolov11 model trained on aerial images runs inference on these frames. If the model returns a human classification detection with confidence level above a certain threshold, a LLava model will be called to caption the image and the operator will receive a pop up informing them that a detection has been made. From this point the Deputy determines

if the detection image is worth investigating further. Deputy Baggett gets a pop up with the caption reading “forest terrain with red hat on the tree”. He determines that it's worth investigating this detection and types into the VITALS software LLM interface “send drone 2 to investigate the detection point”. This message is sent the local Ollama model which reads the message and coordinates new MAVLink commands based on mission context stored in a chroma database. These MAVLink commands are then sent to mission planners which transmit back to the drones on board autopilot which will then move to the detection location and circle the area in an attempt to find more detections.

As drone 2 diverts from its search pattern and heads toward the detection point, Deputy Baggett watches the live video feed and notices the lost hiker laying down. He immediately requests the VITALS software for the coordinates of the detection and writes them down. He ends the mission and all the drones are summoned back to their point of origin via an auto pilot command. The ground team heads to the location and saves the hiker.

The following diagram is a general purpose non technical outline of how an operator can interact with VITALS.

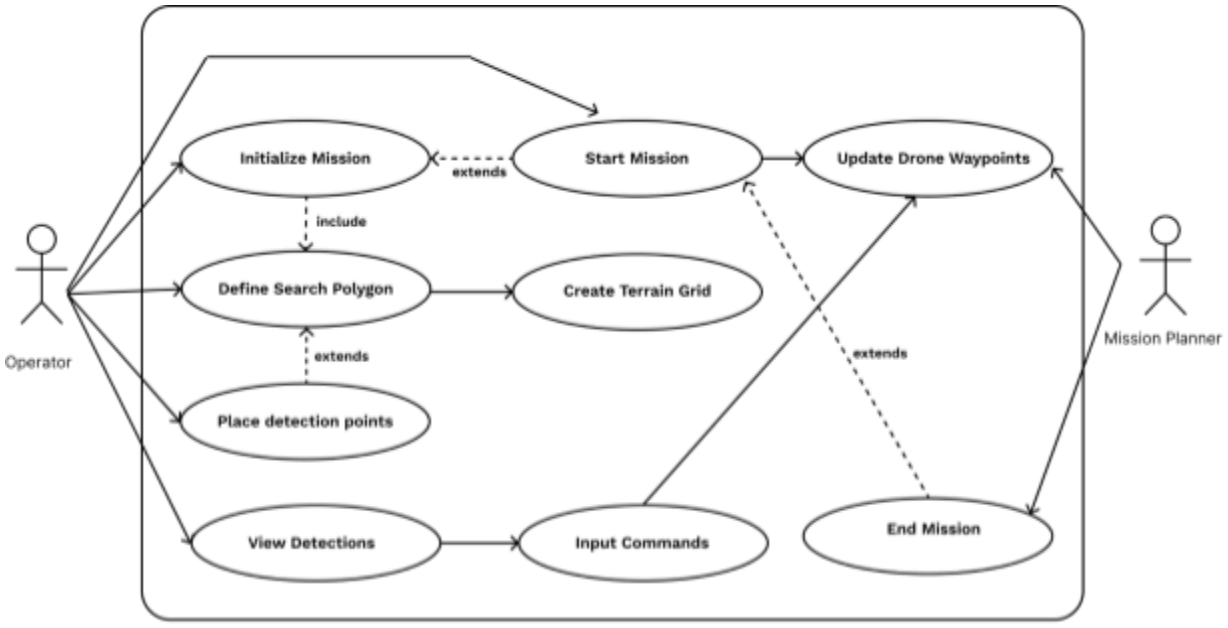


Figure 0.1: Use Case Diagram For VITALS (Non - Technical)

Research

As previously mentioned, we are the second team to work on VITALS. The previous team was only focused on software development and wrote about prospects to integrate into hardware. We spent several weeks reviewing the code base and researching the tools used during development. Our computer engineers have dedicated a lot of time researching how we can integrate our software into the described hardware solution. The following section will discuss this research in detail

The research conducted by our team takes an interdisciplinary approach: hardware and software. Our team of two computer engineers and four computer science students split the research workload into their appropriate disciplines. We discussed amongst ourselves who would work on researching the appropriate topics related to the project.

Tech Stack

Since our project is being inherited from a previous team we were not required to make many considerations on the technologies used for software. Our hardware team will discuss in detail the selections made and considerations taken in the next section. This section will serve as a quick reference for all the technology used in this project. The following is a complete technology stack for VITALS as of November 24, 2025:

Table 1. Software Technology Stack

Purpose	Name	Description
Version Control	Git + GitHub	Used to maintain code base
Project Management	Jira + Google Sheets	Maintains team cohesion and is used to assign tasks.
Communication	Discord + Zoom	Discord for text, Zoom for meetings
Documentation	Confluence	Industry standard documentation tool
Object Detection	YOLOv11 + OpenCV	Produces bounding boxes on drone imagery
Vision - Language Model	LLaVA	Produce image descriptions on image

Communication Protocols	MCP + ACP	Enables agent communications
Vector Database	ChromaDB	Storing mission context items
Event logging	PostgreSQL	Structured mission logging
LLM	LLaMA 3.2 (Ollama)	For user input interpretation
Communication Framework	LangGraph	Structured RAG + MCP pipelines
Map Data	OpenStreetMap	Used in terrain pre processing for informed pathing
Path Planning	A*	For drone waypoint collection
Drone Communication	MAVLink	Interact with onboard flight controller to send commands and receive telemetry data
Ground Control Software	Mission Planner	Enables worry-free initialization with drones.
Programming	Python 3.11.9	Best for AI and ML software such as VITALS
Containization	Docker	Assist deployment onto hardware

Hardware

Team A39 distinguishes the VITALS project by integrating dedicated hardware into the solution. The core of the system is the NVIDIA Jetson Nano Edge Compute Device, which serves as the AI processing hub for running real time object detection models, crucial for identifying victims and hazards during SAR missions. The Jetson Nano also enables an interactive chat interface, empowering operators to send mission planning prompts and receive immediate AI-driven feedback. This architecture supports both fully autonomous operation and direct human interaction, enhancing the agility and effectiveness of the SAR Ground Control Station.

The drone hardware research will be completed by Brendan, who will provide comprehensive documentation on the inherited senior design drone built by a previous team during the Spring 2025 semester. His research will include specifications for the flight controller,

propulsion system, payload capacity, communication modules, and any existing sensor packages that may be leveraged for the VITALS vision system.

Stephen will focus on researching and designing the battery power supply subsystem, which is crucial to achieving our MVP's field operation requirements. He will evaluate battery chemistry options (lithium-ion vs. lithium-polymer), capacity requirements for multi-hour operations, charging circuits, and power distribution to ensure stable voltage delivery to both the Jetson Nano and peripheral devices. Stephen is also responsible for understanding how the NVIDIA Jetson Nano operates as an embedded AI platform, including its CUDA core architecture, thermal management requirements, and GPIO capabilities. He will determine how the team can successfully deploy the VITALS software solution onto this AI hardware device, addressing potential challenges such as dependency management, performance optimization, and integration with the Mission Planner ground control software.

Drones

Drone hardware selection is critical for seamless system integration. All deployed drones must be compatible with ArduPilot firmware to interface reliably with the Mission Planner software. Each UAV is equipped with a compatible flight controller, ensuring robust communication between the drones and the ground control station. Radio modems are key components for validating system functionality; they provide wireless transmission and reception of MAVLink packets, enabling real time control and telemetry exchange. Any drone that supports MAVLink communication and connects with Mission Planner can, in principle, be incorporated into the swarm, ensuring flexibility and scalability for the overall UAV fleet.

The VITALS project’s drone research encompasses both student-built and commercially available UAV platforms. Evaluating both types is essential. Organizations operating under stringent regulatory standards typically prefer commercial drones, such as those from DJI, because of their robust manufacturer support, reliability, and compliance with aviation requirements. In contrast, entities interested in innovation or custom capabilities often opt to build or modify their own drones to enable specialized solutions and experimentation

For instance, companies like Duke Energy, which prioritize operational reliability, are more likely to invest in established commercial UAV products rather than allocate resources to designing custom drones. Conversely, an organization like Lockheed Martin Aerospace and Defense, with a focus on advancing SAR technology, is well positioned to invest in research and development initiatives involving custom drone design, testing of alternative flight controllers, and the exploration of novel UAV solutions to address unique mission requirements.

Swarm Operations & Autonomous Rescue - SOAR

The Swarm Operations & Autonomous Rescue (SOAR) team, active in Spring 2025, developed a drone platform that now forms a part of the hardware foundation of the VITALS (L26) project. This section describes the fundamental hardware elements of the SOAR drone, which include the power systems that ensure reliable energy delivery for extended flight, robust communications equipment to maintain data links between drone and ground stations, and specialized radios and telemetry modules that enable the transmission of MAVLink messages for mission control and monitoring. Networking capabilities allow the drone to coordinate with other UAVs as part of a swarm, while integrated sensors such as cameras, GPS units, and environmental detectors provide the situational awareness and data needed for effective autonomous rescue operations. This overview provides context for how these systems collectively support the technical objectives and mission readiness of the inherited SOAR platform.

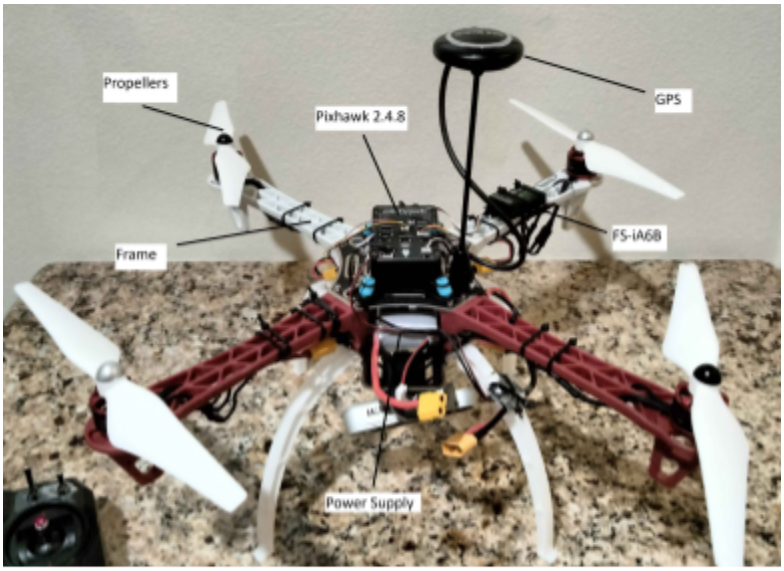


Figure 1.1: Current VITALS Drone

Power Systems

The SOAR drone is powered by a high capacity 4200mAh lithium polymer (LiPo) battery designed to provide efficient energy storage and delivery for sustained multi-rotor flight operations. Electrical distribution is managed through a dedicated power module, which supplies stable voltage regulation and offers real time telemetry on voltage and current to the Pixhawk 2.4.8 flight controller as well as electronic speed controllers (ESCs) for the motors. This module enables accurate battery status monitoring and can automatically trigger failsafe procedures based on low power thresholds, ensuring mission continuity and flight safety.

Flight Controllers

Central flight control is provided by the Pixhawk 2.4.8 autopilot. This module interfaces directly with electronic speed controllers (ESCs), receiver, and navigation sensors, managing stabilization, autonomous navigation, and actuator commands. Backup or advanced research capabilities are supported via the Cube flight controller, allowing future upgrades or redundancy.

Communications Equipment & Telemetry

Wireless communication for the SOAR drone relies on a FlySky FS-iA6B receiver operating at 2.4 GHz, utilizing the AFHDS 2A protocol with six available channels and dual diversity antennas to ensure reliable flight control, payload management, and reduced susceptibility to interference and signal dropouts. The Pixhawk 2.4.8 flight controller facilitates external telemetry module connectivity via multiple UART ports (TELEM1, TELEM2), supporting MAVLink based telemetry radios that enable robust bidirectional data links to ground control stations for real time mission monitoring, control, and extended communication range and bandwidth. Comprehensive failsafe mechanisms are incorporated at both the receiver and flight controller levels, ensuring autonomous landing or return-to-launch (RTL) capabilities upon link loss, thereby maintaining operational safety during communications disruptions.

Mission Planner Connection

Mission Planner integration uses RFD x Series Radio Data Modems to connect the drone and the ground station computer, establishing the critical command and telemetry link essential for mission management and situational feedback.

GPS & Navigation

Navigation and position estimation rely on a u-blox M8N GPS module with integrated compass, mounted on a mast above the main frame to optimize satellite reception and compass accuracy. This unit provides real time geolocation and heading information critical for autonomous flight, waypoint tracking, and situational awareness under varying field conditions.

Frame & Mechanics

The drone frame is a multi-rotor design built with composite arms and 3D-printed landing gear. Arm color coding provides visual orientation. Shock dampers and sturdy baseplates protect sensitive electronics, while all modules are arranged for optimal accessibility, safety, and airflow. Parallel motor mounts enable stable flight and rapid maintenance or upgrades.

Onboard Networking

The Pixhawk 2.4.8 functions as the central processing and networking hub, interfacing with various onboard components through standard UART, I2C, and PWM/SBUS/PPM protocols. Onboard sensor, GPS, receiver, and ESC connections are managed through dedicated labeled connectors, permitting expansion to additional peripherals such as cameras or advanced sensors via the embedded I2C and CAN interfaces. This architecture enables modular subsystem integration and robust intra-system communication.

Prolusion & Motors

The propulsion system comprises brushless motors mounted to each arm , paired with electronic speed controllers (ESCs) fed by regulated power from the central module. Standard mutli-rotor propellers are used, with mountings allowing for quick replacement and balancing adjustments to maintain reliable lift and control.

Payload & Attachments

Mission-specific payloads include the Intel RealSense depth camera, attached to the lower section of the frame for visual mapping and perception. Modular mounting points and

wiring layouts allow rapid integration and testing of experimental sensors or delivery apparatus, supporting evolving project requirements.

Wiring & Connectors

All wiring is routed through robust connectors including JST, servo leads, and XT60 plugs, protected via zip ties, insulation, and shock absorbing mounts to minimize risk of mechanical failure and electrical noise. Each system section features color-coded, labeled connections for hassle-free setup and troubleshooting.

Safety & Troubleshooting

System reliability is enhanced by shock isolation for key modules, strain relief on connections, wiring organization, and visible power distribution paths to ensure robust, mission-ready operation. Redundant power and control features can be added as needed to further support safe autonomous function in field deployments.

Accessible modular construction, removable landing gear, and well-labeled wiring simplify routine maintenance and rapid troubleshooting. The system's architecture facilitates easy component replacement and diagnostic checks for flight control, power delivery, and communications modules.

Commercial Search and Rescue Drones

Commercial search and rescue drones are purpose-built platforms engineered for reliability, rapid deployment, and robust performance in emergency scenarios. Featuring intelligent power systems, advanced communication and networking technologies, and a wide range of integrated manufacturer features, these drones provide professional responders with essential tools for real time situational awareness and mission success. Our team aims to design our hardware and software solutions such that commercial SAR drones could connect with our system.

Power Systems

Commercial search and rescue drones, such as the DJI Matrice M30T and ZenaDrone 1000, use high-capacity intelligent battery systems designed for extended operations in demanding environments. These smart batteries often support rapid charging and hot-swapping, allowing for minimal downtime between flights. For example, the DJI Matrice 30T can utilize multiple TB30 flight batteries, managed through an intelligent battery station, enabling up to 40 minutes of continuous operation per charge and the ability to swap batteries quickly for longer missions.

Communications Equipment

Modern SAR drones leverage a combination of robust radio links and advanced digital communication protocols. Most commercial platforms incorporate high-speed, encrypted wireless transmission to support real time high-definition video streaming and telemetry. Advanced systems feature dual-band radios, cellular (4G/5G), or satellite communication options, enabling reliable data exchange even beyond visual line of sight (BVLOS). For instance, solutions like the Elsight Halo multilink communication module unify cellular, satellite, and private network links to maximize coverage, redundancy, and security for critical missions.

Drone Radios & Networking

Built-in networking features on commercial drones ensure seamless communication between the UAV, mission control, and sometimes other drones within the swarm. Drones employ proprietary and MAVLink-compatible telemetry, using dedicated ground control stations or mobile apps, often with integrated real time mapping, geofencing, and obstacle avoidance data. The use of omnidirectional antennas, mesh networking, and automated failsafe triggers further enhances mission reliability. Networking protocols, such as Wi-Fi, RF, or LTE/5G, are chosen based on range, regulatory, and operational requirements, ensuring secure and stable command and control during emergency response operations.

Manufacturer Features

Commercial SAR drones stand out for their integrated manufacturer features tailored for emergency operations. Key offerings include dual or multi-sensor payloads (visual, thermal, and

night vision cameras), laser range finders, spotlights, speakers, and modular payload capabilities. These drones are built for weather resistance, with IP54+ ratings for dust and water ingress, and are constructed to withstand harsh environmental conditions. Autonomy features, such as automated mission planning, object detection, and collision avoidance, streamline operation. For example, the ZenaDrone 1000 and DJI Matrice 30T offer AI-powered autonomous flight, rugged weather-resistant airframes, and easy deployment for rapid response, all essential advantages for first responders and rescue teams.

NVIDIA Jetson

NVIDIA's Jetson AGX Orin is a high-performance embedded AI platform integrating an Arm Cortex-A78AE CPU complex, an Ampere-architecture GPU with Tensor Cores, and dedicated deep-learning accelerators (DLA), packaged as a compact SOM for edge robotics and computer vision. Depending on module SKU, AGX Orin delivers up to 275 INT8 TOPS within power envelopes configurable from approximately 15 W to 60 W, with an Industrial variant extending to 75 W. Compared to the prior Jetson AGX Xavier generation, AGX Orin increases AI performance by up to 8× at similar form factor.

Architecture

At the heart of AGX Orin is the Orin SoC. The GPU implements NVIDIA's Ampere architecture with 2,048 CUDA cores and 64 Tensor Cores (AGX Orin 64GB), paired with a 12-core Arm Cortex-A78AE v8.2 64-bit CPU cluster. Two NVDLA v2 accelerators offload common inference layers, and a PVA v2 vision accelerator assists pre/post-processing tasks.

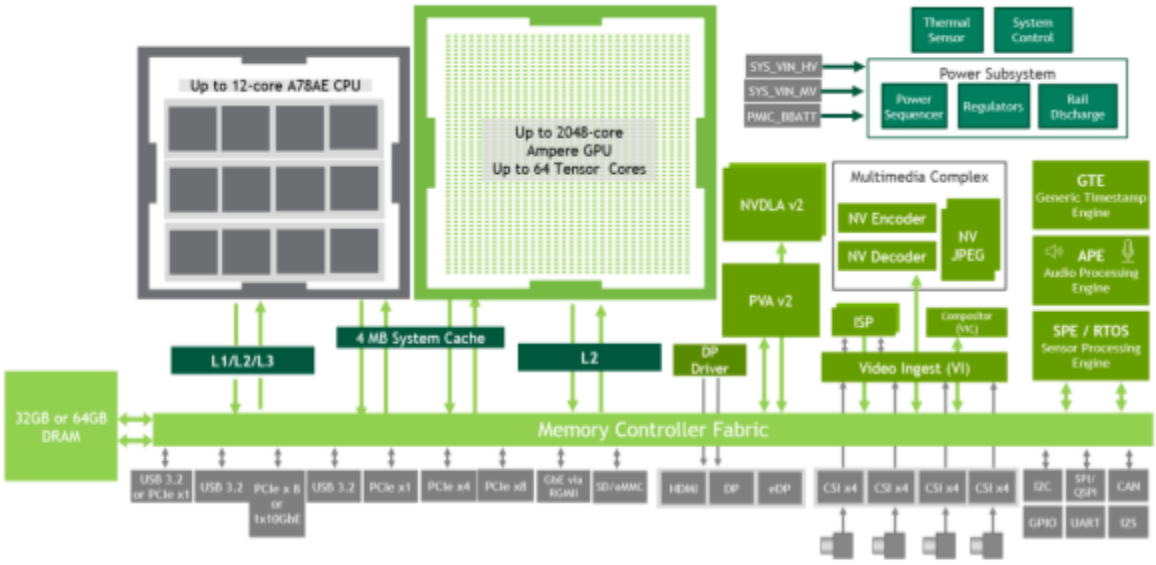


Figure 1.2: Jetson Orin Architecture Block Diagram

Unified LPDDR5 memory (up to 64 GB on the module) provides up to ~204.8 GB/s bandwidth. The developer kit exposes PCIe Gen4, MIPI CSI-2 camera lanes, 10GbE, USB 3.2, and display outputs. A high-level functional block diagram is shown in Figure 1.2.

Table 2. Specifications for the Jetson AGX Orin 64GB module and developer kit

Feature	Jetson AGX Orin (64GB developer kit)
AI Performance	Up to 275 TOPS (INT8)
GPU	Ampere GPU, 2048 CUDA cores, 64 Tensor Cores ≤ ~1.3 GHz
CPU	12-core Arm Cortex-A78AE v8.2 64-bit, up to 2.2 GHz
Deep Learning Accelerators	2× NVDLA v2 (up to ~1.6 GHz)
Vision Accelerator	1× PVA v2
Memory	64 GB LPDDR5, 256-bit, ~204.8 GB/s (ECC on Industrial)
Storage	64 GB eMMC 5.1 (module); NVMe via M.2 on dev kit
Video Encode	Up to 2× 4K60 (H.265); multiple 1080p streams
Video Decode	Up to 1× 8K30 or multiple 4K/1080p streams (H.265)
Cameras	Up to 16-lane MIPI CSI-2; D-PHY 2.1 / C-PHY 2.0
PCIe	Up to x8 Gen4 slot + M.2 Key M (x4 Gen4) + Key E (x1)
Networking	Up to 10 GbE (RJ45 on dev kit)
USB	Up to 3× USB 3.2 Gen2 + 4× USB 2.0 (dev kit)
Display	DP 1.4a/eDP/HDMI depending on carrier
Power (module)	Configurable ~15–60 W (Industrial up to 75 W)
Mechanical	Module ~100 × 87 mm; dev kit 110 × 110 × 71.7 mm (incl. thermal)

Table 3. Jetson AGX Orin Variants

Variant	AI TOPS (INT8)	Memory	Power Range	Notes
Orin 64 GB	≦ 275	64 GB LPDDR5	15-60 W	Max Performance

Orin 32 GB	≦ 200	32 GB LPDDR5	15-40 W	Lower peak power
Orin Industrial	≦ 248	64 GB LPDDR5 + ECC	15-75 W	Industrial quality

Software Stack (JetPack 6.x)

Jetson AGX Orin is supported by NVIDIA JetPack 6.x, which bundles Jetson Linux 36.x (Ubuntu 22.04-based), CUDA 12.x, TensorRT 10.x, cuDNN 9.x, VPI, and the Jetson AI stack. JetPack 6.2 (R36.4.3) adds performance updates and features such as improved generative-AI inference and Super Mode support for selected Orin modules. Developers can leverage higher-level SDKs, such as Isaac (robotics), Metropolis (video analytics), and Jetson Platform Services, to accelerate perception, planning, and deployment at the edge.

Power and Thermal Considerations

AGX Orin supports pre-optimized power budgets (e.g., ~15 W, 30 W, 50 W, with MAXN ≈ 60 W on the 64 GB SKU) using the nvpmodel framework. Power modes gate clocks and the number of active CPU clusters, GPU TPCs, and DLA engines. Designers should budget for peak thermal dissipation at MAXN and validate workloads across modes to meet performance-per-watt targets. Carrier designs must provide the module's HV (7–20 V) and MV (5 V) rails, follow recommended power sequencing, and respect thermal limits on the integrated Thermal Transfer Plate (TTP).

Security Features (Jetson Linux)

Jetson Linux implements a secure boot chain of trust anchored in on-die BootROM with public-key cryptography. Platforms that support Secure Boot Key (SBK) can encrypt boot images; fuses (write-once) store key hashes and settings. Jetson Linux also integrates OP-TEE for a Trusted Execution Environment and provides guidance for factory secure key provisioning. NVIDIA publishes regular security updates via JetPack releases; production deployments should track bulletins and update policies.

Representative Use Cases

AGX Orin targets autonomous mobile robots (AMRs), industrial inspection, multi-camera perception, medical imaging, and edge generative-AI copilots. The combination of

GPU + DLA enables concurrent DNN inference (e.g., detection, segmentation, pose estimation) alongside classical CV on the PVA, while the CPU handles orchestration, real-time control, and I/O servicing. The following are some deployment best practices:

1. Start with the developer kit to validate performance/power modes (nvpmodel) and IO bandwidth (CSI/PCIe).
2. Pin-compatible modules: design the carrier for the 64GB envelope to preserve headroom for upgrades or Industrial.
3. Thermal: use the NVIDIA thermal design guide; instrument hotspots (GPU, CPU, PMIC) and test at environmental extremes.
4. Storage: prefer NVMe for datasets and logs; retain eMMC for OS; use A/B update schemes.
5. Security: plan Secure Boot/FS encryption early; fuse keys in manufacturing only after process validation.
6. Software: target JetPack 6.2+ for CUDA 12.6/TensorRT 10.3; use containerized workflows for reproducibility.

Conclusion

Jetson AGX Orin represents a substantial step-up in edge AI capability versus earlier Jetson generations, coupling server-class inference throughput with embedded-friendly power envelopes and a mature software stack. Its balanced architecture, Ampere GPU, Arm CPU complex, NVDLA, and PVA, combined with high-bandwidth LPDDR5 and rich I/O, makes it a strong platform for robotics and multi-sensor perception. Careful attention to power modes, thermal design, and security provisioning is critical to delivering reliable products at scale.

Power Supply Solution

This section details the design and implementation of a hybrid power supply system developed for an embedded artificial intelligence (AI) platform using the NVIDIA Jetson AGX Orin 64 GB module. The power system is optimized for field-deployable computer vision workloads, emphasizing reliability, safety, and runtime efficiency. The design integrates dual 4S LiPo battery packs, a regulated 12 V distribution rail, and a prioritized external power supply input to support long-duration AI inference and vision processing tasks in mobile environments.

The system was designed with the following objectives: (1) ensure uninterrupted power delivery under dynamic conditions, (2) allow safe integration of lithium-polymer batteries, (3) provide comprehensive telemetry for energy management, and (4) enable modular expansion for peripherals such as a high-brightness field monitor. The implementation uses Analog Devices (ADI) components for source control, inrush management, and power monitoring.

System Overview

The system provides stable DC power for both the Jetson AGX Orin and an Ikan VXF7-HB display. It supports three input sources with automatic priority-based switchover:

- V1 – OEM 19 V Power Supply (through LTC4231 hot-swap controller)
- V2 – Battery A (4S Li-Po 7,200 mAh)
- V3 – Battery B (4S Li-Po 7,200 mAh)

When the PSU is connected, it powers the system (battery charging is not currently supported). If the PSU is disconnected or invalid, Battery A automatically takes over. If Battery A reaches its undervoltage threshold, the LTC4417 seamlessly transfers to Battery B without system interruption.

Power Path and Regulation Design

The power architecture consists of four functional blocks: input protection and hot-swap, prioritized power control, DC-DC conversion, and telemetry monitoring.

1. Input Protection – Each power input includes a fuse and a transient voltage suppressor (TVS) diode to protect against faults. Batteries A and B use SMBJ18A diodes, while the PSU uses an SMBJ26A diode.
2. Prioritized Power Path – The LTC4417 monitors V1-V3 and selects the highest priority valid input. The power brick (V1) is the highest priority, Battery A (V2) is next, and Battery B (V3) is lowest.
3. Voltage Regulation – An LT8640S buck converter supplies a regulated 12V rail for the Jetson and monitor. An optional LT8609S can provide 5 V auxiliary power if needed.
4. Telemetry – One LTC2946 monitors the main bus current, and two LTC2944 devices measure the individual battery parameters.

Each LTC4417 channel drives an external P-channel MOSFET to isolate and switch its source. V1 uses a hot-swappable 19 V brick via a LTC4231; V2 and V3 each connect to a dedicated battery though a fuse, TVS diode, LTC4231, and MOSFET. When multiple sources are valid, only the highest-priority channel conducts. The unused channels remain off, blocking reverse or cross current. Each input defines its valid operating range using resistor dividers connected to the LTC4417's UV and OV pins. For the 4-cell Li-Po batteries, the undervoltage threshold is set to 13.0–13.4 V and the overvoltage limit to 16.8–17.0 V The PSU channel is configured with an 18.5 V undervoltage threshold. Hysteresis resistors on each channel maintain stability during slow voltage transitions or transient dips.

Table 4. Key Analog Devices Components and Functions

Device Function	Key Features
LTC4231	Hot-swap controller (PSU) 36 V max, programmable current limit
LTC4417	Triple PowerPath controller 28 V max, PSU-priority logic
LT8640S	12 V buck converter 42 V input, 6 A output, 95% efficiency
LTC2944 / LTC2946	Telemetry (per battery / bus) High-accuracy current & power monitoring

Telemetry and Control

Telemetry is implemented using two LTC2944 battery gas gauge ICs (one per battery) and an LTC2946 system power monitor. Each device measures current through precision 10 mΩ 4-terminal shunts, providing voltage, current, and power data to the Jetson via I²C. The sensing network includes RC filters (100 Ω + 10 nF) for noise suppression. Collected data allows runtime estimation, energy logging, and low-voltage alerts.

Peripheral Power Branches

A 12 V branch supplies the Ikan VXF7-HB field monitor, rated for 7–24 V DC input and approximately 12 W maximum consumption. Since the monitor is permanently connected, its power path is simplified: a 2 A fuse, an optional ferrite bead (>3 A, 100 Ω @ 100 MHz), and local decoupling capacitors (0.1 µF + 47 µF). A transient suppressor may be added if the cable exceeds 1 m or operates outdoors.

Performance and Runtime Analysis

Two 7.2 Ah LiPo batteries provide approximately 106 Wh each, with an estimated usable capacity of 96 Wh per pack after conversion and reserve margins. At 40 W system load (Jetson ~30 W + monitor ~10 W), the total runtime is approximately 4.8 hours. Efficiency measurements show 88–90% end-to-end conversion efficiency. Table 5 summarizes expected runtimes under different load conditions.

Table 5. Estimated Runtime for Two 7.2 Ah 4S LiPo Packs.

Load (W)	Usable Energy (Wh)	Runtime (h)
30	192	6.4
40	192	4.8
50	192	3.8
60	192	3.2
70	192	2.7

The finalized power architecture successfully integrates ADI analog control, telemetry, and protection devices into a modular, high-efficiency system for the Jetson AGX Orin. The design supports safe LiPo battery operation, automatic PSU priority, and accurate power monitoring without introducing significant weight or complexity. Its scalability allows easy integration of higher-capacity battery modules or additional peripherals while maintaining electrical integrity and field reliability.

Lithium Polymer (LiPo) Batteries

Lithium Polymer (LiPo) batteries are a subclass of lithium-ion energy storage devices characterized by their use of polymer-based electrolytes, high energy density, and lightweight packaging. LiPo batteries have become the dominant power source for unmanned aerial vehicles (UAVs), robotics, and portable electronic systems due to their superior power-to-weight ratio and flexible form factor. This report examines LiPo battery chemistry, manufacturing standards, performance characteristics, safety considerations, and applications within consumer and professional drone systems.

Chemistry and Internal Construction

A LiPo cell consists of a graphite anode, a lithium cobalt oxide ($LiCoO_2$) or nickel manganese cobalt oxide (NMC) cathode, and a polymer-based electrolyte that serves as the medium for lithium-ion transport. Unlike cylindrical lithium-ion cells that use liquid electrolytes and metal cans, LiPo batteries employ a gel-like polymer electrolyte housed within flexible aluminum-laminated pouches. This construction reduces weight and enables non-standard shapes, which is particularly advantageous for aerospace and robotics applications where form factor is critical.

Each LiPo cell operates at a nominal voltage of 3.7 V, with a full-charge voltage of 4.2 V and a discharge cutoff near 3.0 V. Multiple cells are connected in series (S) to increase voltage or in parallel (P) to increase capacity. For example, a 4S LiPo pack consists of four cells in series, yielding a nominal voltage of 14.8 V (4 × 3.7 V), while a 6S pack provides 22.2 V.

Table 6. Typical LiPo Cell Specifications

Parameter	Typical Value
Nominal Voltage	3.7 V
Full Charge Voltage	4.2 V
Discharge Cutoff	3.0 V
Energy Density	150 - 250 Wh/kg
Operating Temperature	-20 °C to +60 °C
Cycle Life	300 - 500 Cycles

Performance Characteristics

LiPo batteries are favored for applications demanding high discharge rates and stable voltage output. Key performance parameters include capacity (mAh), discharge rate (C-rating), and internal resistance (mΩ). The C-rating defines the maximum continuous discharge current as a multiple of the rated capacity: $I = C \times \text{Capacity}$. For example, a 5000 mAh pack rated at 50C can theoretically supply 250 A of continuous current. However, sustained high-current operation increases internal heating and reduces cycle life.

Compared with traditional lithium-ion cells, LiPo batteries offer higher specific power but slightly lower energy density due to thicker polymer electrolytes. Voltage sag under high load is minimized through the use of low-impedance cell chemistry and improved electrode coatings.

Safety and Handling Considerations

LiPo batteries are chemically energetic systems that must be handled with care to prevent overcharging, over-discharging, or thermal runaway. Thermal runaway occurs when internal temperatures exceed 130°C, causing decomposition of the electrolyte and rapid gas generation. To mitigate risk, proper charging, balancing, and storage practices must be observed.
Cells should be charged using a balance charger at ≤1C rate to ensure uniform cell voltage. The recommended storage voltage is approximately 3.8 V per cell (≈50% state-of-charge). Swollen or punctured packs must be immediately decommissioned, as they indicate gas generation and internal shorting.

Table 7. Summary of LiPo Safety Guidelines

Guideline	Recommended Practice
Charging Voltage	4.20 ± 0.05 V per cell
Discharge Limit	≥ 3.0 V per cell
Storage Voltage	3.8 V per cell
Charging Rate	≤ 1C (standard)
Temperature Range	0–45°C (charging), -20–60°C (discharge)

Standards and Certification

LiPo batteries used in commercial products must comply with several international safety and transportation standards. The most common include UL 1642 (Safety of Lithium Batteries), IEC 62133 (Safety Requirements for Portable Sealed Secondary Cells), and UN 38.3 (Transportation of Dangerous Goods). These standards evaluate performance under short-circuit, vibration, shock, and thermal stress conditions.

UN 38.3 certification is particularly critical for air transport, as it ensures that cells can withstand altitude simulation, impact, overcharge, and forced discharge tests without venting or fire. For drone manufacturers, compliance with these standards ensures legal shipment, insurance eligibility, and user safety.

Applications in Drones and Embedded Systems

LiPo batteries are ubiquitous in both consumer and professional drone systems. Consumer drones such as the DJI Mavic series typically use 4S or 6S 'smart' LiPo packs with integrated battery management systems (BMS) that provide telemetry on voltage, current, temperature, and state-of-charge. Professional UAVs, including survey and cinematography platforms, often use redundant 6S–12S configurations with capacities exceeding 20 Ah to extend flight times beyond 30 minutes.

In embedded and robotics systems, LiPo packs are valued for their high discharge capability and compactness, enabling efficient integration with single-board computers, AI accelerators, and brushless motor drivers.

Emerging Trends in LiPo Technology

Advancements in lithium polymer chemistry continue to focus on improving energy density, thermal stability, and lifecycle. High-voltage LiPo (HV-LiPo) cells extend maximum per-cell voltage to 4.35 V, yielding approximately 10% greater energy storage. Graphene-enhanced LiPo batteries reduce internal resistance and improve charge acceptance rates. Smart LiPo systems incorporating CAN-bus or SMBus interfaces provide advanced telemetry and balancing features for modern UAVs.

Beyond LiPo, solid-state lithium batteries promise improved safety and energy density by replacing liquid or polymer electrolytes with solid ceramic materials. While currently limited by cost and manufacturability, these technologies are poised to succeed LiPo cells in high-reliability aerospace and electric vehicle applications.

Conclusion

Lithium Polymer (LiPo) batteries represent a critical enabling technology for the modern UAV and embedded electronics industries. Their high specific power, flexible packaging, and mature manufacturing base make them the preferred choice for applications where energy density and discharge performance are paramount. However, adherence to strict safety and handling standards is essential to ensure operational reliability. Ongoing advancements in materials science and battery management will continue to refine LiPo performance and safety, while paving the way toward next-generation solid-state alternatives.

Software

The VITALS software platform orchestrates all critical functions required for autonomous drone-based search and rescue, integrating real time computer vision, mission planning, and robust operator interfaces. Developed for deployment on the NVIDIA Jetson Nano, the software suite includes advanced AI agents responsible for perception, reasoning, and navigation, as well as direct integration with Mission Planner and MAVLink protocols for precise drone control and situational awareness. Designed for modularity, cross-platform compatibility, and real time responsiveness, the architecture allows seamless communication between autonomous agents, hardware devices, and both GUI- and console-based operator tools delivering a flexible and scalable ground control solution tailored for demanding SAR operations.

Mission Planner

Mission Planner is a powerful, open-source ground control station software used to operate, monitor, and configure unmanned aerial vehicles (UAVs) running ArduPilot firmware. Designed primarily for Windows, it allows users to load firmware, configure and tune vehicle parameters, and plan missions visually by defining waypoints on mapping interfaces. Mission Planner also supports mission logging, real time telemetry monitoring, analysis, and can interface with flight simulators, making it a versatile tool for both autonomous

Figure 2.1: Mission Planner GUI

mission execution and hands-on control across a range of drone and unmanned vehicle platforms.

VITALS and Mission Planner

Mission Planner acts as the operational heart of field deployments in the VITALS project, providing a comprehensive graphical interface for mission configuration, real time flight tracking, and telemetry display. Its map-centric workflow allows precise waypoint management and route adjustments even during live missions. Operators rely on Mission Planner to visualize UAV positions, status, and sensor feeds, granting immediate situational awareness and manual override capability when mission adjustments are needed under dynamic conditions.

The VITALS software suite augments Mission Planner by directly interfacing through mirrored MAVLink streams exchanged over TCP, enabling deep integration without modifying Mission Planner's core. This allows for advanced mission automation from the VITALS platform, such as programmatically updating flight plans, dynamically dispatching or retasking drones in response to AI detections, and synchronizing multi-agent coordination. The integration ensures any command or status processed by VITALS is immediately reflected in Mission Planner, supporting cohesive and synchronized operations across the system.

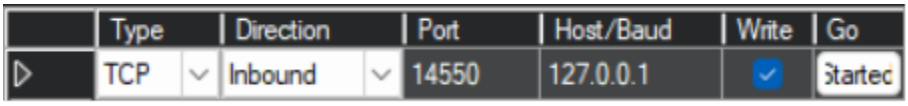


Figure 2.2: MAVLink Mirror TCP connection

VITALS Software

The VITALS software suite delivers the automation and intelligence driving the SAR missions, featuring modular Python agents dedicated to mission state management, computer vision, knowledge retrieval, and swarm coordination. Its extensible architecture is built to support rapid integration with new sensors, hardware platforms, and AI capabilities, while providing clear installation, deployment, and operator workflow tooling. By abstracting low-level communication and mission control, VITALS empowers both novice and expert users

to efficiently launch, monitor, and adapt complex search and rescue drone operations from a unified control station. This is the heart of VITALS as a project.

Current System Architecture

As the current architecture stands, VITALS is composed of six main modules which are called in a central manager file called missionState.py. These modules are: Dispatcher, GUI, Terrain Pre Processing, Path Planning, Computer Vision, and LangGraph. Figure 2.3 is a system pipeline diagram detailing how all the modules communicate with one including external tools like the openstreetmap database used for terrain pre-processing or mission planner which as stated previously, provides telemetry and MAVLink connection to VITALS.

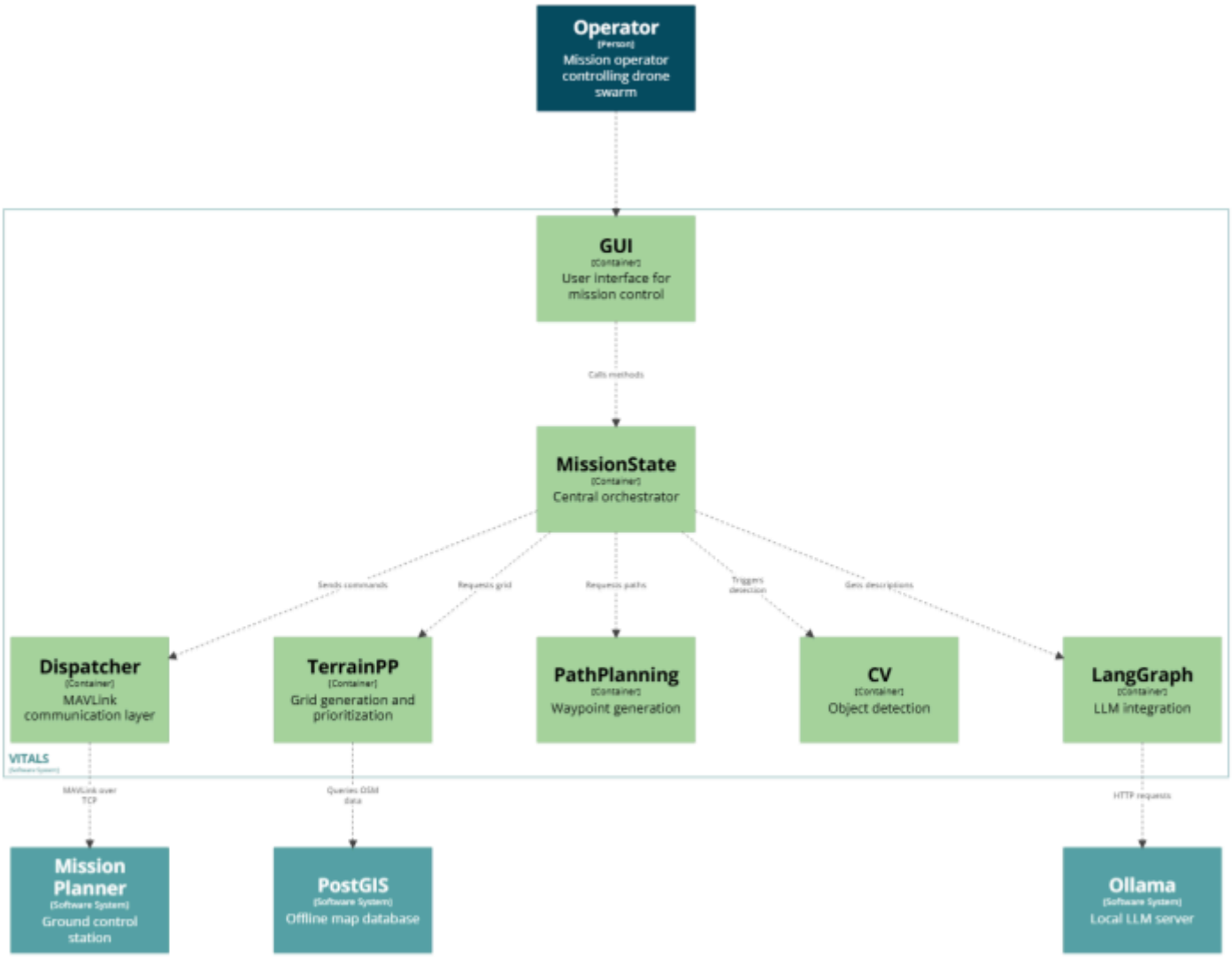


Figure 2.3: VITALS System Pipeline

The following sections will dive into each component and discuss the current state of each. Later in the design plan we will lay out our current plan for a new multi agent system implementation and then a timeline up to March 23rd for milestones we will achieve.

Dispatcher

The Dispatcher module implements the critical communication layer between the VITALS ground control station and the drone fleet, serving as the MAVLink protocol handler responsible for all bidirectional telemetry exchange, mission uploads, and flight control commands. This module transforms high-level mission intents from the missionState coordinator into low-level MAVLink packets understood by ArduPilot flight controllers, while simultaneously processing incoming telemetry streams to maintain real time situational awareness. The Dispatcher's architecture prioritizes minimal latency, reliable command delivery, and graceful handling of communication failures essential characteristics for maintaining safe drone operations during time-critical search and rescue missions.

The Dispatcher uses an asynchronous, event-driven architecture built on Python's asyncio framework. A dedicated event loop runs in its own thread, preventing telemetry processing and mission uploads from blocking the GUI or missionState. A single MAVLink connection (via mavutil.mavlink_connection()) handles packet serialization and runs in non-blocking mode, allowing continuous telemetry polling through the receive_packets() coroutine. Stateful behavior is managed through several lightweight structures:

- uploading_missions for tracking in-progress mission uploads
- unhandled_clears for filtering mission clear acknowledgments
- waiting_for_takeoff for monitoring takeoff altitude checks

Together, these structures implement the necessary MAVLink mission state machine while supporting concurrent mission activity across multiple drones.

Connection Establishment

The connect() method creates a MAVLink 2.0 TCP link (defaulting to tcp:127.0.0.1:14550) and waits for an initial HEARTBEAT to verify communication. Clear

diagnostics are printed when connection attempts fail, and the method returns a boolean indicator so the GUI can adapt accordingly. On failure, the master attribute is set to None to prevent accidental command execution.

Telemetry Processing

The receive_packets() loop continuously drains the MAVLink message queue with minimal delay, dispatching messages to handlers based on type. HEARTBEAT updates drone status, GLOBAL_POSITION_INT updates position and triggers takeoff checks, and mission-related messages drive the upload protocol. An adaptive sleep of 1 ms when no messages are available prevents busy-waiting while keeping latency negligible.

Mission Upload Protocol

Mission upload follows the standard MAVLink handshake: clear existing waypoints, send mission count, respond to sequential MISSION_REQUEST messages with MISSION_ITEM_INT packets, then wait for MISSION_ACK. A home waypoint is inserted at the start of every mission to ensure consistent altitude handling and predictable takeoff behavior. Successful acknowledgments trigger autonomous mission execution; errors halt the upload and are logged for operator review.

Mode Management and Takeoff

Mode transitions (GUIDED, AUTO, RTL) are handled through wait_for_mode(), which monitors HEARTBEAT messages until a mode change is confirmed or a timeout occurs. takeoff() manages grounded launches switching to GUIDED, arming, issuing MAV_CMD_NAV_TAKEOFF, and monitoring altitude until the threshold is met. start_mission() unifies mission execution logic for both grounded and airborne drones.

Command Execution and Safety Procedures

return_to_launch() uses ArduPilot's built-in RTL behavior to recover drones safely, while stop_current_mission() halts autonomous flight by switching to GUIDED and clearing stored waypoints. Additional debug-oriented methods such as arm_drone() and request_mission_list() expose low-level functionality for development and verification purposes.

State Management and Error Handling

Timeouts, retry logic, and explicit state tracking make the Dispatcher resilient to dropped packets, delayed telemetry, and inconsistent flight-controller behavior. Structures like unhandled_clears prevent race conditions between clear operations and mission acknowledgments, while waiting_for_takeoff ensures reliable, asynchronous altitude monitoring.

Integration and Concurrency

Mission uploads originate in the main thread but execute as coroutines on the Dispatcher's event loop via asyncio.run_coroutine_threadsafe(). This design enables concurrent mission uploads to multiple drones and a responsive GUI. Graceful shutdown is supported through coordinated event loop termination and thread joining.

Protocol Compliance and Optimization Opportunities

The module adheres closely to the MAVLink mission specification, using MISSION_ITEM_INT and MAV_FRAME_GLOBAL_RELATIVE_ALT for precision and consistency with ArduPilot. Although performance is already strong, future improvements could include adaptive sleep intervals, a dispatch table for message types, and more efficient upload strategies that minimize round-trip latency.

Conclusion

Overall, the Dispatcher offers a robust and cleanly structured communication layer that abstracts MAVLink's complexity behind an asynchronous interface. Its combination of non-blocking I/O, state machines, and careful threading makes it well-suited for VITALS' multi-drone architecture. With modest optimizations, it will scale even more effectively as VITALS evolves toward agent-based control and expanded swarm capabilities.

Graphical User Interface

The VITALS GUI system implements a multipage interface for autonomous drone swarm control using a modified Model View Controller (MVC) architecture. Built with CustomTkinter, the system manages real time drone telemetry, mission planning, and NLP commands through an

integrated LLM interface. The architecture emphasizes modularity, with twelve distinct classes working together to deliver mission control capabilities. The following is a description of each class and their purpose.

Figure 2.4: VITALS Graphical User Interface

Classes

Due to the lack of documentation for the current code base. Our team has dedicated a sprint to manually reviewing the repository and developing documentation pages for each module. We have decided to include a list describing the core classes since being familiar with the building blocks of the GUI will help readers understand future design plans we have for refactoring the code.

Drone Class

The Drone class maintains relative information such as state, position, altitude, telemetry (roll/pitch/yaw), velocity vectors, and job assignments. The implementation uses a color coding system (red, blue, green, yellow, there is currently a maximum of 4 color assignments) for visual distinction, with each drone managing its own map marker and heading indicator. The class encapsulates all drone related visualization and state management to maintain a single source of truth for each UAV. The setPosition() method performs coordinate conversion from MAVLink's integer format of 1e7 scaling to float coordinates, ensuring compatibility with the mapping system. The heading visualization uses trigonometric calculations to draw a directional line,

providing immediate visual orientation feedback. The job path visualization system allows operators to see planned routes, with intelligent trimming to show only unvisited waypoints. The use of after() scheduling for widget updates prevents race conditions when clearing and recreating UI elements. We can instantiate a drone object given map widget, info container, job container, drone id, system status, and gui ref.

Job Class

A small data structure that represents mission jobs with start position, waypoint list, end position, and path visualizations. This class is intentionally lightweight, serving as a pure data container. The simplicity reflects that job logic and processing occur in MissionState, while the GUI only needs to store and display job information. The path obj reference is intended to maintain the visual representation on the map. We can instantiate a job object given start, waypoints, end, and path obj. The job class doesn't have methods, since it is mainly used as a data container.

Figure 2.5: Drone Job Interface

POI Class

Points of Interest (POI) manage detection locations, target confirmation workflows, and associated imagery. Each POI maintains geographic coordinates, descriptive metadata, detection flags, and interactive popup capabilities. The POI class handles the complete lifecycle of a detection event, from initial marking to target confirmation. The target found() method implements a decision point with a modal popup, forcing operator attention for mission-critical choices (continue searching vs. end mission). The popup window design with scrollable frames accommodates multiple detection images while maintaining a reasonable window size. Image handling integrates with the mission folder structure (./Missions/mission_id/POIs/poi_id/), providing organized storage for evidence collection. The class uses both a map marker and an info box widget for dual representation, ensuring visibility in both map and sidebar contexts. We can

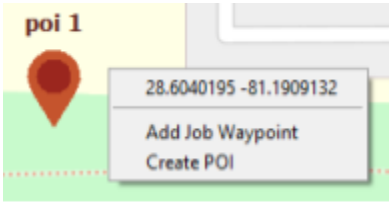


Figure 2.6: POI Display

instantiate a POI object with a given latitude, longitude, name, description, map widget, info container, poi count and a gui ref.

POIInfoBox Class

A compact CustomTkinter frame that displays POI summary information in the sidebar with click to expand functionality. Inheriting from CTkFrame allows direct integration into the CustomTkinter widget hierarchy. The blue background provides visual consistency with active elements. The frame uses grid layout for structured display of name and coordinates, with the entire frame clickable to trigger the detailed popup. This class is self contained and can only be instantiated given parent, name, latitude, longitude, poi count and popup func. No methods exist in this class.

ChatSideBar Class

The LLM interface implements a chat system with asynchronous message processing, tool calling capabilities, and visual message differentiation. The sidebar uses a Canvas based scrollable area rather than a text widget to enable formatting with message bubbles. The blue/gray color scheme distinguishes user/LLM messages. ThreadPoolExecutor enables nonblocking LLM calls, mainly for maintaining UI responsiveness during potentially long API calls. The embedded tool system (create poi investigate job, call return to launch, call end mission) provides a clean interface between natural language commands and drone operations. The future/callback pattern ensures thread safe UI updates after asynchronous operations complete. Dynamic wraplength calculation (70% of sidebar width) ensures readable text formatting across different window sizes.

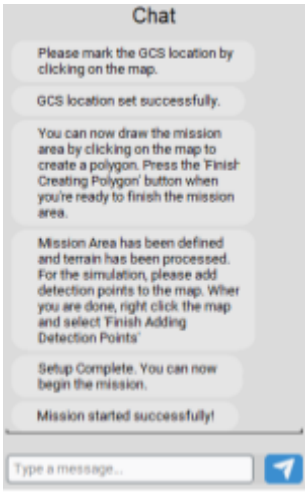


Figure 2.7: Chat Sidebar

jobInfoContainer Class

A container widget that organizes job displays for individual drones using a scrollable frame structure. This small wrapper class provides organizational structure in the bottom panel, with each drone getting its own column. The scrollable frame handles variable numbers of jobs

without breaking the layout. The separation container from items follows the composition pattern, allowing for job list management. This wrapper class doesn't contain methods.

jobInfoItem Class

Individual job representation widgets that display job details with visual status indicators. Each job item is a self contained frame with information displayed such as type, status, waypoints and priority. The color coding (blue for active, gray for queued) provides a visual status recognition. The click binding to toggle path visualization creates an interactive element without cluttering the display with buttons. Grid layout ensures consistent alignment of multiple data fields within each job card. An object of jobInfoItem can be instantiated with the following, parent, job, row, drone, and active flag.

DroneInfoBox Class

real time telemetry display with integrated dronespecific settings management. The class provides a dense information display optimized for the sidebar's limited space. Position coordinates use 7 decimal precision for accuracy, while altitude converts from millimeters to meters for readability. The integrated settings popup allows per drone configuration without leaving the operational interface. Settings include operating altitude, vision model selection (dynamically populated from ./ComputerVision/CVModels/), and camera parameters (FOV, mount angle). The modal settings window uses attributes("-topmost", True) to ensure visibility over the main interface. Status color coding (gray, yellow, orange, green, red) follows standard aviation conventions for system states. An instance of DroneInfoBox can be instantiated with the following, parent, id and drone ref.

MapPage Class

The primary operational interface combines all mission control elements into a cohesive workspace. MapPage uses grid layout to create distinct regions: left sidebar for controls, center for map, right for chat/POIs, bottom for job lists. The offline tile server fallback (http://localhost:8080/tile/z/x/y.png) ensures operation without internet connectivity. Polygon drawing for

Figure 2.8: Map Page Interface

search areas uses state management (editing polygon, first polygon point) to handle the multi click interaction pattern. The right click context menu provides quick access to common operations without cluttering the interface. There is a debug menu hidden behind a button that provides testing capabilities without compromising the production interface. Collapsible elements and scrollable frames maximize usable space on various screen sizes. An object of MapPage can be instantiated with the following, parent, switch to home and gui ref.

HomePage Class

The initial landing page provides mission type selection and welcome messaging. The minimalist design reduces cognitive load at mission start. Modal popup for mission type selection ensures deliberate choice between simulation and real operations. The simulation 5 mode explanation helps users understand the detection point system for testing. Simple pack geometry reflects the straightforward linear flow of the home screen. This class has no methods. JobWaypoint Class: Lightweight waypoint representation for visual job creation. This simple class creates numbered markers during manual job definition. The minimal implementation reflects its temporary nature, waypoints exist only during the job creation process before being converted to Job objects. This class has no methods

GUI class

The main controller orchestrates the entire application lifecycle and intercomponent communication. The GUI class implements a design pattern that provides a unified interface between the UI layer and MissionState backend. Page navigation uses pack/forget rather than destroy/create, preserving state during transitions. The callback method pattern (updateDronePosition, updateDroneStatus, etc.) enables loose coupling with MissionState through event driven updates. Mission folder creation with timestamp based IDs ensures unique storage for each operation. The GCS location setting workflow with confirmation popup prevents accidental misplacement of this critical reference point. Detection point management for simulation mode provides a controlled testing environment. The comprehensive debug function set (call takeoff mission, call arm mission, etc.) enables component testing without full system integration. System chat messages provide operational feedback and guidance throughout mission workflows.

Conclusion

In summary, the VITALS GUI system implements an architecture where the GUI class serves as the central orchestrator, instantiating and managing the two main page frames, HomePage and MapPage. When a mission begins, the GUI transitions from HomePage to MapPage through pack and forget operations, preserving state while switching views. The MapPage acts as a composite container, instantiating and managing collections of Drone objects stored in self.drones list, POI objects in self.pois list, and singular instances of ChatSidebar and multiple jobInfoContainer widgets. Each Drone instance maintains bidirectional relationships with the MapPage's map widget for marker and path rendering, while also creating and managing its own DroneInfoBox and jobInfoContainer widgets that are attached to MapPage's sidebar and bottom panel respectively. The jobInfoContainer dynamically creates jobInfoItem widgets for each job in the drone's queue, with these items maintaining a reference back to their parent Drone for triggering path visualization through click events. When POIs are detected, the MapPage creates POI instances which then instantiate their associated POIInfoBox widgets in the sidebar and place markers on the map, with both representations linked through event handlers that trigger the POI's detailed popup display.

The data flow follows a pattern where the GUI class implements callback methods updateDronePosition(), updateDroneStatus(), updateJobs(), and addPOI()) that receive updates from the MissionState, then propagates these updates to the appropriate MapPage collections, which in turn update their child components. The ChatSidebar operates semi-independently, using ThreadPoolExecutor to make asynchronous calls to the LangGraph module, then executing tool functions that call back into GUI's MissionState interface methods creating a closed loop command system. The Job and JobWaypoint classes serve as passive data structures that are created, stored, and passed between components, with Job objects flowing from MissionState through GUI callbacks to Drone instances, where they're visualized in jobInfoItem widgets and rendered as paths on the map. This architecture creates multiple layers of abstraction where high level mission commands flow down through GUI to MapPage to component hierarchies, while telemetry and status updates flow up through callbacks.

GUI Workflow

The VITALS system was designed with a sequential workflow that guides operators through the process of search and rescue through the operation and coordination of multiple autonomous drones. The software architecture has a balance between automation and operator control, implementing what the design document describes as a ”human-in-the-loop” model where the AI handles routine operations but defers to human judgment for critical decisions and primary mission creation. The software implements a seven phase workflow that transitions operators from mission initialization through active search operations.

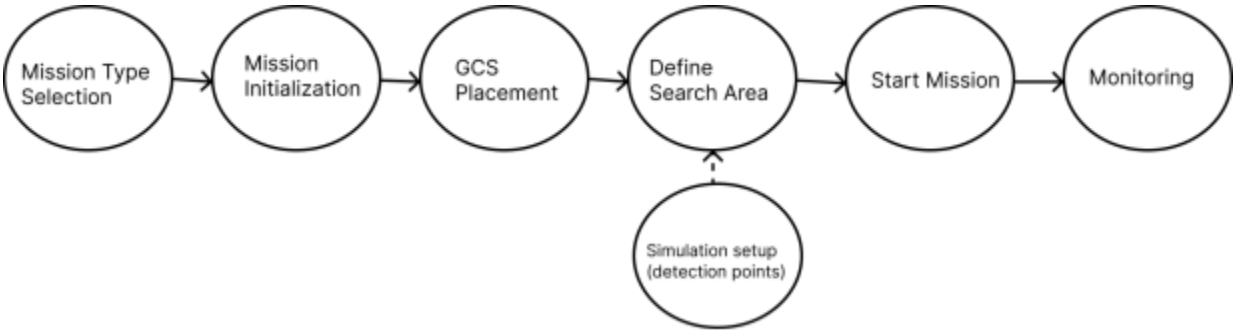


Figure 2.9: GUI Workflow Diagram

Mission Type Selection:

Upon launching the application, operators are immediately presented with a decision point through the HomePage interface. The modal popup forces a choice between simulation and real mission modes. This design decision ensures operators select the appropriate operational context, as simulation mode enables detection point placement for testing without physical drones, while real mission mode expects actual MAVLink connections. The software accommodates both training scenarios and actual deployments through this branching workflow.

Mission Initialization and Connection Once on the MapPage:

The workflow enforces a strict connection sequence. The ”Connect to Mavlink” button initiates communication with the drone swarm through Mission Planner on TCP port 14550. The GUI provides immediate feedback through both button state changes (disabled, text changed to

”Connected”) and system chat messages. This design pattern of dual feedback ensures operators maintain situational awareness throughout the setup process.

Ground Control Station Establishment:

The software requires Ground Control Station (GCS) location definition before any other mission parameter, reflecting the need to have this reference point for return to home commands and range calculations. The left click event handler enters a special mode (choosing gcs location = True) where map clicks trigger a confirmation dialog rather than normal interactions. This modal confirmation popup prevents accidental GCS placement, and upon confirmation, the system automatically transitions to the next phase.

Search Area Definition:

The polygon drawing interface, though obtuse and simple, accommodates a visualization for the area to be searched. The first click initiates polygon creation, subsequent clicks add vertices, and visual feedback shows the developing search boundary in real time. The button text dynamically changes from ”Start Creating Polygon” to ”Finish Creating Polygon,” providing a state indication. Upon completion, the polygon is sent to MissionState for terrain preprocessing, where the grid based path planning algorithm analyzes geographical features like buildings to optimize search patterns.

Simulation-Specific Setup For simulation missions:

The workflow includes an additional phase for placing detection points, simulated targets that replace actual computer vision detections. This design allows for testing of the entire system logic without requiring physical drones or actual missing persons. The right click menu dynamically adds a ”Finish Adding Detection Points” option.

Mission Execution:

The ”Start Mission” button only becomes enabled after all prerequisites are met, preventing premature mission launch. Once started, the button transforms to ”End Mission”, and the system begins autonomous operations. The MissionState generates search patterns based on

the processed terrain data, assigns jobs to drones through priority queues, and manages the coordination of multiple UAVs.

Active Operations and Monitoring During mission execution:

The interface provides multiple synchronized views. The map displays real time drone positions with color coded markers, heading indicators show drone orientation, and path visualizations reveal planned routes. The bottom panel shows job queues for each drone, with active jobs highlighted in blue and queued jobs in gray. The sidebar displays telemetry including position, altitude, velocity, and system status with aviation standard color coding.

LLM Integration and Natural Language Control

The ChatSidebar implements a simple natural language interface with Ollama. The idea is to have operators issue high level commands like ”investigate the northeast corner” or ”send drone 2 to check POI 3,” which the LLM translates into specific function calls. The asynchronous architecture using ThreadPoolExecutor ensures the UI remains responsive during LLM processing. The tool calling architecture provides three primary functions; create poi investigate job: Assigns drones to investigate specific points of interest, call return to launch: Commands individual drones to return home, call end mission: Terminates the entire operation and recalls all drones.

POI Management and Decision Points

The POI system exemplifies the human-in-the-loop philosophy. When computer vision or operator observation identifies a potential target, the system creates a POI with associated imagery stored in the organized mission folder structure. The idea is to have a target confirmation workflow: when a drone confirms a target at a POI, the system doesn’t automatically end the mission but presents the operator with a modal decision dialog. This design respects the complexity of search and rescue operations where finding one victim doesn’t necessarily mean the search should end. The operator can choose to continue searching for additional victims or end the mission based on their assessment of the situation. All POI data, including detection images, remains accessible through clickable info boxes and popups. The job management system reflects the design requirement for autonomous operation with human

oversight. Each drone maintains an active job and a priority queue of pending jobs. The system automatically reassigns drones when high priority tasks emerge, if a new job has priority 3+ points higher than the current job, it preempts execution. The visual job tracking system, with clickable job items that toggle path visualization, allows operators to understand drone intentions without cluttering the display.

Scalability and Swarm Coordination

The architecture accommodates multiple drones through its collection based design. The MapPage maintains lists of drones, POIs, and jobs that can scale to the swarm size. Each drone operates semi-independently with its own job queue while coordinating through the central MissionState. The color coding system (supporting at least four drones distinctly) and per-drone telemetry panels ensure operators can track individual units within the swarm context.

Conclusion

The VITALS GUI demonstrates a workflow design that guides operators through complex mission setup while providing flexibility for various operational scenarios. The progressive revelation of functionality, comprehensive feedback mechanisms, and thoughtful accommodation of field conditions create a system that balances autonomous capability with human control. The architecture's emphasis on visual feedback, state management, and intuitive interaction patterns reflects deep understanding of search and rescue operational requirements, creating a tool that enhances rather than complicates the critical mission of finding missing persons.

Computer Vision & Object Detection

Computer vision is a foundational capability within the VITALS system. A robust CV module enables operators to broaden their search coverage while maintaining confidence that automated detection will highlight potential targets of interest. By ensuring that the swarm can reliably communicate its detections back to the operator, the system improves response times, enhances situational awareness, and helps mitigate human error especially in high-tempo field environments.

To meet operational needs, the CV module must perform several core functions: ingest real time imagery, execute object-detection inference efficiently, annotate detections within the visual feed, and notify the operator when objects are recognized above a defined confidence threshold. As part of this effort, we will evaluate the current CV implementation within the VITALS software to determine performance gaps, documentation shortcomings, and potential opportunities to enhance user experience and system reliability.

The previous development team investigated multiple object-detection frameworks and selected the YOLO family of models due to their favorable trade-off between accuracy, speed, and computational efficiency critical characteristics for edge devices deployed in the field. While YOLOv5s was initially chosen for its lightweight footprint and strong real time performance, the current implementation appears to rely on a customized variant of YOLOv11. Unfortunately, the dataset used for training has not been documented, nor is there any formal explanation of how the object-detection pipeline is integrated into the broader VITALS architecture.

The following section provides a detailed summary of the computer vision system as currently implemented in the VITALS codebase and identifies key areas requiring refinement to ensure consistency, maintainability, and future scalability.

Computer Vision Directory

```
ComputerVision/
├─ objectDetection.py
├─ CVModels/
│  ├─ rf3v1.pt
│  └─ yolov11Custom.pt
└─ temp/
   └─ drone_testing1.jpg
```

The CVModels directory contains two custom trained object detection models designed to identify objects from aerial perspectives: rf3v1.pt and yolov11Custom.pt. Both models serve the same purpose but were developed using different training methodologies to allow for performance comparison. The rf3v1 model was trained using RoboFlow 3.0, a cloud based computer vision platform that automates much of the training pipeline, while the yolov11Custom model was presumably trained directly using the Ultralytics YOLOv11 framework, which

provides a greater control over training hyperparameters and architecture customization. Both models would have required an annotated dataset of aerial imagery with labeled bounding boxes around the target objects, followed by training sessions that iterated through the dataset to learn object detection patterns.

While the dataset used for training is not readily available to us, there are some old deprecated files outside of the app ComputerVision directory that contains references to an AFO-1 data file. It's likely that these models were trained on the Aerial Floating Objects (AFO) dataset, which is a freely available dataset designed specifically for maritime Search and Rescue applications. The AFO dataset contains approximately 40,000 hand-annotated instances of persons and objects floating in water across 3,647 images extracted from 50 drone video clips captured at various resolutions. This dataset is particularly challenging due to the small size of many floating objects, making detection difficult but highly relevant for real-world SAR operations. The training script references suggest the yolov11Custom model was trained for 100 epochs on this data, though the exact training parameters and performance metrics for the rf3v1 model remain undocumented in the available materials.

The temp folder within the ComputerVision directory contains a small set of aerial imagery used primarily by the previous VITALS development team for demonstration and testing purposes. These sample images allowed the team to quickly validate the object detection pipeline without requiring real time data from deployed hardware. While useful during prototyping, this folder does not represent a formal dataset and is not intended for training, evaluation, or operational use. As VITALS continues to advance toward field deployment, these temporary images should be replaced or supplemented with a properly curated dataset that reflects real mission environments and improves model reliability.

objectDetection.py

The objectDetection.py file serves as the core inference module for the computer vision system, providing the primary interface between the trained YOLO models and the rest of the application. This script loads the rf3v1.pt model and exposes functions that enable other components of the project to perform object detection on aerial imagery. The module utilizes the

Ultralytics YOLO library to run inference on input images and returns structured detection data including bounding box coordinates, object class labels, and confidence scores. By dynamically extracting class labels directly from the loaded model using model.names, the script ensures compatibility with the model's trained classes without requiring hardcoded label mappings. This file is critical to the project's functionality as it abstracts the complexity of model inference, allowing other modules to simply call detection functions without needing to understand the underlying YOLO implementation details.

The script defines two functions; detect_objects() and detect_and_draw() that are largely redundant in their core functionality. Both functions perform identical object detection operations: they load an image, run YOLO inference, iterate through detected bounding boxes, extract class IDs and confidence scores, and return a list of detection dictionaries. The only meaningful difference is that detect_and_draw() additionally uses OpenCV to draw green bounding boxes and labels directly onto the image before returning it, while detect_objects() returns only the raw detection data. This design suggests the functions were likely created for different use cases, one for headless processing where only detection metadata is needed, and another for visualization purposes. However, a more efficient design would combine these into a single function with an optional parameter (e.g., draw_boxes=False) to control whether visual annotations should be added, eliminating code duplication and simplifying maintenance.

Future of CV & VITALS

Discussion with our sponsor has led us to shift focus away from improving object detection. Implementing full computer vision elements into the current iteration of VITALS is not part of our minimum viable product. However, this does not mean we will ignore computer vision and its importance to the VITALS mission. Recall our user story hinges on the ability for an operator to let drones make detections autonomously and report back when a confidence threshold is met. To achieve these goals and set up future teams for success we have developed a new agent framework that will be further discussed in the design plan. The computer vision aspect of VITALS will be transitioned into what we call Agent A, responsible for receiving live video feeds and processing frames into detections and captions using the original YOLO model alongside LLaVA for captioning. So while the current implementation of the computer vision

module is severely lacking in functionality, we will continue to make computer vision a core part of our philosophy.

Path Planning & Search Logic

Path Planning and intelligent search logic is a key requirement for any autonomous guidance systems to function properly in real time scenarios. In order for a proper search to be done we must first properly understand the underlying structure of how spatial data is stored, indexed, and queried across the system. To achieve this objective, the previous VITALS team leveraged several core components, including OpenStreetMap data for reliable spatial information, and a combination of PostGIS enabled Docker environments to execute real time database queries and inference operations.

Open Street Map Database

The OpenStreetMap (OSM) database serves as the foundational geospatial data source for the VITALS software system, providing an open source, high-resolution representation of real-world terrain, infrastructure, and environmental features. The OSM database has been built largely by volunteers from scratch and released with an open-content license for individuals and projects to use freely as long as they abide by the Open Database License (ODbL) licensed under the OpenStreetMap Foundation (OSMF).

High Resolution Data Found within OSM:

1. Terrain and Natural Features
2. Infrastructure and Built Environments
3. Environmental and Land Use Features
4. Points of Interest
5. Country/state/town Boundaries

Most developed areas within OSM:

1. UK
2. Netherlands
3. Germany
4. USA
5. Canada
6. Baghdad
7. Haiti

Overview of OSM Relational Schema

A comprehensive breakdown os the OSM database structure is documented in detail on the official OSM wiki, which provides a very detailed explanation of every core table, relationship, and data type used within the platform. This documentation outlines how OSM organizes and provides the real-world information into an ERD centered around three specific aspects: nodes, ways, and relations, and how the supporting tables interact to form a complete database that can be queried using PostGIS and PostgreSQL.

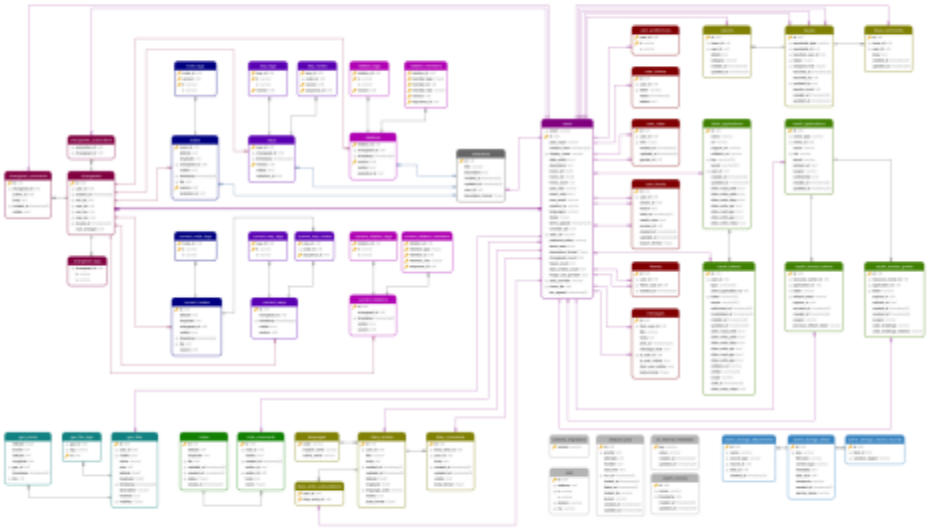


Figure 2.10: Complete ERD of the entire OSM database provided from the OSM wiki.

Furthermore, a complete schema is also provided within a GitHub link, allowing us access to see the source code of the actual SQL schema for each table within the database. By reviewing the schema, it becomes clear how spatial elements are stored, linked, and further enhanced with supporting tables.

Overview of the main OSM elements used by VITALS

The OSM database provides a robust and high-quality source of geospatial information, allowing VITALS access to detailed information such as roads, buildings, and more. This information within the database is held in several key database tables that we can find by analyzing the documentation. Key tables worth mentioning within the OSM database are as follows:

1. Nodes: Points defined by lat & lon that represent specific locations within OSM.

2. Ways: Ordered lists of nodes that form linear features (roads) or polygons (buildings).
3. Relations: Logical groupings defining complex relationships between nodes and ways.
4. Tags: KV pairs are used to denote real-world meanings.

By utilizing these elements within the OSM database, we will be able to access any specific location that is available within the data and any feature, such as a road or building, that is also present. Furthermore, within the data, there are logical groupings that allow us to infer the information being acquired from the data and parse logical context to the agents for proper analysis. A large number of the tables within the OSM database are metadata used to track the contributors adding to the database. The information that VITALS is concerned about is the geospatial information that may be available and its current relations to other geospatial data:

Figure 2.11: tables that contain geospatial information

However, raw OSM data is not immediately usable; it must be parsed, indexed, and served through a backend that supports efficient querying to make it usable. To do this, VITALS currently deploys the OSM Tile Server within a dedicated Docker container that runs alongside the rest of the software suite. This diverse and detailed data is critical for accurately modeling the environment in which the VITALS system will operate, enabling precise mapping and spatial analysis. To efficiently store, query, and manipulate this geospatial information, the system integrates PostGIS, a spatial database extension for PostgreSQL that supports advanced geospatial data.

PostgreSQL and PostGIS

PostgreSQL is an open source, relational database management system that supports complex queries and data types however, it is not able to support the spatial data needed to make this project work. PostGIS is a spatial extension built on top of PostgreSQL that adds support for geographic objects, allowing spatial queries, geometric computations, and spatial indexing.

Overview of the OSM Tile-Server used by VITALS

As mentioned previously, PostgreSQL is unable to support the complex queries and data types held within the OSM database; therefore, the previous VITALS team employed a Tile-Server running in parallel with the Software Suite. The Tile Server's responsibility is to transform the raw OSM data into the map imagery that applications, such as the VITALS Software suite, need to display and utilize. This tile server is not a storage container; moreover, it should be thought of as a rendering tool that parses the raw data into usable geospatial data.

Unlike traditional database engines, the tile server incorporates a rendering engine as well as a tiling engine, both of which operate over the spatial data stored within the PostGIS database. To begin, geographic features are queried and symbolized based on predefined style rules; these rules can be shown in the visualization section (buildings show up red, water shows up blue, etc) as defined by the previous VITALS team. After rendering, the tile engine splits the finished map into a multi-resolution pyramid of small square tiles, which can be stored or served as a query. The OSM tile server enables the VITALS software suite to request tiles incremental and on demand, rather than the need to load the entire tiled database in at once, increasing scalability, robustness, and efficiency for memory and queries.

PostgreSQL/PostGIS in the VITALS Architecture

Within the current VITALS code base, PostgreSQL and PostGIS serve as the primary systems for storing, querying, and managing the geospatial data imported from the OSM database. Because this data is both large in scale and computationally intensive to process, the previous VITALS team implemented a Docker-based deployment to ensure a modular, reproducible, and self-contained environment. This containerized approach allows the database subsystem to operate independently while still running in parallel with the rest of the VITALS

software suite. In practice, this setup consists of several sequential steps, which the previous team structured as follows:

1. Create a persistent database volume within docker to hold the database.
2. Mount the OSM database within the container created.
3. Create and store rendered map-tiles in a new volume titled “Tiles”.
4. Start the container running the tile server and expose the port.

The Docker environment established by the previous VITALS team is a simple yet efficient method for loading, storing, and serving the OSM data to the software suite. By using a few concise commands, the setup initializes two persistent Docker volumes and then exposes the HTTP endpoint for both map tiles and for PostgreSQL direct database queries. We do not think much needs to be changed within the current setup, other than to configure the whole setup into a ‘.yml’ container. Doing this would allow automatic startup of the PostGIS service and any other components needed to support the database.

Terrain Processing

Now that we have established an understanding of the underlying data, as well as how it is loaded and queried, we can begin discussing the processing workflow for this terrain data that the search algorithm will be performed on within the VITALS Software Suite. There are many layers to the current terrain pre-processing that are needed to be broken down in order to understand what is the terrain that is being used within the search.

Connection Establishment

To begin, when the system is first initialized they run a diagnostic module to check and see if the device that the software suite is currently running has a valid internet connection. This is done to ensure that the system is able to properly execute any database queries, API calls, or any other network-dependent operations. This safeguard is mandatory to prevent failed queries and timeouts caused by a lack of connectivity. This safeguard is done by running a function that verifies network integrity by attempting to establish a secure connection between the device running the software and Google’s public DNS server (IP 8.8.8.8, port 53). There are several

reasons that this is chosen as a reliable safeguard for the system: Google's DNS is extremely reliably worldwide and is almost always reachable, furthermore, the connection is a quick and lightweight connection that does not require sending or receiving data beyond the connection.

The current implementation of this function is simple, effective, and concise; it accomplishes the goal without unnecessary complexity and provides a reliable method to verify connectivity before executing any network-dependent operations. One optional improvement that can be implemented is to check several public DNS servers rather than relying solely on Google's public DNS. This improvement can ensure that even if one Public DNS server is down, we can still rely on the others to provide us with an accurate safeguard for the VITALS software suite initialization.

Coordinate System Standardization

Once the internet connection has been confirmed, the system proceeds to create queries from the database and perform other network-related operations. The first of these involves transforming the coordinates of spatial features retrieved from the database from EPSG:3857 (Web Mercator) to EPSG:4326 (WGS84 latitude/longitude).

The Web Mercator coordinate system is a projected coordinate system that represents the Earth's surface on a flat, two-dimensional plane; since the Earth's surface is a sphere, it causes a warped effect when trying to project a sphere onto a flat surface. OSM uses this projection because it allows for efficient and visually consistent tile rendering on web maps, allowing for fast map visualization and navigation. In order to combat the warped coordinates, we must transfer our coordinates from the projected Web Mercator coordinates back to the true lat/lon they represent. EPSG:4326 is a geographic coordinate system that represents locations on Earth using degrees of latitude and longitude. This step is essential because PostGIS expects geometries in EPSG:4326, and ensuring our agents receive true geographic coordinates rather than projected ones improves both spatial accuracy and downstream inference quality.

This transformation between EPSG:3857 and EPSG:4326 coordinates is done using the Transformer package within the pyproj library. Pyproj is a library that provides tools for

performing geographic and cartographic transformations between different Coordinate Reference Systems (CRS), such as the two we are dealing with here. The Transformer initializes a conversion pipeline between the two CRS definitions and applies the appropriate projection formulas to each coordinate pair as the data is pulled from the database. During transformation, it projects, or unprojects, the coordinates as needed, ensuring that the coordinate pair is accurately converted to degrees of lat/lon.

Several improvements can be made to enhance the efficiency and reliability of this transformation. Since large amounts of spatial data may be retrieved at once, implementing vectorization and batching would allow multiple geometries to be transformed simultaneously, improving speed and scalability. This key improvement could allow the VITALS software to run two different search areas in parallel and have the results remain accurate, with quick results. Another key improvement that could be made is simple error and safeguard handling. Currently, the file put in place assumes every coordinate entering will be of the form EPSG:3857, running a safeguard will transformations taking place can ensure that there are no duplicate transformations done and errors are handled accordingly when they arise. Finally, another key improvement could be controlling the precision of the return data from the transformation. Since the amount of data will be rather large, with the possibility of running multiple search areas at the same time, we can limit the data size of the transformed coordinates to save on space and computation power.

Spatial Index Construction

After the database has been established, and we are confident we can transfer coordinates to our proper CRS, the system can begin running inference and spatial queries on the geospatial data; however, we need a functional and clear way of storing the information to be able to access the specific indices required while running the search. To accomplish this, the prior VITALS team used the index library within the R-tree package. An R-tree is a tree data structure used for indexing multidimensional information such as geographical coordinates. The key idea of this data structure is to group nearby objects (data points) and represent them with their minimum bounding rectangle in the next higher level of the tree.

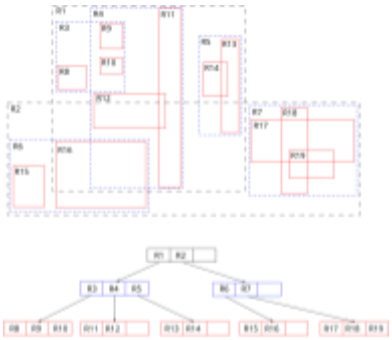


Figure 2.12: 2D R-tree

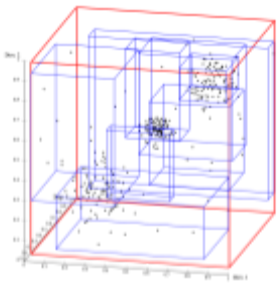


Fig 2.13: 3D R-tree

Similar to Numpy, the R-tree module in Python is a thin wrapper around libspatialindex, a C++ library that implements fast spatial indexing structures such as the R-tree used within VITALS. The Index class, provided by the index import within the R-tree library, provides us with support for core operations such as insert, intersection, nearest, count, and delete, enabling fast lookup of objects that intersect or lie near a given bounding box. Furthermore, this class can be configured to alter properties such as leaf size or variant type; however, by default, the data structure balances insert and search performance equally.

When the user is ready to define a search area for R-tree loading and processing, the workflow begins in the GUI.py file, which serves as the entry point for selecting and configuring the search region. The GUI.py file defines a variable that is then used as the parameter to define the search area for the terrain processing. Currently, my teammate Giorgio was tasked to review the GUI.py file and is planning on reworking the way that the polygon_points definition is done.

Currently, we can receive polygon points from the GUI.py just fine in the form of a list from the OSM database as Web Mercator points, and we begin running PostgreSQL queries from a library called alchemySQL. The file terrain_queries.py is responsible for generating spatial queries to gather map features from our PostGIS-enabled OpenStreetMap database that is running in parallel within our Docker container. These queries define a bounding box around the user-selected search region, retrieve only the features that match the specific search tags used, and then populate the R-tree defined previously. From there, the map is broken into grid tiles, and each tile is analyzed to determine the specific features it contains so that the agent can be

provided with the environment and plan viable search paths. A supporting file titled query_disambiguation.py is key here to support the queries in distinguishing the features being found within the tiles stored within the grid. The OSM database has a highway tag associated with each relevant data record, providing a raw classification that we use for downstream inference. This enables the system to convey more detailed contextual information to the agent about the type of environment the drone is operating within.

SQLAlchemy as Database Interface

To support the queries, the prior VITALS team chose to select a module known as SQLAlchemy (SQLA) as the interface to the PostGIS database. SQLA provides a clean way to build and execute database queries, without manually writing raw SQL every time a query needs to be done, while still allowing for precise control over database queries. SQLA acts as a high-level Object Relational Mapper (ORM), it translates Python calls into optimized SQL commands and handles database connections as well as managing data serialization to ensure that spatial features coming from PostGIS are returned as a structured Python Object. Furthermore, SQLA seamlessly integrates with the PostGIS extension for PostgreSQL, meaning spatial filtering, bounding-box queries, and geometry extraction operations can run directly inside the database engine where they are most efficient.

Currently, much of the terrain processing logic happens in Python after retrieving raw features from the database; A major improvement would be to push as much of this work directly into the PostGIS engine so that the spatial filtering, tag normalization, and feature counting occur inside the database rather than in application loops. This would include practically wiping out the query_disambiguation.py file and moving as much, if not all of it, to the database engine to increase the efficiency of our data queries. Furthermore, it also means that we will open up the option to using PostGIS spatial functions such as ST_Intersects, ST_Within, and ST_MakeEnvelope to generate tile grids, count gestures per grid cell, and run intersections all within SQL. PostGIS is optimized for exactly these types of spatial queries, meaning that this refactor of the query module would drastically improve the performance, especially as map regions and the number of features grow. The two main benefits of this would be transforming the PostGIS from a database into a true geospatial processing engine and taking a majority of the

heavy computation away from Python and into a faster and more efficient language, such as SQL.

Terrain Pre-Processing Summary

To provide a clearer understanding of the terrain pre-processing, the following figure (Fig #.#) presents the complete preprocessing pipeline utilized within the VITALS software suite. This flow chart illustrates each of the core states within the workflow, beginning with raw user inputs collected from the GUI, progressing through the bounding box generation, grid initialization, spatial feature extraction, and finally the grid population phases. Furthermore, the figure visualized the dynamic branching that determines whether OSMnx or PostGIS will serve as the data source, depending on safeguards put in place.

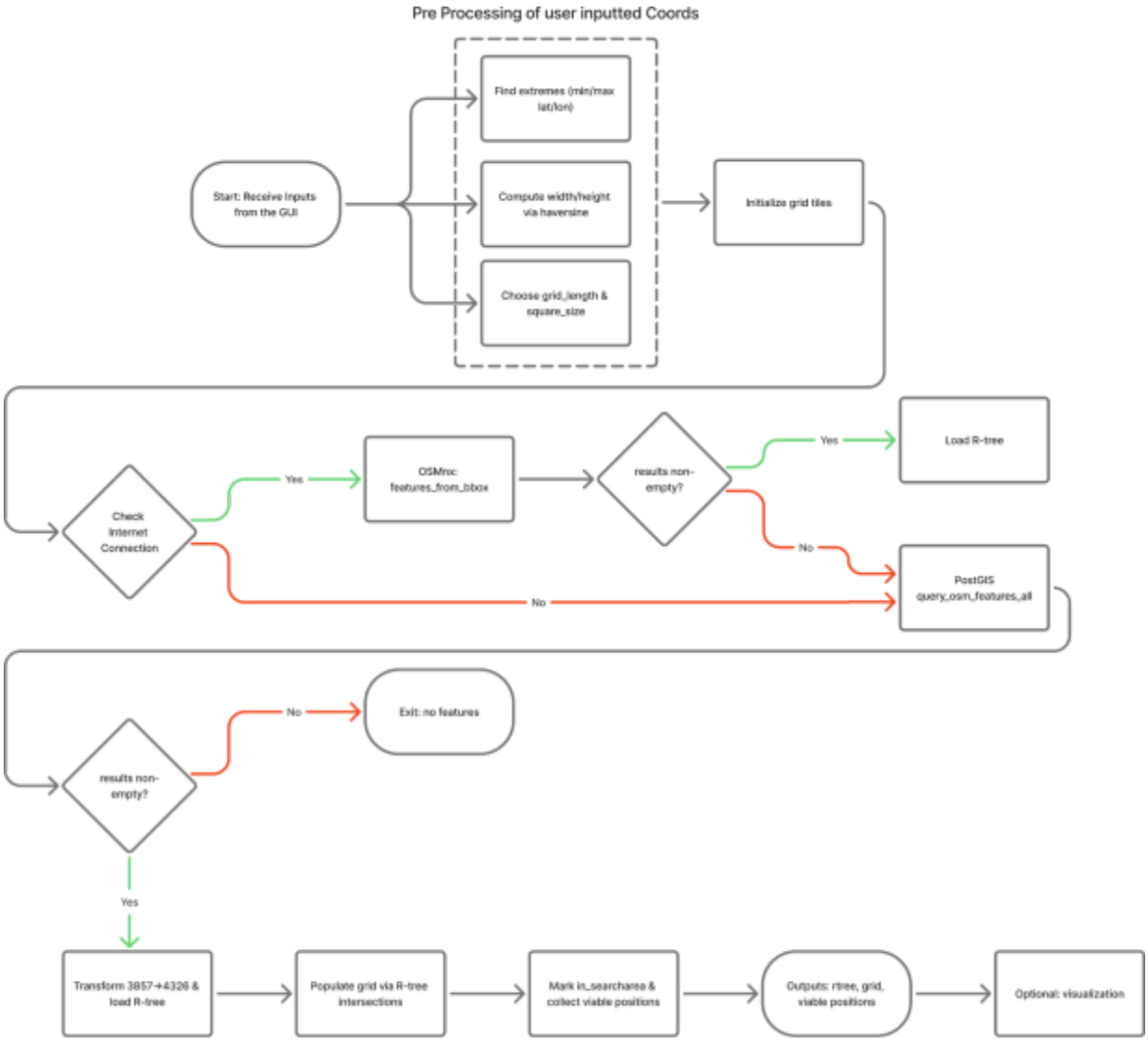


Figure 2.14: Terrain Preprocessing Pipeline within VITALS

The results of the pipeline are a fully constructed grid and R-tree ready for spatial indexing and downstream processing. At this point within the system, the terrain is now decomposed into discrete search tiles, each tagged with feature counts, positional flags, and geometric definitions that allow for efficient spatial indexing.

Way Point Generation

Now that we have established our data and how we are going to properly structure it, we can perform our search and path planning algorithm for the drones. The current way point algorithm is simple and adequate only for the existing simulation; substantial enhancements and extensions are required to meet the project's vision for high-quality, multi-drone path planning.

To begin, this module consists of an individual file that assigns search waypoints to multiple drones by ranking grid cells and distributing them across the drones. It takes in a 2D grid where each cell carries metadata (such as buildings, highways, forestry, etc.) and repeatedly selects the next best cell for each drone using a greedy score. For each chosen cell within the grid, it will then extract the coordinate corresponding to the centroid of that cell and append that coordinate to the corresponding drone's destination list, then remove that cell from the pool; therefore, no two drones visit the same spot. The grid and cell attributes come from the terrain pre-processing pipeline, which labels tiles with several key components:

- In or Out of bounds
- Tile target Intensity
- Geometry

The main task of this file, provided with the following components, is to focus on task allocation. By using the pre-computed target intensities, it ensures that the drones visit the most valuable cells first, then uses the simple proximity via Manhattan distance to reduce travel between consecutive picks. This file is limited to waypoint generation; its role ends there. The subsequent path-planning stage is where my planned improvements and extensions will begin.

Opportunities for Improvement

Before talking about the path planning, there are several improvements that can be made to the waypoint generation itself to improve efficiency and accuracy, ensuring higher-quality

inputs for the forthcoming path-planning algorithm. To begin, one major improvement that can be made is to ensure that one drone does not monopolize the high-priority tiles. To accomplish this, we can implement something like a priority queue, round robin, or Hungarian-style assignment to ensure that each drone is being assigned an appropriate number of high-priority tiles, and it is not being dominated by a single drone. Building on this idea, we can also implement methods to take the drone's physical constraints into account (such as battery range or current battery life) while path planning to ensure that drones are not assigned paths that they are not able to complete due to insufficient battery life. Furthermore, another improvement we would like to make is shifting the tile-selection strategy from a purely greedy approach to something closer to a nearest-neighbor method. In its current form, greedy logic is not ideal for multi-drone coordination. For example, if two high-priority tiles happen to be far apart, then the drone may constantly jump back and forth across the grid to chase the highest score. By instead prioritizing the next closest valuable tile, we can dramatically cut down on inefficient zig-zag movement, reduce flight time, and create a more realistic mission path for the drones.

Finally, to prevent loss or damage, the system should support continuous terrain processing so that newly observed features can be immediately integrated into navigation decisions. Because VITALS is designed for deployment on edge devices in real-world field environments, temporary structures, uncharted obstacles, and other unexpected conditions will inevitably arise that are not represented in the original OSM dataset. Incorporating this real time information is essential to ensure that each drone adapts safely and that the system can intelligently route assets toward emerging points of interest. This also introduces the idea that the importance of a tile may evolve over time. A tile that once had a high search priority might become less important, or vice versa. Incorporating time-aware tile updates allows the system to continuously refine its search plan, ensuring that drones are always focused on the most relevant tiles in dynamic conditions.

Path Planning

Currently, the module produces an ordered list of centroid waypoints per drone, chosen greedily by a priority-minus-distance score; however, the missing piece to make this into a true path planning algorithm is the inference layer that defines how the drone safely gets to its

destination. Taking into consideration safe, dynamic, and obstacle-aware paths that connect to the waypoints is an important first step to creating a true path planning algorithm that can work in a real time situation with unknown changing terrain.

To keep concerns separated, the waypoint generator should serve strictly as a guidance source, assuming an ideal environment. The path-planning module then refines each step (adjusting headings, velocities, and intermediate points) to ensure the drone can traverse the route safely and efficiently while reacting to newly detected hazards being encountered. Creating this separation between the waypoint generation and path planning keeps each layer simple, testable, and maintains a modular software suite for future expansion and scalability.

Global vs Local Planning Layers

Furthermore, another implementation that would increase the robustness and reliability of the path planning algorithm would be to utilize both a global planner to connect waypoints while also relying on a local planner to avoid obstacles and produce smooth paths. For a global planner, we will start with a simple robust path planner, such as A*, to provide a basis for the big picture path planning of each drone. The global stage is responsible for producing an optimal large-scale route that covers the intended search region, prioritizing POIs, buildings, and other areas indicated by the user.

From there, we will work on layering in a local avoidance planner to help produce smooth, collision-free paths around obstacles that were either provided within the original data or detected during the drone's flight. The purpose of the local planner is to take the global path as a reference and keep you close to it while guaranteeing dynamic separation but maintaining the path. The local planner helps to deal with hazards such as moving entities, temporary structures, and uncharted obstacles while keeping the drone en route to its intended path. This method will ensure that each drone is able to hit its waypoints and dynamically adjust to anything that it may encounter. This layered architecture is essential for realistic deployment to ensure the drones follow an optimal high-level macro route while retaining flexibility and safely adapting to any encountered obstacles.

Preventing Route Overlap

Additionally, an implementation that could be taken into account is to avoid route overlap between drone paths. Instead of layering paths so each drone searches the entire area, we are proposing partitioning the grid into non-overlapping sectors and assigning each of the drones to a sector. By segmenting the terrain, we can then assign time stamps to each segment keep a record of when each one was searched by the drone. When a drone completes its sector, we can reassign it to another region with the oldest timestamp. By time-stamping sectors and rotating drone assignments, the system maintains persistent coverage, prioritizes areas that have not been visited recently, and increases the chance of detecting features in a dynamic and robust environment that might have been missed on the first pass.

Visualization

Finally, the Visualization.py file renders the processed terrain grid and the waypoint trajectories derived from the search. Below are example outputs illustrating both the terrain visualization and the corresponding drone waypoints:

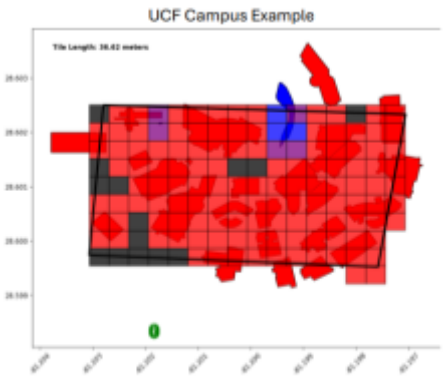


Figure 2.15: Terrain Pre-Processing Visualization

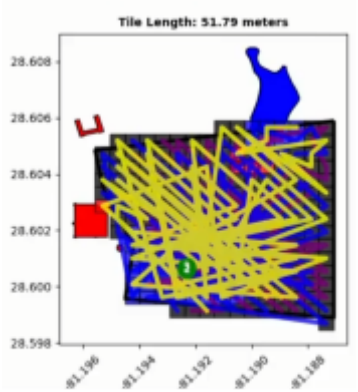


Figure 2.16: Waypoint Visualization

When the visualization first initializes, it loads the PostGIS geometries and shades them using specific colors. If a feature belongs to multiple map categories, its color is blended to reflect all the types detected, and then it adds the tile to the grid. If the user draws a polygon to represent the mission search area, the outline is added as another dedicated layer (as shown by the black outline on the edge). All of these elements are organized into internal dictionaries that track which artists belong to which visualization layer. The system then auto-generates toggle

buttons for specific layers, such as geometry, grid, and outline, allowing the user to simultaneously turn visibility on and off.

Furthermore, the file also builds a dedicated interface for drone visualization. When a drone's path or live data is provided, each drone gets its own interactive controller, allowing the user to toggle its visibility and text boxes. This interface allows the user to scrub through a route, viewing only certain segments, creating a more interactive and robust solution for human interaction within the system.

Altogether, this visualization.py file acts like a very lightweight mission-control GUI that bridges the processed environment data with a highly interactive map, with the support built in for multi-drone playback and live mode. We do not feel that this is a focus of mine, especially since my teammate Giorgio is focusing heavily on improving the VITALS GUI built as an extension for Mission Planner for Ardupilot. We believe that the improvements needed for this file will be all incorporated into the GUI file and that the terrain pre-processing engine should output the data to that file for visualization.

LangGraph - LLM Communication and Toolset

The LangGraph subsystem within the VITALS software suite is responsible for building the framework for the agents tasked with coordinating the drone swarm for VITALS. Currently, there is a single agent set up managing all coordination logistics throughout the entire system; however, as detailed within the design section, we believe that breaking this up into a multi-agent architecture will improve the accuracy and efficiency of the agents.

One major challenge with trying to condense the responsibility of several specialized agents into a single agent is the rapid saturation of its context window, which can then lead to confused behavior and failures. The initial system prompts define the agent's core objects; however, additional incoming sensor data, updates, and drone reports consume available context space. Given the number of drones communicating at once with the single base station, we can expect the context window to fill quickly within a real mission scenario. It is essential to

preserve a clean, relevant context window at all times for the agents to operate reliably and accurately.

Flight Planning & LLM Integration Module

The Flight Planning & LLM Integration Module serves as the coordination layer between the autonomous routing and the LLM components that assist with planning the drone swarm. This module's primary responsibility is to orchestrate the flow of data required to support real time waypoint generation, drone behavior, and mission-level command responses. Within the pipeline, this module receives geospatial data, metadata, and imagery (depending on the job it is assigning) captured in drone flight and transmits this information to an LLM subsystem running within Ollama. This module enables the LLM to augment traditional path-planning approaches rather than relying on a solely deterministic routing algorithm. Using this subsystem, we are able to generate smarter waypoint sequences that respond dynamically to terrain classification, image descriptions, and mission constraints by having the LLM run inference on mission-critical data.

Call LLM

The current file contains a main routine titled ‘call_LLM’, which is the main routine within the file. This routine is an execution method used to invoke the LLM endpoint by formatting input parameters into a JSON according to whichever model endpoint is in use. Once the model responds, the routine extracts the relevant fields from the response and returns them in a standardized format in the form of the ‘Job’ class setup within the code base. This is currently the VITALS standardized way of formalizing routines being carried out by the drones.

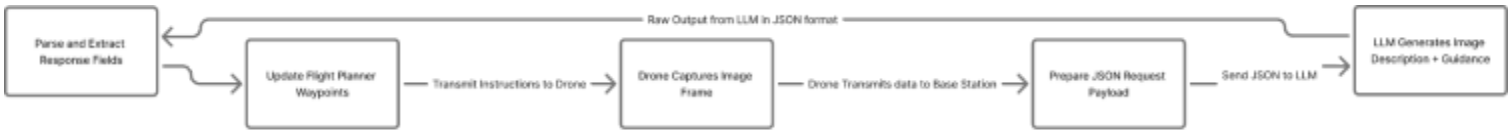


Fig 2.17: LangChain FlowChart

Job Class

The Job class, currently defined within both missionState.py and GUI.py, is VITALS’ way of formalizing the instruction set provided to a drone within the swarm. After the above

LLM module provides instructions to a drone, they are all served to the rest of the system in the form of the Job Object. The purpose of the Job Class is to act as a structured representation of a mission request, encapsulating all request information for execution and tracking.

Each Job instance contains the job type, its currency status, the ordered list of waypoints associated with that assignment, the mission state that produced that job, and the priority assigned to that Job by the LLM. When a Job is created, it is automatically assigned a unique ID linked to a MissionState Object. The purpose of this is to ensure that each task can be tracked and referenced across the whole system to a corresponding mission state object. Finally, a very unique and key element of this class is the inclusion of the custom comparator that inverts the comparison behavior so that higher priority tasks are popped first when stored within the min-heap data structure. This is the key element used within the VITALS system to control job scheduling based on job priority.

Image Description

The subroutine within this file is the image description routine, which focuses on converting raw drone images taken from the camera into LLM descriptions suitable for analysis and reasoning. These components leverage the LLM model titled ‘LaVva’ to interpret visual content from the drone and translate that into contextual understanding. LLaVA (Large Language and Vision Assistant) is a large multimodal model that combines a vision encoder and an LLM to enable general-purpose visual and language understanding. The descriptions produced allow the flight system to make deductions that conventional computer-vision pipelines cannot, thanks to the increased inference from the LLM.

Search Coordination Agent Module

The search coordination module (langGraphMain.py) is responsible for defining a tool set allowing the agent to enrich its context window by utilizing specific queries if needed. Rather than hard-coding all the decision logic directly into a prompt, this module exposes a curated collection of functions defined by the programmer that the agent can invoke when additional information is needed. Each tool is wrapped in a standardized signature and description so that the LLM decides when and how to call it during reasoning and effectively turns the backend capabilities into callable actions within the agent’s chain of thought.

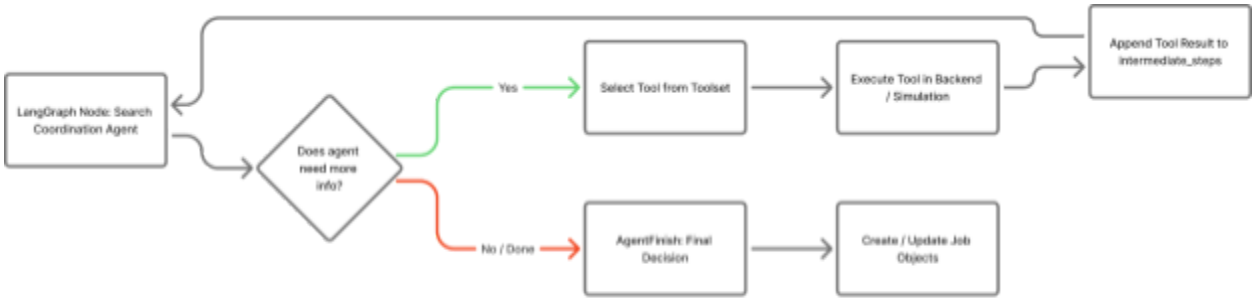


Fig 2.18: LangGraph FlowChart

Agent State Class

Currently, there is a class defined to keep track of the agent's state. This class has three class variables used to define what the agent takes in, outputs, and the steps used to decide the agent's output decision. This class is responsible for maintaining the information needed for the agent to process user input and the raw data from the drones. Although these values are presently hard-coded within the code base for early development and prototyping purposes, they are intended to evolve as the agent architecture changes. The following variables are what is used:

1) **Input: (str)** - The input prompt used to instruct the agent.
    a) Currently, these are all hard-coded within the codebase and will be adapted as we evolve the agent architecture.
2) **Agent_out:** (Union[AgentAction, AgentFinish, None]])
    a) The AgentAction and AgentFinish are both imports from the langchain, which is an agent's library created to help create functional agents.
        i) AgentAction represents a single step in an agent's reasoning process
        ii) AgentFinish represents the agent's conclusion.
3) **intermediate_steps:** Annotated[list[tuple[AgentAction, str]], operator.add]
    a) As stated previously, the Agent Action defines a single step of the reasoning process; therefore, compiling them into a list showcases the agent's chain of thought as it chooses its conclusion.

The AgentState class serves as the foundational structure for representing the agent's reasoning as the mission state evolves. We are going to use this class to maintain the input,

output, and intermediate steps of the agent's reasoning, allowing the system to track the conclusion and logic flow.

Current Tool Set

Within the agent pipeline, there is a collection of tools allowing the agent to interface with both the simulated drone environment and the backend systems. These tools provide discrete operational capabilities that the agent may invoke to gather information, issue commands, or evaluate observations. Furthermore, each tool exposes a single well-defined function that contributes to the broader coordinates. The following tools are currently set up:

1) Find_available_drone

   A tool used to search through the active drone registry and return the first available drone capable of taking on a user or agent-input task.

2) Get_poi_location

   A tool that retrieves the stored geographic location of a specific POI using its name as the parameter. This tool allows the agent to determine whether a POI is actionable.

3) Command_drone

   A tool that sends a direct command to a specific drone. It represents the primary mechanism through which the agent deploys the drones.

4) Terminate_search

   A tool that terminates an ongoing mission by recalling all drones back to the base station location.

5) Object_identification

   A tool that processes an image using a YOLO model and helps classify threats, hazards, or targets within the image.

6) Rate_object_priority

   A tool that takes the information from the output of the object_identification model and outputs the priority rating for the detected item.

7) Calculate_gps_coordinates

   A tool that takes a detected object's screen space position within an image and computes its corresponding GPS coordinates using environmental metadata.

8) Send_poi_to_backend

   A tool that sends a full structured POI to the backend, which includes the following information:

   - POI name: str,
   - POI desc:str,
   - search_priority: float,
   - poi_location: tuple[float, float]

Proposed Improvements

The current subsystem is functional, but we propose a restructure of the current singular agent to a multi agent pipeline with clean context boundaries to better control memory management. This includes migrating to specialized agents separated by responsibility, implementing stateful execution through LangGraph, refining tool I/O schemas, improving telemetry injection, and integrating both short-term and long-term memory layers for increased stability of the agent architecture. Below is a low level breakdown of each element and how we plan on better integrating these improvements:

Job Class

The current Job class that is set up within the MissionState file is a very simple and sophisticated way of keeping track of the Jobs while keeping in Sync with corresponding Mission State Objects using Unique IDs. However, we feel that the big issue is that we need to redefine this class to encapsulate both Job Class definitions that are defined within the MissionState file and within the GUI file. Having two classes with the same name doing the same function is redundant within the code base, and that is going to be phase one of vitals: removing redundancy.

LangChain

LangChain is a framework that enables us to build LLM agents that can call tools, remember context, and pass data to each other using the MCP and ACP protocol. Within our A-D architecture, LangChain gives us the building blocks allowing us to define each agent within its separate chain of prompts, tools, and memory.

LangGraph

LangGraph extends the framework created with LangChain by allowing us to treat the agents as nodes in a stateful graph. This furthermore allows us to treat the edges of these graphs as the flow of information and further visualize the flow of information. We recommend transitioning the entire multi-agent logic to a LangGraph implementation, introducing explicit edge conditions showing the flow information from agent to agent. Furthermore, this can allow us to introduce items as branching pathways, merging nodes, and other conditions later down the road if it so calls.

General Tooling Improvements

Currently, the entire tool set is all structured under a single agent; ideally, we need to separate the tools into their appropriate files such that each agent has access to its tools it needs to make appropriate and accurate inference and nothing else. For example, only Agent A needs to use the tool Object_identification, while the other Agents B-D need their own unique tool sets defined to carry out their goals. Since we are formalizing the agent communication protocols using ACP and MCP, we also propose formalizing the tool set provided. Some improvements we propose are as follows:

- Formalize tool Schemas
- Add Validation, Result Standards, and Structured Return Contracts
- Add dynamic telemetry prompt augmentation
- Add Token Rate Limit Protection
- Add Tool Specific error codes and Logging

missionState.py

The Mission State module (missionState.py) serves as the central orchestration layer of the VITALS software system, functioning as the critical bridge between the graphical user interface, MAVLink communication protocols, and autonomous mission logic. This module implements a state management architecture that maintains comprehensive awareness of all active drones, mission parameters, job assignments, and points of interest while coordinating the complex interactions required for multi-drone search and rescue operations.

Architectural Overview

The missionState.py file implements a hierarchical object model consisting of four primary classes that work in concert to manage mission execution: the Drone class, the Job class, the POI class, and the missionState class itself. This architecture follows a composition pattern where the missionState maintains collections of Drone and POI objects, while each Drone maintains its own priority-based job queue. The design allows for independent evolution of each component while maintaining clear communication pathways through well-defined method interfaces.

The module's responsibility extends beyond simple data storage; it implements the business logic that governs how the system responds to events such as new detections from the computer vision pipeline, operator commands issued through the GUI, telemetry updates received via MAVLink, and mission state transitions reported by the drone fleet. By centralizing this coordination logic within a single module, the architecture avoids the distributed state inconsistencies that would arise from having multiple components independently modifying mission parameters.

Drone Class Implementation

The Drone class encapsulates all state and behavior associated with an individual unmanned aerial vehicle within the swarm. Each Drone instance maintains comprehensive telemetry data including GPS position (latitude, longitude, altitude), relative altitude above home position, heading, velocity components (vx, vy, vz), and attitude information (roll, pitch, yaw).

The position data is stored in MAVLink's native integer format (degrees multiplied by 1e7) to maintain precision and avoid floating-point accumulation errors during long missions.

Beyond telemetry storage, the Drone class implements a job management system through its integrated jobPriorityQueue. This queue uses Python's heapq module to maintain jobs in priority order, with higher-priority tasks automatically bubbling to the front of the queue. The class exposes methods for adding new jobs (addJob), managing the active job (setActiveJob), and handling job state transitions (setJobComplete, setJobFailedUpload, pauseJob). The job management logic implements a preemption policy where new jobs with priority values 3 or more points higher than the current job automatically interrupt execution, allowing urgent tasks such as target investigation to override lower-priority search patterns.

The class maintains references to both the parent missionState object and the GUI, enabling bidirectional communication. When telemetry updates arrive, the updatePosition method both stores the new data internally and triggers a GUI refresh through gui.updateDronePosition(). This callback pattern ensures the interface remains synchronized with the actual drone state without requiring the GUI to poll for changes. The Drone class also tracks important operational parameters such as operating altitude (configurable per drone) and the vision model path, allowing individual drones within the swarm to operate with different detection configurations based on their specific missions or hardware capabilities.

Job Class and Priority Queue System

The Job class represents discrete mission tasks assigned to individual drones, encapsulating the complete specification of what a drone should accomplish. Each job contains a job type descriptor ("Initial Search", "Investigate POI"), a status field tracking execution state ("pending", "loading", "Running", "Completed"), a list of waypoints formatted as (latitude, longitude, altitude, waypoint_type) tuples, and a priority value determining execution order. The job_id field provides unique identification for tracking and logging purposes, automatically assigned by incrementing the missionState's jobIDCounter.

The waypoint_type field within each waypoint tuple enables the system to specify special MAVLink commands beyond simple navigation. A value of 0 indicates a standard waypoint, 1 designates a takeoff command, and 2 specifies a loiter turns command that causes the drone to circle a location for investigation. This flexibility allows the path planning algorithm to construct complex flight patterns including takeoff sequences, search grids, loiter patterns for POI investigation, and return-to-launch procedures all within the unified job framework.

The jobPriorityQueue class implements a min-heap-based priority queue using Python's heapq module. Jobs are inserted with their priority values negated, effectively converting the min-heap into a max-heap where higher-priority jobs sort first. The queue exposes three primary methods: add_job() for insertion, get_next_job() for removal and retrieval, and peek_next_job() for inspection without removal. The is_empty() predicate allows efficient testing for queue exhaustion. This priority queue architecture enables the system to dynamically adapt mission plans, with new high-priority tasks immediately available for execution once the current job completes or is preempted.

Point of Interest Class

The POI class manages the lifecycle of potential target detections, from initial computer vision identification through operator validation and potential confirmation. Each POI maintains geographic coordinates, a descriptive name, an automatically generated caption from the LLaVA vision-language model, a status indicator, and a type classification. The class tracks a positive flag counter that increments each time a drone confirms the presence of the target, implementing a confidence threshold system where three confirmatory detections trigger the target found workflow.

The poi_target_at_location boolean flag prevents duplicate alerts, ensuring operators are notified only once when sufficient positive detections accumulate. When this threshold is reached, the target_found() method creates a modal popup dialog that blocks mission continuation until the operator makes a critical decision: end the mission immediately (appropriate when the search objective is a single missing person) or continue searching (appropriate when multiple victims may be present or when the initial detection requires ground team verification before mission termination).

The POI class maintains integration with the mission's file storage structure, automatically creating directories at ./Missions/{mission_id}/POIs/{poi_id}/ for storing detection images. Each time the computer vision system captures an image associated with the POI, the annotated detection is saved to this directory with a timestamp-based filename. This archival system provides critical documentation for post-mission analysis and creates an evidence chain for search and rescue operations where legal or investigative review may be required.

MissionState Class: Central Coordination Layer

The missionState class serves as the single source of truth for all mission data, maintaining authoritative collections of active drones (self.drones list), detected points of interest (self.pois list), and global mission parameters. The class implements the coordination logic that connects the GUI's operator input, the Dispatcher's MAVLink communication, the terrain preprocessing engine's grid generation, the path planning algorithm's waypoint generation, and the computer vision pipeline's detection events into a cohesive operational system.

Initialization of the missionState establishes the foundational communication infrastructure by instantiating the Dispatcher object and initializing the asyncio event loop that enables asynchronous MAVLink message processing. The connect_to_mavlink() method delegates to the Dispatcher's connection routine, establishing the TCP link to Mission Planner and initiating the continuous packet reception loop. The successful connection triggers a cascade of initialization activities including drone discovery (as HEARTBEAT messages arrive), mission parameter configuration, and GUI state updates that enable operator interaction.

The drone management methods maintain the drone collection in a consistent state, creating new Drone objects as vehicles are discovered and routing telemetry updates to the appropriate instances. The position update pathway is particularly critical, as it not only stores the latest GPS coordinates but also triggers the detection point proximity checking logic for simulation mode. When a drone approaches within 20 meters of a manually placed detection

point, the system automatically invokes the computer vision detection workflow, simulating the behavior of real object detection for testing purposes.

Conclusion

The missionState module exemplifies a well-designed coordination layer that successfully abstracts the complexity of multi-drone mission management behind a clean object-oriented interface. Its architecture balances the competing demands of modularity (enabling independent evolution of components), coordination (maintaining consistent system-wide state), and performance (avoiding unnecessary data copying or excessive callback overhead). The clear separation between data storage (Drone, POI, Job classes), business logic (missionState methods), and external interfaces (GUI callbacks, Dispatcher delegation) creates a maintainable foundation that has successfully supported the VITALS project's evolution from initial prototype to functional search and rescue platform.

The module's design demonstrates several software engineering best practices including encapsulation of state within appropriate classes, use of priority queues for intelligent task scheduling, callback patterns for loose coupling between components, and comprehensive error handling for robust field operation. While opportunities for enhancement exist such as implementing more sophisticated job preemption policies, adding formal state machine patterns for mission lifecycle management, or introducing event logging for forensic analysis the current implementation provides a solid foundation that meets the MVP requirements while remaining extensible for future development phases including the proposed multi-agent architecture integration.

Installation Requirements

Looking into what is required for the software to run locally on a Windows machine, there are three major components:

1. Mission Planner
2. Python Packages
3. Ollama

To first make the connection between the VITALS software and the ground control station,

Mission Planner would be installed, which also includes the SITL firmware for the multirotor drone swarm. Since this installation would be on a Windows machine it is already assumed that the .NET framework is already installed, as well as X11 capabilities, which are required for Mission Planner to run properly

In order for the VITALS software to now run in conjunction with Mission Planner, the Python dependencies specified in app/requirements.txt would also need to be locally installed using the python package manager pip. The dependency file for this project can be further split into five portions:

1. MAVLink
2. GUI
3. Path planning & Geospatial
4. Database/PostGIS
5. LLM & Computer Vision

Right on top of the file lies the MAVLink communication package, pymavlink. This package is responsible for ensuring the software is able to communicate via MAVLink, utilizing key modules such as mavutil and mavwp, which manages communication links, and waypoints, respectively. Included in the file are other crucial packages such as tkintermapview, a GUI package responsible for creating the embeddable map widget, or Ollama, tying in the local LLM to the software are needed to satisfy the minimum viable product.

After installing everything necessary for communication between core modules and Mission Planner, installing Ollama would now make it so that a local LLM can be run as our Agent C, which is meant to be the reasoning aspect, all while being offline. The last install would then be an image recognition agent, and for this project it was inherited to be LLaVA.

Platform Support (Windows vs. Jetson/Linux)

Since this application would be developed and deployed on a Docker container, VITALS would be capable of running on any Linux platform natively, as long as the hardware specifications are met. Docker utilizes its host system's hardware in order to run the container. The communication between container and hardware is done by the Docker daemon which

communicates with the host system for any processing needs. As for storage, the host machine allocates where the container can hold its memory, which is called a c-group, and the Docker daemon communicates with that shared space. For Windows and Mac, a Linux VM is required, which is typically done by using Docker desktop, as it is capable of not only running the container on its own VM, but is able to pull the image from a central source akin to GitHub.

In order for the VITALS software to run, we use Mission Planner, which as mentioned before is responsible for sending the MAVLink packets from the GCS to the drone swarm. Since Mission Planner runs natively on Windows using the .NET framework this causes issues with the plan to have the software suite running Linux on the Jetson Orin. In order to circumvent this issue, the container would be fitted with Mono, which acts as a .NET emulator for Linux. On top of this, in order for the Mission Planner GUI to display on the container, an X11 server must be running as well. X11 forwarding software like X-server can be used to allow for graphical interface capabilities in a client-server model, which would also be responsible for user input as well.

Docker

In order to run the software as lightweight and portable, Docker would be the core to hardware-software integration. Using Docker would make deployment of various agents modular, for example, the path planning part of the software would be able to run on a separate container than the computer vision, making dependencies on each agent limited. This reduces conflicts that may arise in the use case of this project. A way to make this isolation and modularity is with the docker compose tool which organizes the agents as mentioned before. Furthermore, utilizing the Docker compose tool can make communication between agents streamlined. For example, Agent A would make its detection points, then communicate to Agent C that makes a decision on how the swarm would act, then communicate with the UI agent to update the user, then communicate with the MAVLink module to make the drone swarm commit to its next move. Another perk to using Docker would be the cross compatibility of architectures, since the Jetson products typically use ARM64, while laptops use x86, this can cause compatibility issues, but with Docker, the same containerized code can be run on both architectures.

Other than being used in production, Docker is also imperative to the development process. Having a central location to make changes and updates to the software, as close to what the software integration would look like creates a productive environment in a team setting. By doing this, we can also separate the agents into their own containers in order for the development of each agent to be independent of each other. Fine tuning the functionality of each by creating their own test cases would allow for this independent development to come together seamlessly

Interfacing with Hardware Devices

Since we are using the Jetson Orin as our hardware device, we would be interfacing mainly with a Docker container fitted with Mission Planner and the VITALS software in tandem. The plan for a user to interface with the Jetson would be using a keyboard and monitor. We would configure the VITALS software to be easily accessible with only a keyboard, while showing the GUI which includes telemetry data, mission visuals, and LLM interaction. This setup would be contained in a briefcase for easy deployment in use case situations. Other hardware additions would be a power supply, transmitter and receiver for powering the Jetson and communicating with the drone swarm respectively.

Networking/Protocol Configuration

Model Context Protocol (MCP):

MCP defines how perception and contextual data are represented and shared. Each MCP message encapsulates detection results or operator inputs as structured JSON.
Every fact generated by Agent A is indexed by Agent B as a vector embedding, forming the basis of the RAG memory store. This integration allows later retrieval of perceptual data, effectively linking sensory input to long-term context.

Agent Communication Protocol (ACP):

ACP governs coordination between reasoning and planning components. Messages describe intents, constraints, and goals, while remaining independent of execution details. Because these intents may reference prior context retrieved through RAG, ACP acts as the translation layer from contextual reasoning to concrete action. By decoupling “what to do” from

“how to do it,” ACP enforces clean separation between decision logic and execution logic, improving maintainability.

MAVLink

MAVLink provides the bridge to physical hardware. It handles telemetry, mission uploads, and control messages between Agent D and UAVs. MAVLink’s maturity and widespread adoption make it ideal for VITALS, ensuring compatibility with many commercial and open-source flight controllers. In the VITALS context, it represents the execution layer of the architecture.

On this channel the data follows a particular series of bytes arranged into its own MAVLink packet, which includes a 6 byte header, 255 byte payload and a 2 byte checks. Looking into how the packet is structured, in the header, the bytes are split up into a system ID, component ID, and message ID. The system ID represents which system is sending the packet, whether that be the ground control station or the drone swarm. Meanwhile the component ID corresponds with the individual components of a system’s hardware, for example the ground control station would be designated as 1, and a specific drone in the swarm would be designated some other number as its ID. This allows the packet to be specific as to where its payload is being dropped off. After the component of the system has been identified, the message ID of the header is then responsible for specifying the length of the payload, and the sequence number, which makes sure that the drone receives the right command in the right order. The payload is pretty simple holding the actual instructions, which can be as large as 255 bytes.
The combined use of MCP, ACP, and MAVLink provides a layered communication hierarchy analogous to the OSI model: Perception → Reasoning → Actuation. Each layer has a distinct data model and reliability requirement, minimizing cross-coupling and simplifying debugging.

The combined use of MCP, ACP, and MAVLink provides a layered communication hierarchy analogous to the OSI model: Perception → Reasoning → Actuation. Each layer has a distinct data model and reliability requirement, minimizing cross-coupling and simplifying debugging.

Running VITALS

Upon inheriting a previous project's repository, some work had to have been done understanding and checking extensively for bloat that may come from the dependencies. As this project continues the installation requirements are bound to change. Install required software:

1. Python 3.11.9
2. Ollama
3. Mission Planner

Installing the dependencies (Windows 11):

In PowerShell navigate to the VITALS repository and type:

```
cd app
```

```
pip install -r requirements.txt
```

```
Ollama pull llama3.2
```

```
Ollama pull llava
```

Once those initialization steps are complete we can begin launching VITALS.

Launching Mission Planner

The current state of the app is able to run on top of a simulation created using Mission Planner. In this simulation the drone object uses a multirotor firmware running on a local host. The drone is then connected to the VITALS software via TCP. Once this TCP connection is made, the software is able to now communicate with the simulated drone using MAVLink and vice versa. The following steps will create the TCP server in order to allow the VITALS software to connect to the simulated drone:

1. Open Mission Planner on your windows machine
2. Navigate to the "Setup" heading found on the upper left corner navigation bar
3. Click on "Advanced"
4. Now to "Mavlink Mirror"

5. Fill the following information out and ensure that the server has “Started”

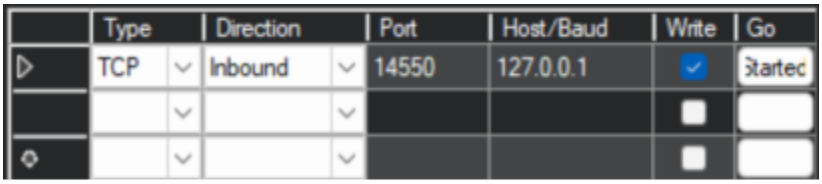


Figure 2.19: MAVLink Mirror TCP Config

6. Once the server has started, navigate to the “Simulation” tab which would be located on the upper left corner of the screen.
7. On this screen, you should be able to find four options under “Please select a firmware to run”, to which you click “Multirotor”. This will now download the SITL (Software in the loop) firmware for the simulation to start.
8. After the download is finished, the software will then automatically connect via TCP to the server that was started in previous steps. When that connection is made, you should end up with a multirotor drone icon on the screen in the location set by default, along with live telemetry data shown to the left.

Launching VITALS

Now that we have successfully installed all our dependencies and created an open TCP MAVLink mirror in Mission Planner, we are ready to start VITALS and begin the mission. Follow the steps outlined below to launch the software:

1. Change your directory to the app folder
2. Run missionState.py
3. You should now be met with a screen asking you to select a mission type. Select simulation and the main VITALS GUI will appear.
4. Click “Connect to MAVLink” in the top left.
5. Once connected the design polygon prompt will appear in the chatbox and you can begin the mission.

Regulations

Our team will always check federal and local guidelines to ensure we are safely flying and testing our drones. The previous team did a great job at researching these regulations and how they pertain to VITALS. The following is a list of regulations that we will be following.

Federal

Drone Registration

Every drone weighing more than 0.55 pounds (250g) must be registered with the FAA. Registration ensures accountability, requiring operators to display the registration number on the drone. The process is straightforward: operators register their drones online via the FAA DroneZone platform, paying a small fee. Failure to register can result in fines and potential confiscation of the drone.

Visual Line of Sight (VLOS)

Operators are required to keep their drone within direct visual contact throughout the flight. Advanced testing, such as flying beyond the operator's line of sight (BVLOS), requires a waiver from the FAA. These waivers are particularly important for operations like search and rescue missions, which often involve testing drones in larger or blocked areas.

Altitude and Airspace Restrictions

Drones cannot fly above 400 feet above ground level (AGL). Additionally, operators near controlled airspace, such as the area surrounding Orlando International Airport, must use the Low Altitude Authorization and Notification Capability (LAANC) system to obtain real time flight permissions. For example, testing a drone within 5 miles of the airport would require prior authorization through LAANC.

Flights Over People

Under Part 107, drones cannot fly over non-participating individuals unless the operator obtains a special waiver. This rule is particularly relevant near UCF, where outdoor events and

crowded areas are common. Operators must carefully plan their flight paths to avoid unintentional violations.

Night Operations

Although Part 107 limits drone flights to daylight hours, twilight operations (30 minutes before sunrise or after sunset) are allowed if the drone is equipped with anti-collision lights. Full night operations require an additional waiver, which involves showing how the operator will maintain safety during reduced visibility.

Waivers and Exemptions

FAA waivers allow operators to bypass specific Part 107 restrictions for advanced testing scenarios. For instance, a researcher testing autonomous drones for SAR missions may apply for waivers to operate at night, fly BVLOS, or carry external payloads.

State

Florida has implemented additional state-specific regulations to address privacy concerns, public safety, and the protection of critical infrastructure. Florida's House Bill 1027 pre-empts local governments from regulating drones, meaning only the state legislature can enact laws governing drone use. However, cities and counties may impose ordinances related to public safety, harassment, or privacy violations. Key statewide regulations include:

Critical Infrastructure

Drones are prohibited from flying over critical infrastructure such as power plants, water treatment facilities, and communication towers without prior permission. Testing near these areas requires special permits and coordination with authorities to avoid violations.

Privacy Concerns

Florida’s Senate Bill 766 prohibits drones from capturing images or videos of individuals on private property where they have a reasonable expectation of privacy. This law applies even when the drone is flying over public airspace. For testing conducted near residential areas or private property around UCF, operators must ensure they are compliant with privacy regulations.

State Parks and Forests

The Florida Administrative Code restricts drone flights in state parks, forests, and other managed lands. Testing conducted in these areas requires special authorization from the Department of Agriculture, which evaluates the potential impact on wildlife, forest resources, and public safety.

Local

Although Florida's state laws pre-empt local drone regulations, some municipalities still impose additional restrictions related to public safety and privacy. Near UCF, the following local rules apply:

City of Orlando Ordinance

This ordinance restricts drone flights within 500 feet of city owned parks, schools, and large venues like the Camping World Stadium and Lake Eola Park. Testing near popular locations that draw large crowds requires special permits. Since UCF often hosts events that gather large groups, operators must plan carefully to avoid violating this ordinance.
UCF Campus Drone Policy: UCF enforces strict rules for drone flights on campus. Unauthorized drone flights are prohibited to protect the safety and privacy of students and staff. Operators must seek approval from UCF's administration before conducting any tests on campus grounds. This process typically involves submitting a formal request, detailing the purpose of the flight, safety measures, and expected duration.

Large Events

Orlando frequently hosts large events, and special permits are required for drone operations during these gatherings. Operators conducting testing near UCF must consider additional safety precautions to ….?

Special Considerations for SAR Drone Testing

Certain types of drone testing, particularly those related to specialized missions such as search and rescue, require additional permissions and waivers:

Search and Rescue (SAR) Operations

Drones used in SAR missions may qualify for exemptions under FAA's Part 107 regulations. However, operators must still meet basic safety standards, including maintaining visual line of sight or obtaining waivers for BVLOS operations. SAR testing often requires the flexibility of real world conditions, and operators should ensure they are compliant with all necessary legal requirements.

Research and Development Waivers

For advanced research projects involving artificial intelligence (AI) or machine learning, operators may apply for research and development (R&D) waivers. These waivers provide additional flexibility in testing environments, allowing for the development of cutting-edge technologies such as swarm intelligence, autonomous navigation systems, and real time object detection.

Drone testing near UCF, Florida, requires strict adherence to FAA regulations, state laws, and local ordinances. Operators must ensure that they follow the required steps to obtain waivers, permissions, and special use authorizations where necessary. By complying with these legal guidelines, drone operations can be conducted safely and within the bounds of the law. For specialized testing, such as search and rescue missions or AI-based research, the necessary exemptions and permits must be sought well in advance to avoid delays.

Design Plan

The current architecture of VITALS is complex. There are several moving parts with complex algorithms and third party applications that are required for VITALS to succeed. Moving forward, our team has developed a new multi-agent model that will replace the currently lacking LangGraph implementation. We want to leverage AI to provide unparalleled reasoning and context to our mission execution.

Search-and-rescue (SAR) missions demand rapid situational assessment, precise coordination, and safe execution under unpredictable environmental conditions. Traditional operations rely heavily on manual decision-making, which can be slow and prone to information overload. VITALS addresses these limitations by embedding intelligent perception and reasoning capabilities directly at the mission edge. By integrating onboard artificial intelligence, UAV telemetry, and operator oversight, the system allows responders to cover larger areas faster while maintaining situational awareness.

The goal of VITALS is not to replace human operators but to amplify their decision-making ability through real time information synthesis.Its architecture embodies the principles of distributed autonomy, human-centered control, and energy efficiency. Instead of a monolithic AI engine, VITALS uses multiple agents, each with a clearly defined role and interface. This separation of concerns simplifies verification, enables independent upgrades, and allows the system to operate even if one component is degraded. The result is an intelligent assistant that cooperates with the operator to search, identify, and prioritize rescue zones efficiently.

At its core, VITALS operates as a system-of-systems combining onboard computation, communication, and coordinated control. The physical architecture consists of a NVIDIA Jetson AGX Orin ground-station (hosting the multi-agent AI stack, Mission Planner and the GUI Adapter) and one or more UAVs running ArduPilot firmware. Each UAV provides live video, telemetry, and status data. The Jetson AGX Orin processes perception workloads locally,

minimizing latency and ensuring functionality even in disconnected or bandwidth-limited environments.

The high-level mission loop follows three continuous processes: observe, reason, and act. With the new agentic view of VITALS, “Observe” will occur through Agent A’s visual perception of camera feeds; “reason” occurs through the cooperative interpretation of contextual data by Agents B and C; and “act” is executed through Agent D’s mapping and planning components, which send flight missions via MAVLink. The human operator remains embedded in this loop, able to supervise or override at any point through the Mission Planner interface.

New Software Design

The new VITALS architecture is described using the C4 Model, which captures context, containers, and components. At the Context Level, VITALS sits between the operator and the UAV fleet. It ingests sensor and mission data, interprets it through embedded AI, and produces actionable plans. At the Container Level, the system decomposes into Agents A–D, a Vector Database, a MAVLink Router, and a GUI Adapter. Note that this diagram is intentionally large and may violate document requirements, however we have decided it's important to ensure all readers can view this diagram as its the heart of further implementation.

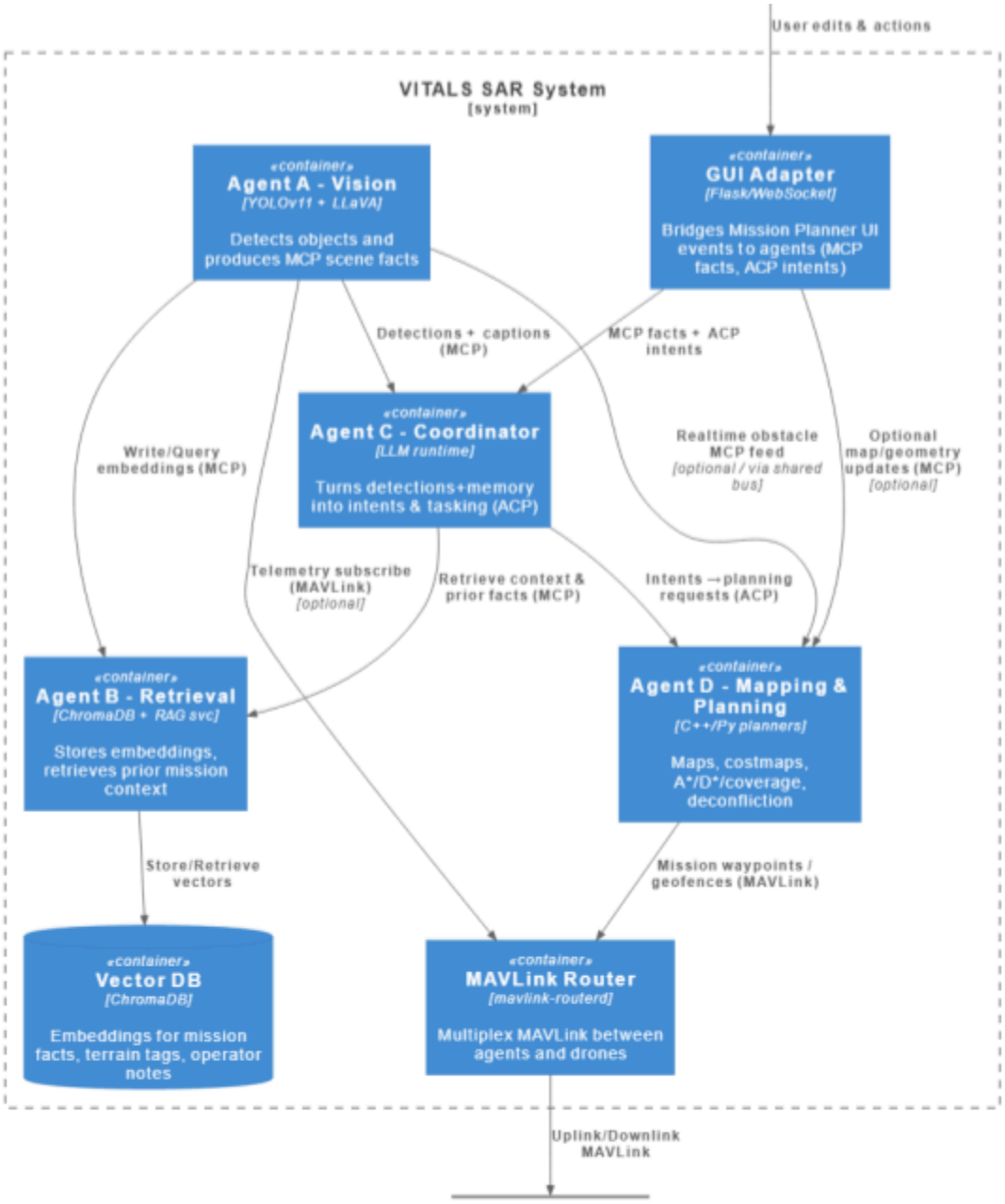


Figure 3.1: New Agentic View of VITALS

Finally, the Component Level reveals each agent's internal modules such as perception pipelines, costmap generation, and reasoning adapters. This hierarchical structure promotes modularity and clear interface definitions.

Each agent communicates through well-defined JSON schemas transported over lightweight message buses. The design avoids centralized state servers; instead, data persistence

and retrieval are handled through a local vector database to maintain speed and autonomy. The choice of message-passing protocols (MCP and ACP) eliminates tight coupling, enabling developers to test or replace agents independently. From an engineering standpoint, the architecture balances computational efficiency, energy management, and fault isolation.Using Jetson hardware allows GPU-accelerated inference while maintaining a small power footprint. By running perception and reasoning locally, the system avoids latency associated with cloud processing, enabling SAR missions where connectivity is uncertain.

Agents and Responsibilities

Moving forward, the VITALS system envisions a distributed multi-agent architecture designed to transform the current proof of concept computer vision capabilities into a fully autonomous search and rescue platform. This proposed architecture decomposes the complex problem of autonomous SAR operations into four specialized, cooperating subsystems, each responsible for a distinct aspect of mission execution: perception, memory, reasoning, and planning. Rather than relying on a monolithic control structure, this multi-agent approach would provide several key advantages: parallel processing of computationally intensive tasks (such as running vision models while simultaneously querying historical data), graceful degradation when individual components fail, and independent scaling and optimization of each agent based on mission requirements. Each agent would communicate through standardized protocols primarily the Model Context Protocol (MCP) for sharing perception facts and the Agent Communication Protocol (ACP) for conveying high level intents creating a loosely coupled system where agents can be developed, tested, and integrated incrementally without requiring a complete system overhaul. The following sections outline the envisioned responsibilities and technical approach for each agent, representing our roadmap for evolving VITALS from a single-function object detector into an intelligent, adaptive SAR system.

The intelligence of VITALS would reside in four cooperative agents, each functioning as a semi-autonomous service within the embedded environment. These agents are designed to operate in a pipeline architecture where data flows from raw sensory input through progressively higher levels of abstraction from perception to memory, memory to reasoning, and reasoning to actionable flight plans. Each agent maintains its own internal state and communicates

asynchronously with others through standardized protocols, allowing individual components to be developed, tested, or replaced independently as technology evolves.

Agent A - Vision and Perception

Agent A serves as the system's sensory front-end and will be extracted from the existing ComputerVision directory. The current objectDetection.py module provides the foundation, but must be refactored to support continuous video stream processing rather than single-image inference. Development will migrate from the existing rf3v1.pt model to YOLOv11 for rapid object detection while maintaining backward compatibility with the current detection pipeline. The integration of LLaVA for visual-language captioning represents a parallel development track that can proceed independently.

Agent A fuses detection results from both systems into structured Model Context Protocol (MCP) facts describing what is seen, where it is located, and the confidence of each observation. The agent's interface will expose MCP-formatted detection facts as JSON messages containing bounding boxes, confidence scores, class labels, GPS coordinates (when available), and LLaVA-generated semantic descriptions. These MCP messages establish a shared "world model" that higher-level agents consume. Because visual inference is computationally intensive, Agent A is GPU-accelerated and can dynamically adjust its frame rate to balance performance and energy use.

Agent B - Retrieval and Knowledge Management

Agent B functions as the system's long-term memory and retrieval engine, requiring net-new development as no dedicated memory system currently exists. This agent will implement ChromaDB as the vector database backend, storing MCP facts from Agent A, operator annotations, mission parameters, historical detection data, prior detections, and contextual metadata as vector embeddings.

When queried by other agents, particularly Agent C, it performs Retrieval-Augmented Generation (RAG) by returning semantically relevant information drawn from this memory through similarity search. Through RAG, Agent C gains immediate access to mission-specific

intelligence such as previously scanned sectors, historical detections, and operator annotations. This mechanism prevents redundant search coverage, supports consistency across missions, and grounds reasoning in real operational context instead of the LLM's pretrained data alone. Initial development will focus on ingestion pipelines that convert MCP messages into vector embeddings and basic retrieval functionality that supports simple semantic queries.

Agent C - Coordinator and Reasoner

Agent C is the cognitive hub of VITALS and builds upon the existing LangGraph implementation found in the current codebase. The LLaMA language model integration through Ollama provides the reasoning foundation, but the current implementation is tightly coupled to the GUI's ChatSidebar. Agent C must be decoupled into a standalone service that accepts MCP perception facts and RAG-retrieved context from Agent B, then synthesizes mission intents using the large-language-model (LLaMA 3, served locally through Ollama). These intents, formatted as Agent Communication Protocol (ACP) messages, describe what tasks to perform and under what constraints.

The reasoning loop will implement a sense-reason-act cycle where Agent C continuously monitors incoming MCP facts, queries Agent B for relevant context, and generates ACP intents for Agent D. Agent C also enforces mission policies, resolves conflicts, and records rationale for operator review. By combining LLM reasoning with RAG-based contextual retrieval, it ensures decisions remain both adaptive and traceable.

Agent D - Mapping and Planning

Agent D converts high-level intents into actionable flight missions and leverages the existing terrain preprocessing and path planning modules in the PathPlanning directory. The PostgreSQL/PostGIS infrastructure and OSM data integration are already functional. Development focuses on wrapping these capabilities behind an ACP interface that accepts high-level mission intents and translates them into MAVLink mission commands.

Agent D maintains real time terrain maps and costmaps, executes A*/D* path planning algorithms for route optimization, and packages result in MAVLink missions. It continuously

monitors telemetry and dynamically updates routes as new MCP detections or ACP priority changes arrive. In essence, Agent D operationalizes the reasoning outputs of Agent C into safe, efficient UAV trajectories.

Agent A Plan

Agent A represents a significant architectural evolution of the existing computer vision infrastructure found in the ComputerVision directory. The current system provides a foundational object detection capability through the objectDetection.py module, which loads the rf3v1.pt model and exposes two primary functions for detecting persons and floating objects in aerial imagery. This existing implementation demonstrates proof-of-concept functionality for the AFO dataset, successfully identifying bounding boxes, class labels, and confidence scores from static images. However, it operates as a standalone utility without integration into a broader intelligent system, lacks real time video stream processing, and provides no semantic understanding beyond object classification.

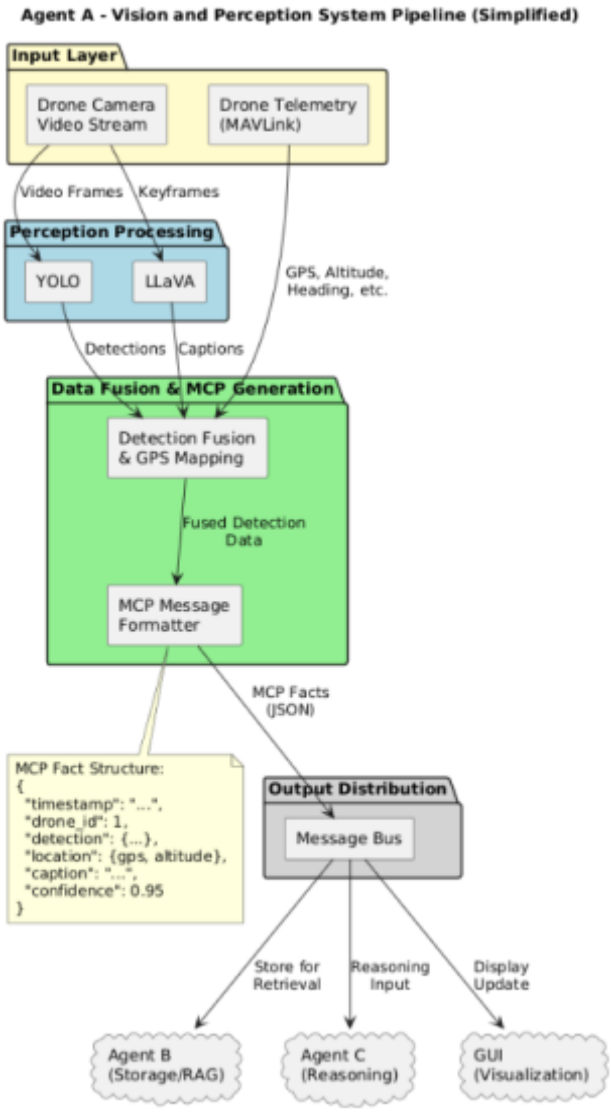


Figure 3.2: Agent A System Pipeline

Agent A will build directly upon this foundation by retaining the core YOLO-based detection pipeline while introducing several critical enhancements. First, the current single-model approach (using only rf3v1.pt) will be upgraded to YOLOv11, which offers improved accuracy and inference speed essential for real time SAR operations. The existing yolov11Custom.pt model already present in the repository may serve as a transitional

stepping stone or provide a comparison baseline during this migration. Second, Agent A will transform the current static image processing workflow into a continuous video stream processor, replacing the hardcoded image_path references with a dynamic frame capture system that ingests live drone footage. Third, and most significantly, Agent A will introduce multimodal perception by integrating LLaVA (Large Language and Vision Assistant) alongside YOLO detections. While YOLO provides fast, structured object localization (what and where), LLaVA will add visual-language understanding generating natural language descriptions of scenes, contextualizing detections (e.g., "person in orange life vest waving near capsized boat"), and identifying environmental conditions that pure object detection cannot capture.

The integration will be achieved by wrapping the existing detection functions within Agent A's service architecture. The current detect_and_draw() function's visualization capabilities will be preserved for operator interfaces, while the raw detection output from detect_objects() will be reformatted into structured Model Context Protocol (MCP) messages. These MCP facts will include not only the bounding box coordinates and confidence scores already extracted by the current system, but also temporal information (frame timestamps), spatial metadata (GPS coordinates from drone telemetry), and LLaVA-generated semantic annotations.

Agent B Plan

Within VITALS, Agent B acts as the central knowledge and memory subsystem. It preserves, organizes, and serves mission-relevant information generated by all other agents. Whenever Agent A detects an object, whenever an operator annotates a map or issues a manual override, or whenever Agent C produces a rationale or intent, these events must be logged, contextualized, and made queryable. Agent B is responsible for all of these tasks, establishing itself as the factual backbone of the system.

Agent B also enables retrieval-augmented generation (RAG), the method by which Agent C’s reasoning is grounded in the actual state of the mission. Rather than relying only on the static knowledge internalized by the LLM, Agent C requests contextual information from Agent B. Agent B responds with the most relevant excerpts from the mission memory, selected through

vector similarity search and, when needed, re-ranked using a more computationally intensive scoring model. This retrieved content is inserted into the LLM prompt, enabling contextual reasoning. Thus, Agent B not only stores past events but also actively shapes the reasoning cycle by influencing how Agent C interprets them.

Finally, Agent B serves as an audit log for the entire system. In search-and-rescue operations, accountability and transparency are critical, particularly for after-action review or compliance with regulatory requirements. Agent B maintains an append-only record of mission activity that captures decisions, observations, and system health metrics, ensuring that mission behavior can be reconstructed and analyzed after the fact.

Conceptual Architecture

Agent B can be conceptualized as a pipeline that transforms raw mission events into an evolving knowledge base. The subsystem includes an ingestion layer that accepts incoming mission events, a durable log for storing them, an embedding subsystem for semantically meaningful event transformation, and a vector database for storing those embeddings. On top of these, it exposes a retrieval engine that performs similarity-based searches and returns relevant context bundles for reasoning and user-facing applications. A summarization mechanism monitors the mission data for redundancy or high-frequency updates and generates compact summaries that preserve meaning while reducing storage load. Finally, a long-term archive exports mission data into Parquet files for analysis, training, or compliance.

These components operate asynchronously but form a logically cohesive system. Events from Agents A, C, and D flow into Agent B continuously. Each event is validated, timestamped, and written to a relational database. Selected subsets of these events, typically those containing semantically meaningful text, are transformed into natural-language sentences and encoded using a dense embedding model. These vectors are stored in ChromaDB, allowing queries to retrieve similar events based on semantics rather than simple textual matches. The summarization component monitors the data stored in the log and vector store, periodically condensing related events into more general statements. Over time, older events are exported to Parquet and removed from the hot storage layer, keeping the system performant.

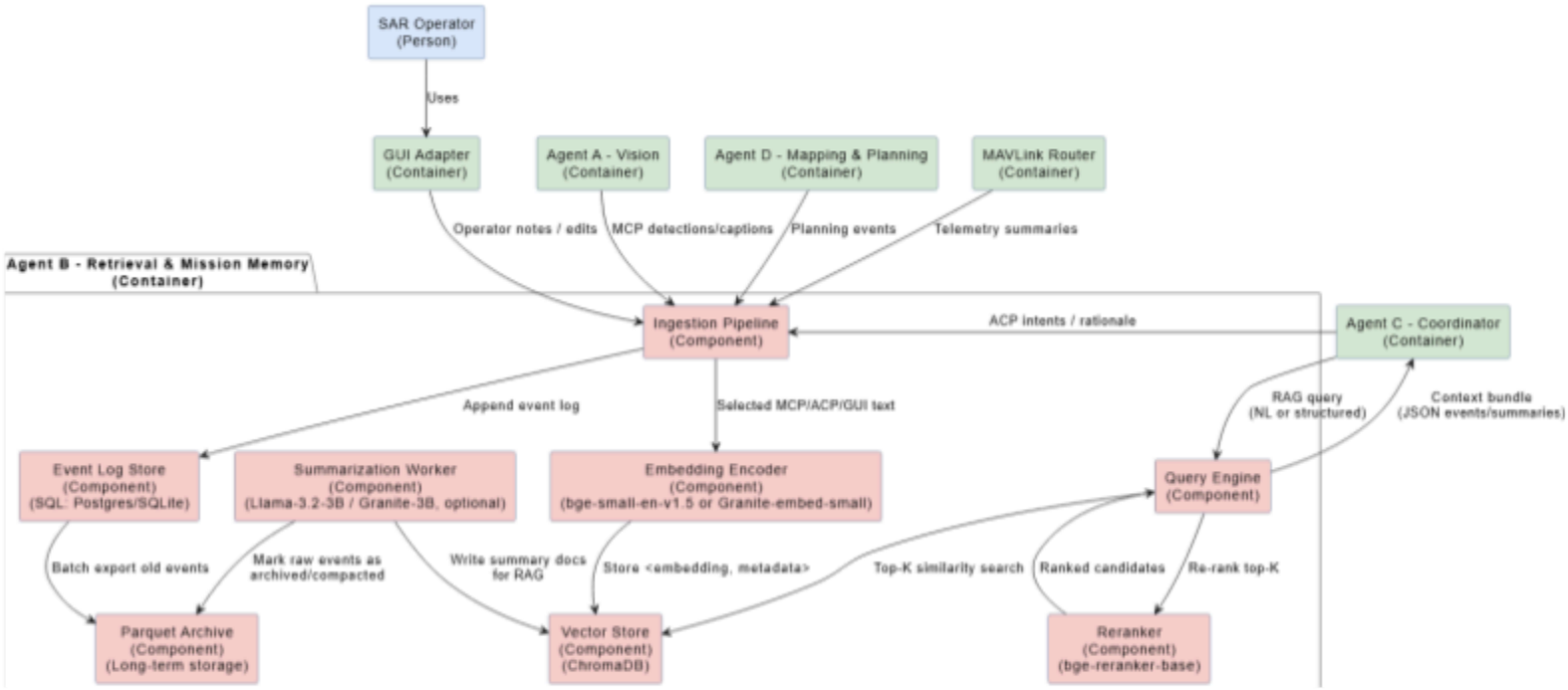


Figure 3.3: Agent B System Pipeline

Event and Data Model

Agent B relies on a unified event envelope, a structured JSON format that encapsulates all system events regardless of their source. This approach promotes consistency and simplifies validation, storage, and retrieval. Each event contains a unique identifier, mission ID, timestamp, source agent name, event type, a flexible payload section, and a version tag that allows for schema evolution. Correlation and span identifiers are used to trace relationships between events, enabling developers and analysts to map complex workflows and interactions.

Ingestion Pipeline API and Behavior

The ingestion pipeline forms the entry point into Agent B. Other agents interact with it primarily through a HTTP endpoint, typically a POST/ingest request containing a single event envelope. The ingestion process begins by validating the correctness of the event. Required fields are verified, timestamp formats are checked, and optional fields are filled with default or inferred values when appropriate. Every event receives an ingest_ts timestamp that records when Agent B received it, which is useful for latency measurement and system diagnostics.

Next, the event is normalized to ensure consistency of casing, tag formatting, and identifier structure. Once normalized, the event is inserted into the SQL database using idempotent insert semantics. If the primary key already exists, due to retrying an event for example, the insert is simply skipped, maintaining consistency.

After persistence, the ingestion pipeline determines whether the event should be considered for semantic embedding. Text-rich events, such as operator notes, MCP detections, and ACP rationales, are passed to the embedding encoder. The pipeline also incorporates backpressure mechanisms to prevent overwhelming downstream components. If embedding operations slow down or the SQL database becomes congested, a write-ahead log ensures that data remains safe until the pipeline can catch up.

Embedding Encoder Architecture

At the core of Agent B's retrieval capability is the embedding encoder. The encoder transforms selected event data into dense vector representations. These representations capture high-level semantics and allow similarity comparisons in a continuous vector space. The default model used in the system is bge-small-en-v1.5, a compact model that provides balanced speed and accuracy suitable for embedded systems. This model is capable of mapping natural-language sentences into a 384-dimensional vector space.

To generate embeddings, the ingestion pipeline first constructs a natural-language summary of the event. For example, a detection event might be represented as "A person was detected in sector B4 with 91% confidence by UAV_2 at 18:13Z." This ensures that the embedding captures semantic meaning rather than raw structured data. Metadata such as event ID and timestamp accompany the vector, enabling downstream filters and query refinement.

ChromaDB Vector Store Design

ChromaDB serves as the vector database for Agent B. It provides efficient storage and retrieval of embeddings along with associated metadata. Embeddings are organized into collections that may be filtered by mission ID, event type, time ranges, or spatial identifiers.

When Agent C makes a retrieval request, the query text is embedded using the same model as the stored vectors, and cosine similarity is used to rank the relevance of stored events.

The vector store supports metadata filtering, enabling queries constrained to the current mission or a specific time frame. This prevents irrelevant historical data from interfering with reasoning. The store can also trim older entries or store only summary-level embeddings for prior missions. By maintaining the vector store in memory when possible, ChromaDB allows low-latency retrieval essential for real time decision-making.

Conclusion

Agent B is the core memory and retrieval subsystem in the VITALS search-and-rescue architecture. It ensures that the system can reason over time, understand mission context, and provide transparent and accountable decision-making. Its design balances robustness, efficiency, and flexibility while remaining practical for embedded systems. By combining structured event logging, dense embeddings, vector search, reranking, and summarization, Agent B establishes itself as an essential component for reliable and intelligent mission execution. This report provides a detailed, implementation-ready analysis suitable for academic evaluation and real-world deployment.

Agent C Plan

Agent C functions as the cognitive core of VITALS. This agent will be responsible for the mission reasoning, interpreting operator intent, and interfacing memory from Agent B to inform mission decisions. It serves to bridge the gap between VITALS and operator intention by translation natural language commands into structured mission intents and maintaining awareness through Agent A and B.

The current implementation of this coordination logic in VITALS is a proof of concept reasoning architecture using the LangGraph module and Ollama for mission reasoning. It suffers from architectural limitations like being tightly coupled to the GUI ChatSideBar component making it impossible to deploy or test it independently as a stand alone container. The system also does not exhibit any memory or storage that Agent B offers with ChromaDB.

Function

The new Agent C will use a retrieval-augmented large language model to avoid hallucinations and to remain grounded in real mission data. By integrating MCP detections from Agent A, contextual information retrieved from Agent B, operator adjustments from the GUI, and planning outputs from Agent D, Agent C produces a coherent stream of mission logic. This report presents a complete engineering analysis of Agent C with a focus on architectural decisions that support explainability, robustness, and operational integrity.

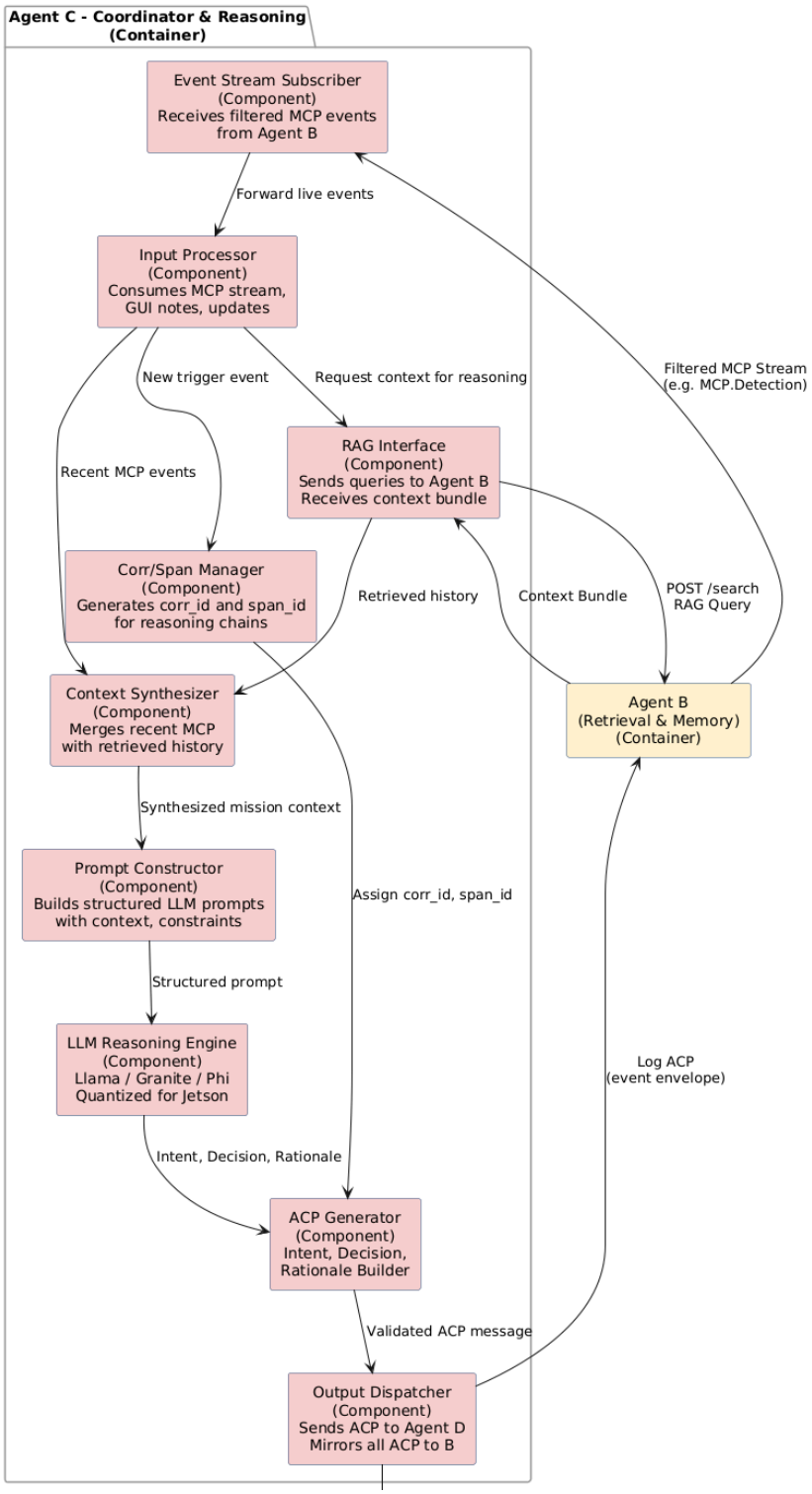


Figure 3.4: Agent C System Pipeline

Input

Agent C can be understood as a layered reasoning engine, incoming perceptual levels and operator inputs flow into the input processing layer that determines whether reasoning should be triggered. When reasoning begins then, Agent C will retrieve historical context from Agent B through the RAG interface. Agent C then constructs a structured prompt that contains perceptual information, relevant history, operator intent, and mission context.

Output

The LLM processes this prompt and produces a structured action and a supporting rationale. The ACP message that results from this process is validated for safety and structure. The output is then dispatched to Agent D for execution, and also logged in Agent B for traceability and future retrieval. This entire sequence forms the core of Agent C's conceptual architecture.

Performance

Agent C requires significant computational resources due to the cost of LLM inference. To meet the latency requirements of search and rescue missions, the model is quantized and runs on hardware that supports GPU acceleration. Retrieval reduces prompt size and improves relevance, increasing inference efficiency. Additional performance improvements are achieved through caching, batching, and rate limiting of reasoning cycles to prevent excessive LLM calls.

Recap

Agent C is the reasoning and coordination subsystem of the VITALS architecture. It synthesizes perceptual input, mission history, operator intent, and system constraints to produce structured decisions. These decisions guide the UAV through complex and dynamic search and rescue environments. Through its integration with Agent B's mission memory, its subscription to perceptual event streams, and its alignment with structured planning through Agent D, Agent C provides a reliable and explainable mechanism for autonomous mission control. The design of Agent C balances performance, reliability, transparency, and mission-aware reasoning, making it a critical component of the VITALS ecosystem.

Agent D Plan

Agent D functions as the bottom of the agent stack where the high level intents and world knowledge get turned into geometry and control commands. Currently there is a fundamental construction put in place for Agent D however it is extremely fleshed out and only generates waypoints considering the beginning inputs provided. The new design for the Mapping and Planning is going to be tuned to be able to handle continuous inputs in MCP and ACP form enabling the agent to have the ability to create safe trajectories for the drones.

Function

Currently, our proposed plan for the Agent D architecture is to split up the Agent to have 2 main functions:

1) **Mapping:** During this phase the agent will receive and decode the MCP and ACP data, along with any telemetry data provided, and use this information to begin constructing an updated map.
2) **Planning:** This phase is provided with the map from the mapping phase utilizing local and global path planning algorithms to update the waypoints generated.

Currently put in place is a waypoint generation algorithm that uses a greedy approach to generate a global path for each drone. To alter the current approach, we plan on changing the greedy approach to a more inference based approach where we guide the agent to implement cost map updates to the currently created map. The cost map will provide weighted factors, such as priority and risk values, that will be held within the tile class and updated within the Mapping procedure.

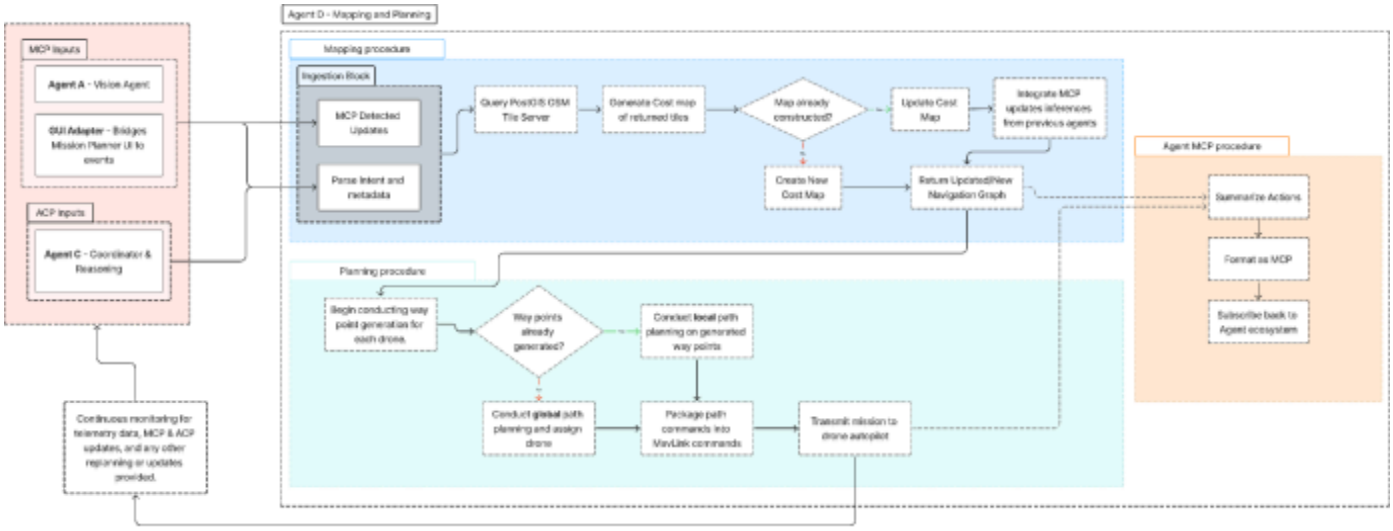


Fig 3.5: Diagram for Agent D

Input

Agent D ingests information from four primary streams, as shown in the diagram:

1) **Agent C:** This agent provides a high-level ACP intent specifying mission objectives, targets, constraints, and priority.

2) **Agent A:** Agent D is a subscriber to the MCP perceptual facts and world-states updates. Majority of the information will be originating from Agent A which is the vision agent providing the agents eyes into the spatial information that the drone is present in.
3) **Telemetry:** This is raw data provided from the MissionState that allows the Agent to keep track of where the drone is and other values at all times.
4) **Open Street Map Tiles:** This source is acquired after the prior sources are broken down as we only need to query the OSM tile server if we are generating a new map or needing to update the boundaries of the prior map. The OSM tile server is the starting point of our map and should not be queried unless we are updating our map and need more formal data for map generation.

Output

The output of Agent D is inherently operational in the sense that some drones do not need replanning or updates to their current way point. The missions that are generated are dispatched directly to the appropriate drone and simultaneously tracked within the internal state of the VITALS software suite. In addition Agent also subscribes executive summaries back to the MCP environment for the other agents to evaluate what the agent has done and assigned to the drones. This is an extremely important function for the Agent architecture to work due to both Agent C and Agent B requiring the mission history for future reasoning within active missions.

Performance

The performance of Agent D is heavily reliant on the aspects such as path computation, terrain lookup, map reconstruction/generation, and not so much by the inference cost. To help improve performance, we are going to generate the costmap through caches across successive planning cycles to avoid the need of regenerating the entire cost map. Furthermore, the path solver chose will be dynamic as well as D* tends to be more cost heavy computationally we will dynamically choose the planner based upon criteria such as telemetry information.

Recap

Agent D serves as the gateway within the VITALS Software system between the MAVLINK commands and the entire agent architecture. Agent D's main task will be to

continuously replan in response to real-time perceptual and telemetry data through systematic integration within the agent architecture proposed within this document.

Graphical User Interface Refactoring

The VITALS drone control system represents a search and rescue platform that enables autonomous drone swarm operations through natural language commands. At the heart of this system lies a graphical user interface that serves as the primary interaction point between operators and the drone fleet. This interface, originally implemented as a single Python file containing 1,262 lines of code, presented significant challenges for maintenance, testing, and collaborative development. These were the primary reasons why we refactored the GUI, enabling us to get to a modular, component based architecture that maintains all original functionality while dramatically improving code organization and maintainability.

The original GUI implementation consolidated all interface logic, widget definitions, entity representations, and page management into a single file. This approach, while initially expedient for rapid prototyping, created numerous complications as the project evolved. The file contained twelve distinct classes ranging from simple data structures to complex interface pages, all tightly coupled through direct references and implicit dependencies. The Drone class, responsible for visual representation of aircraft on the mission map, was intertwined with the Job class that managed waypoint assignments, which in turn was coupled to the POI class handling points of interest. Widget classes for displaying drone information, job queues, and chat interfaces were scattered throughout the file, making it difficult to locate specific functionality or understand the relationships between components.

The refactoring process began with a careful analysis of the existing code to identify natural boundaries and logical groupings. Three primary categories emerged from this analysis. Entity classes represented core visual and data elements within the system, including drones, points of interest, and job assignments. Widget classes provided reusable interface components for displaying information and enabling user interaction. Page classes defined the major screens of the application, managing layout and coordinating between different interface elements. This categorization formed the foundation for the new modular architecture.

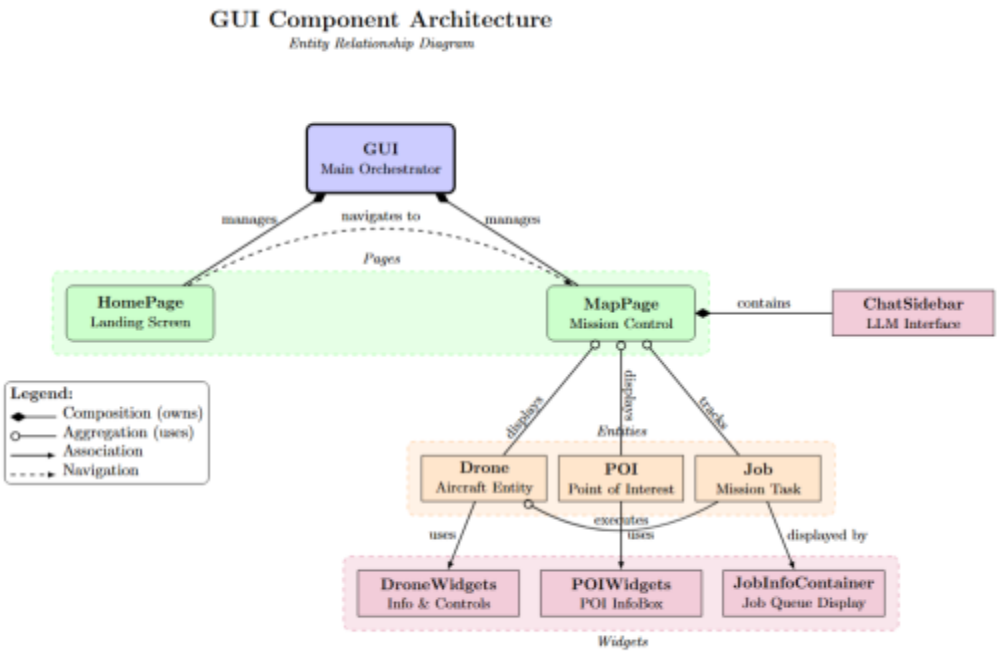


Figure 3.6: GUI Refactored Diagram

The transformation from monolithic to modular required careful attention to dependency management and communication patterns. The original code relied heavily on a gui_ref pattern, where components maintained references to the main GUI object to facilitate callbacks and intercomponent communication. This pattern was preserved during refactoring to maintain backward compatibility with the existing missionState integration, but the references were formalized and made explicit through constructor parameters. Import statements that previously existed implicitly within the single file now required precise orchestration to prevent circular dependencies. Strategic placement of imports within method bodies rather than at module level resolved these circular reference issues while maintaining the necessary connections between components.

The new architecture organizes code into four primary directories within a GUI package. The Entities directory contains classes representing visual elements on the map and their associated data structures. The Drone class manages the visual representation of aircraft, including position markers, heading indicators, and path visualization. The POI class handles points of interest, managing both map markers and popup displays for detailed information. The Job class defines job assignments and waypoint structures used for mission planning. Each entity

maintains its own state and rendering logic while exposing clear interfaces for external interaction.

The Widgets directory houses reusable interface components that can be composed into larger interface structures. The DroneWidgets module contains three related classes: DroneInfoBox displays real time telemetry and status information for individual drones, jobInfoContainer manages the scrollable list of jobs assigned to each drone, and jobInfoItem represents individual job entries within the queue. The POIWidgets module provides the POIInfoBox class for displaying point of interest details in the sidebar. The ChatSidebar class implements the conversational interface for natural language interaction with the language model that translates operator commands into drone instructions. These widgets encapsulate their own rendering logic and event handling while communicating with other components through well defined callbacks.

The Pages directory contains the two main screens of the application. The HomePage class implements the initial landing screen where operators select mission types and configure basic parameters. The MapPage class provides the primary mission control interface, integrating the map display with various control widgets and managing the complex interactions required for mission planning and execution. The MapPage underwent significant refactoring beyond simple extraction, with improvements to polygon drawing logic, GCS placement mechanisms, and event handling that resolved several long standing bugs in the original implementation.

At the root of the GUI package, the refactored GUI.py file serves as the central orchestrator, reduced from 1,262 lines to approximately 434 lines of focused logic. This class manages page navigation, maintains communication with the missionState system, and coordinates events between different components. Rather than containing all functionality, it now delegates to appropriate modules while maintaining the high level application flow. The initialization process creates instances of both pages, manages the transition between them, and establishes the necessary connections to external systems.

The refactoring process revealed and resolved several architectural issues present in the original code. Duplicate class definitions were discovered, with both missionState.py and the original GUI.py containing incompatible Job class implementations that used different attribute names for priority values. This duplication caused runtime errors when the dispatcher thread attempted to access attributes that existed in one implementation but not the other. The refactoring eliminated these duplicates, establishing single sources of truth for each component. Import path problems were resolved by organizing code into proper package structures with explicit import statements. The original code's reliance on implicit availability of classes within the same file was replaced with clear import dependencies that make relationships between components obvious and manageable.

The modular structure provides numerous benefits for ongoing development and maintenance. Individual components can now be tested in isolation, with mock objects replacing dependencies during unit testing. This capability was essentially impossible with the large single file structure, where extracting a single class for testing would require bringing along hundreds of lines of unrelated code. Team collaboration is significantly improved, as we can work on different modules without creating merge conflicts in a single large file. The clear separation of concerns makes it easier for new team members to understand the system architecture and locate specific functionality.

Performance considerations were carefully addressed during the refactoring process. The modular structure does introduce a small overhead from additional import statements and module loading, but this impact is negligible compared to the computational demands of map rendering and drone communication. Memory usage remains essentially unchanged, as the same objects are created regardless of their file organization. The lazy import pattern used to resolve circular dependencies actually provides a minor performance benefit by deferring module loading until necessary.

The preservation of external interfaces was a critical requirement throughout the refactoring process. The missionState system, which serves as the bridge between the GUI and the MAVLink drone communication protocol, expects specific method signatures and callback

patterns. These interfaces were meticulously preserved, ensuring that the refactored GUI remains fully compatible with the existing system. The gui_ref pattern, while not ideal from a pure architectural perspective, was maintained to avoid requiring changes to the broader system. Future iterations might replace this pattern with a more formal event bus or message passing system, but such changes would require coordination with modifications to the missionState integration.

Error handling and edge cases received particular attention during the refactoring. The original monolithic structure made it difficult to trace error propagation through the system, as exceptions could originate from any of the twelve classes contained within the single file. The modular structure provides clearer error boundaries, with each module responsible for handling its own exceptional conditions and propagating only necessary error information to calling code. The polygon drawing logic, which had exhibited several bugs related to state management and cleanup, was completely rewritten with proper state tracking and comprehensive cleanup procedures that prevent ghost polygons and marker artifacts.

The documentation generated alongside the refactored code provides multiple levels of guidance for developers working with the system. Inline comments explain specific implementation decisions and highlight areas requiring special attention. Module level docstrings describe the purpose and responsibilities of each component. The migration guide provides step by step instructions for transitioning from the monolithic to modular structure. These documentation artifacts ensure that the knowledge gained during the refactoring process is preserved and accessible to future developers.

The refactoring project successfully transformed a challenging codebase into a well organized modular architecture without sacrificing functionality or compatibility. The new structure provides clear separation of concerns, explicit dependencies, and improved maintainability while preserving all original features and external interfaces. The reduction of the main GUI.py file from 1,262 lines to approximately 500 lines, redistributed across fourteen focused modules, demonstrates the effectiveness of the modular approach. Each component now

has a clear, single responsibility, making the codebase more understandable, testable, and maintainable.

This transformation represents more than a simple reorganization of code; it establishes a foundation for future enhancements and continued development of the VITALS system. The modular architecture enables parallel development efforts, facilitates testing and debugging, and provides clear extension points for new functionality. As the VITALS project continues to evolve with additional drone capabilities, enhanced search algorithms, and improved operator interfaces, the refactored GUI architecture provides the flexibility and maintainability necessary to support these advancements. The success of this refactoring effort demonstrates the value of investing in architectural improvements even for systems already in active use, showing that careful planning and systematic execution can achieve significant structural improvements while maintaining operational continuity.

Over the next few months, the VITALS system will undergo a comprehensive optimization initiative focused on three critical areas that directly impact operational efficiency. These enhancements represent the progression of the system from its current functional prototype state toward a production ready platform capable of supporting real world deployments with physical drones. The optimization effort will proceed through overlapping development phases, with each improvement building upon the foundation established by previous work

Containerizing VITALS

Creating a containerized Docker architecture for deployment, especially for a multi-agent software would ensure compatibility between development environments, as well as bringing modularity, parallel development and testing. These features would benefit greatly to providing the reliability needed to create an agentic drone swarm capable of search and rescue operations.

Container Architecture

In order to modulate the software, the current state of the software would have to be split into an architecture that accounts for the four primary AI agents as well as auxiliary services, that are functionally autonomous, and that is capable of providing lightweight and fast message

passing between containers. By splitting agents and services, each container is responsible for one module so as to follow the single responsibility principle, and by having agents operate on their own input and output, this maintains the interface segregation principle. Following SOLID design principles is crucial on the low-level implementation of this software, as a good foundation can weather tough storms. Since each container would hold aspects of a whole codebase, dependency management would be taken care of with the help of Docker compose. Instead of loading the same dependencies in all the containers, each container would only load dependencies that would be needed for their individual function.

The proposed architecture with these principles in mind would implement a shared virtual network to address the fast and lightweight communication, as well as seven containers, that being:

1. Agent A (Vision) - responsible for image inference, perception and visual-language analysis
2. Agent B (Memory) - responsible for managing mission data and semantic retrieval through a vector database
3. Agent C (Reasoner) - responsible for mission reasoning using a local Ollama runtime
4. Agent D (Planner) - responsible for generating autonomous flight paths and sending the plan to MAVLink router

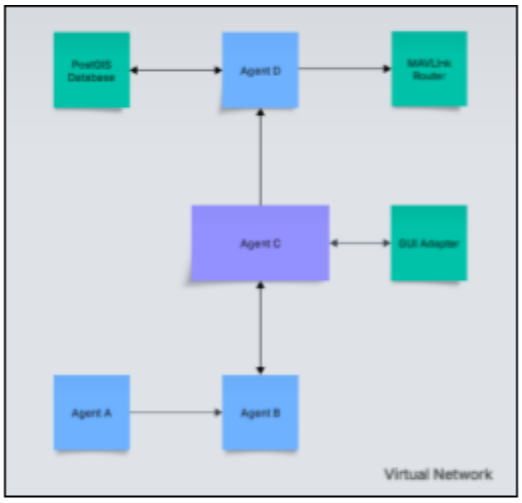


Figure 3.7: Container Architecture Diagram

5. PostGIS Database - responsible for storing geospatial and terrain data for mission planning
6. MAVLink Router - responsible for communication between VITALS running on the Jetson and Mission Planner
7. GUI Adapter - responsible for providing real time GUI data between Agent C and the operator

Now that the agents and services are split into containers, the shared virtual network acting as a bridge between each other is now responsible for allowing asynchronous communication. The overall architecture would then look something like the figure shown.

Docker Networking Design

As mentioned before, to create an architecture that allows for agents to freely communicate, a lightweight and reliable network has to be established. Luckily, Docker comes equipped with networking capabilities. Using this network provides an isolated virtual communication environment that mirrors a distributed microservices architecture, while also maintaining a secure connection between the agents that could potentially be strengthened to fend from outside threats.

While Docker has its own default network called a bridge, we would be using a user-defined Docker bridge network instead. Being defined in the Docker compose file, many fields can be predetermined as Docker is able to manage internal hostnames, IPv4 addresses, and DNS entries, which in the default Docker bridge they would be randomly assigned. Since the containers for the software would reside on the same network, communication between them would be seamless, using HTTP or TCP endpoint without exposing ports to the host system. This feature synergizes well keeping in mind that the ground control station and drone swarm would be communicating using radio frequencies over TCP. The separation between internal and external networking also reduces configuration complexity and ensures deterministic interagent messaging.

Further investigating the communication between containers, certain protocols such as Model Context Protocol, and the Agent Communication Protocol would also be transported around the VITALS network using TCP. For instance, Agent A would generate a JSON packet that follows the MCP schema to then be sent to Agent B via TCP. Most of the container communication would be through JSON packet exchange.

Overall the essential benefits of the user-defined Docker bridge network includes isolation from external systems, easier configuration by resolving to hostnames, reduced attack

surface as the MAVLink port is the only exposed port, and reproducibility on other identical machines.

Protocol Implementation and Architecture

With agents isolated, the second phase implements the communication protocols that enable coordination. Both Model Context Protocol (MCP) and Agent Communication Protocol (ACP) must be formalized as JSON schemas with versioning, validation, and backward compatibility considerations.

MCP

MCP Schema Design will define the structure of perception facts emitted by Agent A. Each MCP message includes a unique fact ID, timestamp, source identifier (drone ID, camera ID), detection type (bounding box, semantic description), confidence score, and spatial metadata (GPS coordinates, altitude, heading). The schema must accommodate both YOLO detections (structured bounding box data) and LLaVA outputs (unstructured natural language descriptions). Version fields ensure that schema evolution does not break existing agents.

ACP

ACP Schema Design defines mission intents flowing from Agent C to Agent D. ACP messages describe goals (investigate POI, search grid sector, return to launch), constraints (altitude limits, no-fly zones, battery reserves), and priority levels. Unlike MCP's perception-focused structure, ACP emphasizes declarative specifications of what should be accomplished rather than how to accomplish it. This separation allows Agent D's planning algorithms to evolve independently of Agent C's reasoning logic.

RAG Integration and Contextual Reasoning

The third phase transforms Agent C from a simple LLM wrapper into a context-aware reasoning engine by integrating Retrieval-Augmented Generation. This capability prevents the reasoning agent from operating in a vacuum by grounding its decisions in mission history, operator directives, and environmental observations.

ChromaDB Deployment on the Jetson requires careful resource management. The vector database will store embeddings generated from MCP facts, with each fact embedded using a lightweight sentence transformer model (e.g., bge-small-en-v1.5 with 384-dimensional embeddings). Collections will be organized by mission ID to enable efficient historical queries across deployments. Persistence will use local disk storage with periodic backups to prevent data loss during power failures.

Embedding Pipeline development connects Agent A's outputs to Agent B's storage. As MCP facts arrive, Agent B extracts textual descriptions (YOLO class labels, LLaVA captions, GPS coordinate strings) and generates embeddings. Metadata fields (timestamp, confidence, source drone) are stored alongside embeddings to enable filtered retrieval. The pipeline must handle high-frequency fact arrival (potentially 10+ detections per second during active search) through batched embedding and asynchronous ingestion.

Query Interface for Agent C provides semantic retrieval capabilities. When generating mission intents, Agent C formulates natural language queries (e.g., "previous detections in northeast sector," "areas already searched by drone 2") and receives ranked results from Agent B. These results are injected into Agent C's LLM context window as additional system messages, providing grounding that prevents hallucination and ensures consistency with prior observations. Retrieval scope can be limited by time windows, spatial regions, or confidence thresholds to keep context relevant.

Prompt Engineering for Agent C must structure the LLM interaction to effectively use RAG context. The prompt template will include sections for current mission status, recent MCP detections, RAG-retrieved historical context, operator directives, and the specific reasoning task (generate search plan, prioritize POIs, allocate drones). Few-shot examples will demonstrate proper ACP intent formatting. Chain-of-thought prompting will encourage Agent C to explicitly reason about trade-offs before generating intents, improving transparency for operator review.

Ground Control Station Consideration

The VITALS software was initially planned to run with Mission Planner as its GCS, capable of sending MAVLink packets, displaying drone telemetry, and access to drone firmware. The decision to use MP is based on the assumption that the Jetson would be running the software on a Windows machine, as it requires an X11 server and .NET framework. Upon further research and design considerations, our group has decided to work with Linux rather than Windows in order to make the software as lightweight and as robust as possible. By doing this though, several more software would be required to emulate the .NET framework and X11 server on a Linux machine, which would cause negligible change if we were to make the switch.

As a workaround to support the change to Linux, we discovered QGroundControl, a software that runs natively on Linux, and provides much of the same services as Mission Control. In the interest of aligning with the mission statement of VITALS, to create a lightweight and fast search and rescue drone swarm, also as we are still fresh into this project and from understanding that there are little to no direct ties to Mission Planner from the repository, this change would seem to be a consideration worth spending extra time researching more into.

Expected Outcomes and Future Considerations

Upon completion of Senior Design, the VITALS system will have transformed from a functional but sometimes cumbersome prototype into a streamlined mission planning platform. The automation of Mission Planner initialization will eliminate one of the most significant barriers to rapid deployment, allowing search and rescue teams to begin operations within seconds rather than minutes. The enhanced polygon tool will provide the flexibility and precision users need to define complex search boundaries that accurately reflect real world terrain constraints. The optimized map performance will create a fluid, responsive experience that maintains user focus on mission planning rather than fighting with sluggish interfaces. These optimizations also establish architectural patterns and infrastructure that will support future enhancements. The automation framework can be extended to support additional ground control station software beyond Mission Planner. The interactive polygon system provides a foundation for more advanced boundary definition features such as exclusion zones or priority regions. The

caching and rendering optimizations create headroom for displaying additional overlay information such as weather data or terrain analysis results without compromising performance.

Hardware Design Plan

The hardware design for the VITALS project is subdivided into two main categories: drone hardware and the ground control station. All supplied hardware was provided by our Lockheed Martin sponsor, Joe Rivera, which guides much of our implementation strategy. Originally, the project was allocated a $3,000 budget; however, after consultation with Joe, this was revised to $300 due to the project's strong software emphasis, requiring us to work resourcefully with inherited and existing hardware. Below, we detail our approach for integrating and expanding these hardware systems.

Drone Hardware

Our primary UAV platform is a quadcopter equipped with the Pixhawk Flight Controller 2.4.8, a widely-used open-source autopilot system known for its robust capabilities and community support. The drone itself was inherited from a prior UCF Senior Design team, with flight readiness and overall build integrity thoroughly tested and confirmed by Brendan Rodriguez through manual flight trials. While manual flight validation went smoothly, the transition to autonomous flight introduced several technical challenges.

The central blocker was the initial batch of radio modems, which proved incompatible due to mismatched firmware and versioning, halting any serious progress on autonomous flight. Recognizing the critical nature of these issues, our team collaborated closely with Joe Rivera, who subsequently provided us RFD 900x-US radio modems, which are industry standard devices with a solid track record of reliable telemetry in field operations. With this upgrade, our team was finally able to begin experimentally sending and receiving MAVLink packets between the Pixhawk and ground control station as of November, setting the stage for autonomous mission validation.

Moving forward, the RFD 900x-US modems will be the backbone of our wireless communication infrastructure, enabling robust long-range telemetry and bidirectional command

uplink. Each radio is interfaced with both the drone's Pixhawk and the ground station computer, making possible the transmission of telemetry and command data over significant field distances, used in real-world SAR scenarios. Additional sensors or interfaces added to the drone will be designed to integrate seamlessly with both the Pixhawk's I/O and the wireless backbone established by the modems.

Ground Control Station Hardware

The Ground Control Station (GCS) serves as the mission planning, data visualization, and operational control hub for VITALS. Our implementation is anchored around the use of a laptop-class field computer, configured to interface directly with the RFD 900x-US modem, and running the Mission Planner software suite. This setup provides a familiar, flexible platform for team members and operators, supporting both command scripting and live telemetry monitoring. At a hardware level, the ground station features:

- RFD 900x-US Radio Modem: Paired with the drone's onboard modem, this enables real time MAVLink telemetry transfer, mission uploads, and live health/status data exchange.
- Edge AI/Video Processing Platform: For advanced builds and final demo, we are considering integrating a small-form-factor single-board computer (NVIDIA Jetson Orin Nano) for real time AI inference on incoming video streams. This will allow us to prototype mission-critical computer vision and alerting functions within the field GCS, independent of cloud connectivity.
- Operator Interface: We employ monitors/displays, field keyboards, and external batteries or field power banks to ensure uninterrupted operation through search and rescue scenarios. Power management is a key design consideration due to the unpredictable nature of field deployments.

An important hardware consideration throughout the design is extensibility, ensuring that additional sensors, operator input devices, and network interfaces can be integrated as the project's needs evolve. All critical hardware connections are documented and modular wherever possible, to support future expansion or troubleshooting, and to facilitate handoff to subsequent project teams.

Hardware Acquisition and Budgeting

Given the drop in available funds, our approach prioritizes maximizing the capabilities of inherited systems and selectively supplementing them only where necessary to achieve reliable baseline operations and project MVP (minimum viable product) criteria. Future hardware requests are reviewed based on clear gaps, such as the need for spare power modules, telemetry range extenders, or cooling for edge-AI platforms, and are justified against core project requirements.

Regulatory Requirement Considerations

To ensure full compliance with federal, state, and local regulations governing unmanned aerial systems, our flight operations for the VITALS project will be conducted exclusively on private land with the explicit permission of the property owner. All testing and demonstrations will take place at a designated site in Wekiva Springs, Florida, which has been reviewed to meet relevant airspace, safety, and municipal requirements. This approach allows us to adhere to FAA guidelines, Florida state statutes, and local ordinances concerning UAV use, privacy, and public safety during every phase of our project.

Test Plan

Hardware Testing

To validate the VITALS project's hardware system integration and operational reliability, our team defined a hardware test plan focused on real-world mission scenarios and incremental validation of drone and ground station connectivity. The test plan is structured into sequenced stages:

Test Objectives

- Confirm successful autonomous flight using the Pixhawk Flight Controller 2.4.8 as the autopilot, with inherited hardware and updated radio modems.
- Establish robust wireless MAVLink communication between the drone and ground station over RFD 900x-US radio modems.
- Validate seamless command and telemetry exchange in both Windows and Ubuntu Linux field environments.

Test Procedures

1. Manual Flight Baseline
    - Manually fly the quadcopter inherited from the prior UCF Senior Design team to confirm flight readiness and hardware integrity.
    - Document baseline performance and physical safety checks before advancing to autonomous modes.
2. Autopilot/Autonomous Flight Validation
    - Configure Pixhawk for autopilot mode.
    - Upload pre-planned waypoints and monitor autonomous flight behavior.
    - Confirm correct execution of mission plan, stability, and emergency protocols (e.g., "Return-to-Home," battery failover provided by dual packs).
    - Perform tests indoors without propellers before outdoor validation for system safety.
3. Radio Modem MAVLink Testing
    - Connect RFD 900x-US radios to both drone (Pixhawk) and ground station laptop.
    - Transfer MAVLink packets that validate bidirectional uplink and downlink for live telemetry and command/control data.

- Carefully check cable and connector fit; use FTDI and JST 6-pin connectors for robust setups, avoiding insecure or makeshift wiring.
- Troubleshoot and document any pairing or firmware issues, using matching radio modem models and versions.
- Repeat tests after unplugging/reattaching cables and in different environments to confirm stable performance under real deployment conditions.

4. Environmental Interface Testing
    - Run Mission Planner and custom VITALS interface on Windows, then migrate scripts and ground station software to Ubuntu Linux for field deployment.
    - Test video input and edge-AI inference (if available) from the NVIDIA Jetson onboard, confirming correct telemetry transfer and AI event reporting.
    -

Documentation and Criteria for Success

- All test results are logged in field notebooks and digital records, noting connectivity status, flight performance, cable/connector configurations, and telemetry quality.
- All hardware modifications, such as sensor additions, are tested for compatibility with Pixhawk I/O and MAVLink relay over RFD modems.
- The system is considered validated once autonomous flights perform as planned and the ground station reliably sends/receives commands and telemetry in both software environments.

Software Testing

Continuous Integration Testing Suite

To create a streamlined environment that would mitigate bugs in production, a continuous integration and deployment pipeline would be implemented into the GitHub repository. This pipeline would aim to help test and validate code changes in a pull request before merging into the main branch of the repository. GitHub workflows would be used to create this automation, as it allows for defined jobs to run in response to events, such as a pull request. These defined jobs are defined in a configurable YAML file.

Once the workflow is triggered, the job starts its setup by creating a runner with Ubuntu 22.04 as its image. Right after that, the runner proceeds to checkout of the branch and clones the repository. After it creates the space to start testing, it then installs any software or dependencies needed for the job, in this case it would be installing the Docker Buildx plugin in order to build Docker images. Now that everything needed to build and test the software has finished, the runner then begins building the Docker image, which comes from the Dockerfile already in the repository. Soon after that has finished the runner can now run the container and now the tests may begin.

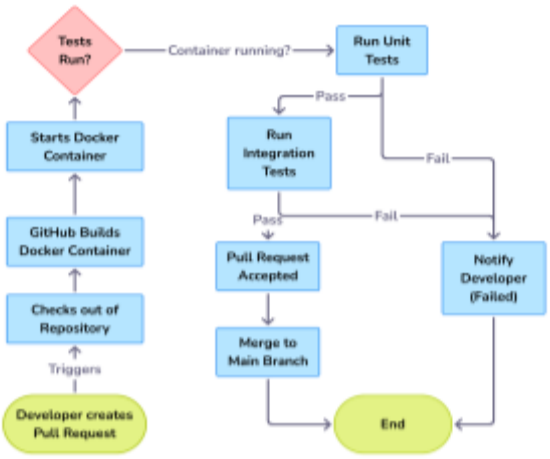


Figure 4.1: Proposed CI/CD Pipeline

As of now the only tests conducted are on whether the Docker container exists and if it is running, further testing would be split up into multiple aspects of the project, being:

1. Container health
2. VITALS functionality
3. Router connectivity

Ensuring that the containers are all running, communicating with each other, and taking in the proper input and output would require further unit and integration testing, as well as VITALS functionality, focusing on the JSON packet structure and content. In regards to the router connectivity, this would primarily encompass the MAVLink packets, testing whether the software can successfully send and receive packets, and or validating the contents of the packet conform to proper schemas.

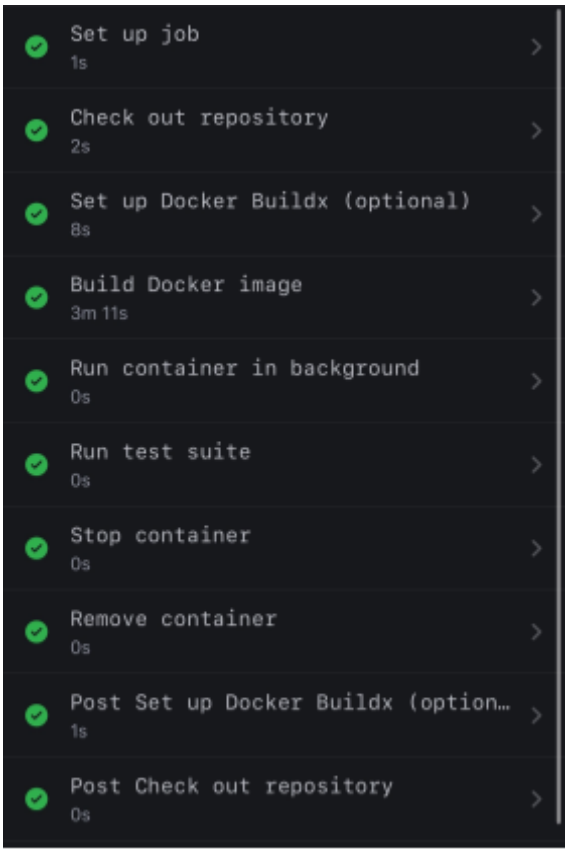


Figure 4.2: Docker Pipeline test results

When the testing concludes, one of two options can be sent back to the user, either a success or failure. If the testing concludes to be successful, then all the checks have been made to finally push the pull request into the main branch, and if the tests fail, the user simply receives the fail state. Whether the testing succeeds or fails, the runner concludes the workflow by performing proper cleaning tasks, such as stopping the container, removing the container, and cleaning up any caches or certificates.

Testing Files

The software team will incorporate unit testing as a standard when developing any new files or modifying existing files. Since our project is mainly using python, the team will become familiar with pytest. Pytest enables rapid, modular test development and maintains compatibility with standard Python tools, making it ideal for continuous integration and long-term maintainability of the VITALS codebase.

The testing plan focuses on the following core objectives:

1. Validate correctness of core modules such as path planning, mission state management, computer vision preprocessing, GUI logic, and MAVLink communication abstractions.
2. Ensure stability of asynchronous components including the Dispatcher event loop, telemetry handlers, and mission upload coroutines.
3. Verify integration points between VITALS and external systems such as Mission Planner, OpenStreetMap databases, and the multi-agent reasoning framework.
4. Detects regressions early through automated and repeatable test execution.
5. Support safe development cycles by ensuring changes do not negatively impact flight-critical logic.

The project will follow standard Python testing conventions by storing all tests in a top level tests/ directory. Individual test files will follow the naming pattern test_<module>.py and each will be standalone test implementations using assert. For example there will exist a test_agent_A.py file which will affirm that the agent can receive video feeds, call LLaVA and YOLO, and output MCP messages.

The testing pipeline will be integrated into the CI pipeline so that tests will automatically run upon each pull request, merge, and on scheduled builds. The team will dedicate one sprint to learning Pytest

Timeline

The following is a proposed software implementation timeline from now till March 23, 2026. This timeline is not intended to be rigorously followed and is expected to change dynamically. This is easy to work around with our Agile methodology that we describe later in the document. Be aware that this timeline was made with assistance from artificial intelligence per recommendation by UCF faculty.

Week 1-2: Development Environment Setup (Dec 1 - 14)

1. Set up Docker development environment with compose configuration
2. Establish container architecture for Agent A, B, C, D + supporting services
3. Configure shared Docker network for inter-agent communication
4. Set up CI/CD pipeline in GitHub with automated testing workflow
5. Establish development branches and merge strategy

Week 3: Core Protocol Implementation (Dec 15 - 22)

1. Agent D - Planning
    a. Define ACP (Agent Communication Protocol) JSON schema
    b. Implement ACP message validation and serialization
    c. Create basic Agent D service skeleton that accepts ACP intents
2. Agent A - Vision
    a. Define MCP (Model Context Protocol) JSON schema
    b. Refactor objectDetection.py into Agent A service structure
    c. Implement frame buffer system for video stream processing
3. GUI
    a. Complete GUI refactoring into modular architecture
    b. Implement MCP fact visualization in sidebar
    c. Create debug panel for protocol message inspection
4. Agent B - Memory
    a. Set up ChromaDB vector database in container
    b. Design event ingestion API endpoint

    c. Implement basic storage for MCP facts

Week 4-6: Winter Break Work - Asynchronous Work (Dec 23 - Jan 12)

1. Agent A - Vision
    a. Integrate YOLOv11 for continuous frame processing
    b. Implement keyframe selection algorithm
    c. Begin LLaVA integration for semantic captioning
    d. Create MCP fact generation pipeline
2. Agent B - Memory
    a. Implement embedding encoder
    b. Create RAG retrieval interface
    c. Build ChromaDB collection management
    d. Implement metadata filtering for queries
3. Agent C - Coordinator
    a. Decouple LLM from GUI ChatSidebar into standalone service
    b. Implement Ollama integration with LLaMA 3.2
    c. Create prompt template system for RAG context injection
    d. Build ACP intent generation logic
4. Agent D - Planning
    a. Wrap existing terrain preprocessing behind ACP interface
    b. Implement A* global path planning
    c. Create MAVLink mission upload module
    d. Build telemetry monitoring for dynamic replanning

Week 7-8: Integration Testing (Jan 13 - 26)

1. Mon Jan 13: Sprint planning meeting, review individual progress (First day back)
2. Integration testing of Agent A → Agent B (MCP facts storage)
3. Integration testing of Agent B → Agent C (RAG retrieval)
4. Integration testing of Agent C → Agent D (ACP intent translation)
5. Fri Jan 24: Demo Day 1 - Show MCP/ACP message flow through all agents

Week 9-10: Hardware-Software Integration (Jan 27 - Feb 9)

1. Deploy full VITALS Docker stack on Jetson Orin
2. Configure Mission Planner MAVLink mirroring
3. Test Agent D → MAVLink → Pixhawk command chain
4. Validate telemetry loop back to VITALS
5. Optimize Agent D for Jetson ARM64 architecture
6. Implement hardware status monitoring in GUI
7. Profile Agent A inference performance on Jetson
8. Optimize ChromaDB for embedded storage

Week 11-12: Natural Language Control (Feb 10 - 23)

1. Implement chat interface → Agent C → Agent B → Agent C loop
2. Create tool calling system for drone commands
3. Build context window management for mission history
4. Implement operator confirmation workflow for critical commands
5. Create mission intent library (search patterns, RTL, POI investigation)
6. Implement job priority queue integration with Agent D
7. Build waypoint type system (standard, takeoff, loiter)
8. Optimize Agent A frame rate based on detection confidence
9. Implement detection filtering to reduce false positives
10. Create POI creation workflow from high-confidence detections

Week 13-14: Simulation & Physical Testing (Feb 24 - Mar 9)

1. Run full mission simulations in Mission Planner
2. Test multi-drone coordination (if multiple SITL instances possible)
3. Validate terrain preprocessing with real OSM data
4. Test POI detection → investigation workflow
5. Stress test Agent B memory under extended missions
6. Physical drone test flights (controlled environment)
7. Test autonomous takeoff → waypoint navigation → RTL
8. Validate GPS coordinate accuracy from Agent A detections

9. Test failsafe procedures (communication loss, low battery)
10. Document all flight test results

Week 15: Final Polish & Documentation (Mar 10 - 16)

1. Bug fixes from testing phase
2. Performance optimization based on profiling data
3. Complete user documentation for operators
4. Create demonstration script for final presentation
5. Prepare backup demos in case of hardware issues
6. Record video demonstrations as fallback

Week 16: Panic Week (Mar 17 - 23)

1. Connect to drone via Mission Planner
2. Issue natural language command: "Begin search pattern"
3. Show Agent A → B → C → D message flow in real time
4. Display MAVLink waypoint upload to drone

Table 8: Tabular version of weekly timeline objectives

Week	Objective
1	Establish a Docker and CI/CD development environment with multi-container architecture and configure a shared network for inter-agent communication.
2	Complete development environment setup with all containers operational and communicating via JSON protocols.
3	Define and implement MCP and ACP JSON schemas with validation across all agents. Create skeleton services for Agents A-D that can exchange protocol messages over Docker network, completing foundational communication layers.
4	Individual asynchronous development during winter break
5	Continue independent agent development winter break
6	Complete individual agent implementations and begin local testing.
7	Integration testing begins with a full team back from break - validate Agent A → B (MCP storage), B → C (RAG retrieval), and C → D (ACP translation) message flows.

8	Complete end-to-end integration testing from detection to waypoint generation with all agents communicating successfully.
9	Hardware-software integration sprint focuses on deploying VITALS stack to Jetson and establishing Mission Planner MAVLink connection with container.
10	Successfully send first MAVLink waypoint mission from VITALS to a physical drone, achieving a critical MVP milestone.
11	Implement natural language control system with chat interface integrated into Agent C reasoning loop and RAG context retrieval.
12	Complete natural language command pipeline with mission intent library and job priority queue integration. End-to-end testing validates operator can issue commands like "investigate northeast corner" resulting in autonomous drone waypoint execution.
13	Run comprehensive mission simulations in Mission Planner SITL testing multi-drone coordination and terrain preprocessing with real OSM data.
14	Conduct physical drone test flights in controlled environment validating autonomous takeoff, waypoint navigation, and RTL sequences. Test GPS coordinate accuracy, failsafe procedures, and document all results for Go/No-Go decision on final demonstration readiness.
15	Final polish sprint with bug fixes, performance optimization, and complete operator documentation.
16	Demonstration week

Project Management

For VITALS to succeed, or any project for that matter, the team working on it must be well coordinated and organized. To ensure progress is steady and frequent, we spent the first 2 weeks establishing team organization and creating the necessary environments for communication, management, and development not only amongst ourselves but also with our sponsor. We created a time keeping excel spreadsheet for recording team meetings, a confluence page for documentation and meeting notes, a jira board for task assignments and sprint planning, and a discord server for fast casual communication between team members. In this section we will detail each of these and how we plan to use them going forward into the next stage of VITALS. We heavily encourage any future VITALS teams to adopt a similar system to ours as we have found great success with it.

Organization

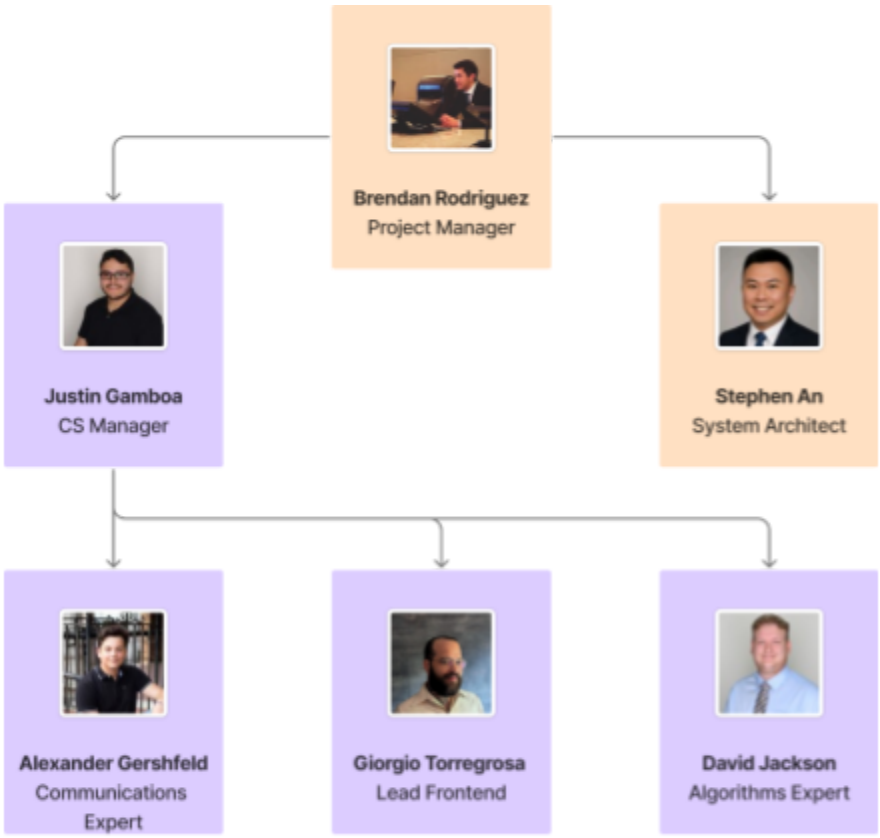


Figure 5.1: VITALS Org Chart

Meetings

Team Meetings

We discussed how meetings should go as a team and used when2meet.com as an easy free coordination tool. Our members filled out which times worked best for them and we settled on having consistent weekly meetings every Monday at 8pm with an additional meeting slot on Saturday at 12pm if we deemed it necessary. Provided is a screenshot of that Excel spreadsheet. Certain portions of the sheet have been cut off to meet design document specifications.

	A	B	C	D	E	F	G
1	VITALS Meeting Tracking Sheet				CONFLUENCE		ZOOM
2	Week	Date	Meeting Lead	Meeting Type	Agenda (Link)	Minutes (Link)	Recording (Link)
3		9/8/2025	Brendan Rodriguez	Team	N/A	N/A	N/A
4	1	9/13/2025	Brendan Rodriguez	Team	N/A	N/A	N/A
5		9/15/2025	Brendan Rodriguez	Team	N/A	N/A	N/A
6		9/18/2025	Brendan Rodriguez	SME	Agenda Link	Minutes Link	N/A
7	2	9/20/2025	Justin Gamboa	Sprint Planning	Agenda Link	Minutes Link	N/A
8		9/22/2025	Brendan Rodriguez	SME Team	Agenda Link	Minutes Link	N/A
9	3	9/27/2025	Justin Gamboa	Sprint Review	Agenda Link	Minutes Link	N/A
10		9/29/2025	Brendan Rodriguez	Sprint Planning	Agenda Link	Meeting Link	Zoom Recording
11	4	9/30/2025	Justin Gamboa	Team Client	Agenda Link	Minutes Link	
12		10/4/2025	Justin Gamboa	Team	Agenda Link	Minutes Link	
13	5	CANCELLED	Giorgio Torregrosa	Team	N/A	N/A	N/A
14		10/7/2025	Brendan Rodriguez	Sprint Planning	Agenda Link	Minutes Link	Zoom Recording
15	6	10/11/2025	Brendan Rodriguez	Team	Agenda Link	Minutes Link	Zoom Recording
16		10/13/2025	Brendan Rodriguez	Team	Agenda Link	Minutes Link	Zoom Recording

Figure 5.2: VITALS Meeting Tracker Google Sheet

These meetings were held in discord at first but later we moved to zoom for easy recording and transcripts for review. These meetings will be continued until our time with VITALS as a group is over.

Agendas & Minutes

For each meeting, our team creates an agenda and then minutes after the meeting is over. These help us prepare and document what the meeting was for and what was accomplished for that meeting. Agendas are created before the meeting starts and states what we expect to achieve during the meeting. The agenda is available to all team members for review on confluence. The minutes documents are stored in our team confluence for easy access and are filled out using the recordings and transcripts from zoom after the meeting is over

Sponsor Meetings

Joseph Rivera is our Lockheed Martin Sponsor. He was able to meet with us for the first time on September 30, 2025. In that meeting we outlined our questions, comments and concerns regarding VITALS and set up recurring biweekly meetings with him on Tuesday at 1:30pm. These meetings occur remotely via Microsoft Teams. In these meetings we discuss project status and ask any questions which may have occurred during research. Recently, our team has shifted from research to development and we expect these meetings to shift in tone as well.

In Person meet ups

During the course of the project, several team members have had the privilege of meeting Joseph at the Lockheed Martin Missiles and Fire Control campus. We were given a tour of some of the facilities and we allowed supervised access to the Applied Research Lab to acquire some hardware our computer engineers needed. We have also met up as a team in person a few times. Most of these occurred in the Harris building at UCF. However, one of these meetings was held at our team member Giorgio's house. That meeting was more for team bonding but we did bring the drone and attempted to launch it using just Mission Planner. That didn't work and we encountered some failure to arm errors from the drone which we figured out was the result of a low power battery and poor GPS connection.

Agile

The computer science department required that all senior design teams use agile development methodology for software development. To meet this requirement we utilized Jira to assign tasks and created sprints for research and implementation tasks. Our backlog has been consistently changing throughout the course of the project and we will continue modifying it as we see fit.

Sprints

We have had a few sprints since the project officially started after our first sponsor meeting with Joseph. Sprints are started by either Justin or Brendan. The items are picked from the backlog and our team works on completing them throughout the time frame devoted to that sprint. At the end of each week the team has a quick sprint check in or review depending on the

status of the deadline. During the week the team is encouraged to utilize the standup discord bot. We decided to use Dailybot, DailyBot is an automation platform and chat assistant for your work. It takes chat to a new level by providing automation and chat use-cases that are underserved by the current messaging platforms. It helps users get more done in chat, be more efficient, and save time with daily standups, forms in chat, recognition with kudos, random virtual coffees and special add-ons that you can build yourself. The team will be required to use the Daily bot more actively during SD2.

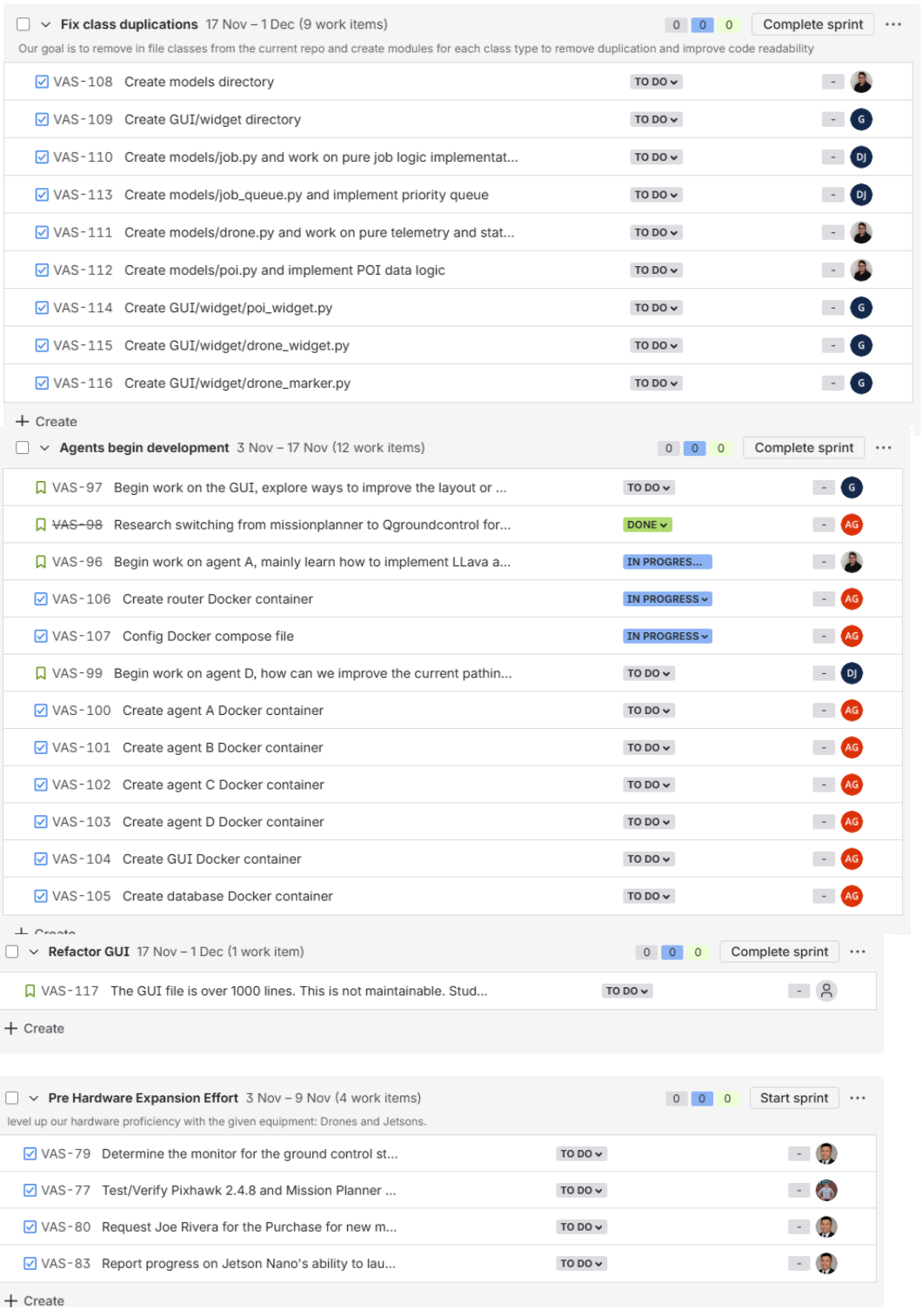
Table 9: Current Sprints

Sprint	Date	Description
1	2025/09/22 - 2025/09/29	Review and study assigned sections of the L26 design document. Each team member will complete their assigned section by September 27, 2025.
2	2025/09/29 - 2025/10/06	Every team member will set up their local development environment by cloning the VITALS repo, creating an individual branch named after their first name, installing all required dependencies, and successfully launching the VITALS GUI (missionState.py). This sprint ensures everyone can contribute code and validate that the application runs locally, laying the groundwork for further collaborative development.
3	2025/10/06 - 2025/10/20	Each member will be assigned a portion of the current code base to review and document. The previous team did not leave us with a lot of additional documentation so we must be thorough
4	2025/11/03 - 2025/11/09	Level up our hardware proficiency with the given equipment: Drones and Jetsons.
5	2025/11/17 - 2025/12/01	Our goal is to remove in file classes from the current repo and create modules for each class type to remove duplication and improve code readability

Sprints will continue and we will begin enforcing the daily stand ups as a requirement for all members. Future sprints will be based on the previously mentioned project timeline which can be seen in Table 7.

Backlog

The following pages consist of only screenshots from the team's Jira backlog to meet document requirements laid out by UCF faculty.



Pre Software Expansion Effort 3 Nov – 10 Nov (8 work items) 0 0 0 Start sprint

initial documentation and review of assigned files within the repo has been completed. Issues have been identified. Propose and or consider any re-factorization /...

VAS-84	Determine what kind of modification if any need...		TO DO		-
VAS-85	Determine what kind of modification if any need...		TO DO		-
VAS-86	Determine what kind of modification if any need...		TO DO		-
VAS-87	Determine what kind of modification if any need...		TO DO		-
VAS-88	Determine what kind of modification if any need...		TO DO		-
VAS-89	Determine what kind of modification if any need...		TO DO		-
VAS-90	Test and Validate Docker Fresh Install on untest...		IN PROGRESS		- AG
VAS-93	Locate funding for the Drone	ADD A SECOND DRON...	IN PROGRESS	Nov 7	-

+ Create

Backlog (97 work items) 0 0 0 Create sprint

VAS-61	Document app/Dispatcher to understand purpose		TO DO		- G
VAS-71	Run software on Docker		IN PROGRES...		- AG
VAS-62	Document app/GUI to understand purpose		IN PROGRES...		- G
VAS-95	Launch QGroundControl (QGC) and connect to VITALS to QGC		IN PROGRES...		- AG
VAS-118	Setup CI/CD pipeline		IN PROGRES...		- AG
VAS-127	Create unit tests for the YOLO model pipeline	AGENT A	TO DO		-
VAS-129					-
VAS-130					-
VAS-131					-
VAS-132					-
VAS-133					-
VAS-134					-
VAS-135	Establish data cable connection between QGC on Jetson and SOAR Drone				-
VAS-136	Establish Radio Modem Connection between QGC on Jetson and SOAR Drone				-
VAS-137	Confirm ZED and FLIR Cameras are working				-
VAS-138	Define ACP (Agent Communication Protocol) JSON schema	AGENT D	TO DO		-
VAS-139	Implement ACP message validation and serialization	AGENT D	TO DO		-
VAS-140	Use VITALS software to connect to a real drone		TO DO		-
VAS-141	Create basic Agent D service skeleton that accepts ACP intents	AGENT D	TO DO		-
VAS-142	Set up Docker development environment with compose confi...		TO DO		-
VAS-143	Establish container architecture for Agent A, B, C, D + suppor...		TO DO		-
VAS-144	Configure shared Docker network for inter-agent communicat...		TO DO		-
VAS-145	Set up CI/CD pipeline in GitHub with automated testing workf...		TO DO		-
VAS-146	Establish development branches and merge strategy		TO DO		-
VAS-147	Order Monitor for Ground Control Station		TO DO		-
VAS-148	Define MCP (Model Context Protocol) JSON schema	AGENT A	TO DO		-
VAS-149	Order Keyboard for Ground Control Station		TO DO		-
VAS-150	Refactor objectDetection.py into Agent A service structure	AGENT A	TO DO		-
VAS-151	Implement frame buffer system for video stream processing	AGENT A	TO DO		-
VAS-152	Complete GUI refactoring into modular architecture	GUI	TO DO		-

VAS-155	Set up ChromaDB vector database in container	AGENT B	TO DO	-
VAS-156	Design event ingestion API endpoint	AGENT B	TO DO	-
VAS-157	Implement basic storage for MCP facts	AGENT B	TO DO	-
VAS-158	Integrate YOLOv11 for continuous frame processing	AGENT A	TO DO	-
VAS-159	Implement keyframe selection algorithm	AGENT A	TO DO	-
VAS-160	Begin LLaVA integration for semantic captioning	AGENT A	TO DO	-
VAS-161	Look into MAVProxy for testing commands without GUI		TO DO	-
VAS-162	Create MCP fact generation pipeline	AGENT A	TO DO	-
VAS-163	Implement embedding encoder	AGENT B	TO DO	-
VAS-164	Create RAG retrieval interface	AGENT B	TO DO	-
VAS-165	Build ChromaDB collection management	AGENT B	TO DO	-
VAS-166	Implement metadata filtering for queries	AGENT B	TO DO	-
VAS-167	Decouple LLM from GUI ChatSidebar into standalone service	AGENT C	TO DO	-
VAS-168	Implement Ollama integration with LLaMA 3.2	AGENT C	TO DO	-
VAS-169	Create prompt template system for RAG context injection	AGENT C	TO DO	-
VAS-170	Build ACP intent generation logic	AGENT C	TO DO	-
VAS-171	Perform object recognition tests on cameras connected to Je...		TO DO	-
VAS-172	Wrap existing terrain preprocessing behind ACP interface	AGENT D	TO DO	-

To avoid bloating the document we will stop here. There are about 30+ additional items in the backlog that have been omitted for this reason.

Motivations

The following are short messages from each of the current VITALS team members. We describe our personal motivations and aspirations for this project and our experience thus far.

Brendan Rodriguez

As a senior computer engineering student at UCF, I continually seek opportunities to expand my technical skill set and apply classroom knowledge to real-world challenges. Throughout my academic journey, from dismantling electronics in my childhood bedroom to conducting GPU benchmarking research for post-quantum cryptography at UCF's DRACO Lab, I have been driven by a fundamental curiosity about how systems work and how they can be made better. The VITALS project immediately caught my attention because it intersects several of my core interests: embedded systems, AI-powered autonomy, hardware-software co-design, and wireless communication. My past experiences in hardware design, signal analysis, and machine learning have taught me that the most rewarding engineering problems are those that demand innovative solutions across multiple domains.

When I first joined the VITALS team as Project Manager and hardware lead in Fall 2025, I found myself facing a unique challenge. Our interdisciplinary team, composed of two computer engineering students and four computer science students, was tasked with developing an autonomous search and rescue drone system sponsored by Lockheed Martin. Despite having limited drone experience at the outset, I embraced the opportunity to become our team's subject matter expert in UAV hardware integration. What excited me most was not just the technical complexity, but the real-world impact: building technology that could save lives in search and rescue operations.

The VITALS project represents more than just a senior capstone requirement. It embodies the kind of engineering work that drew me to computer engineering in the first place, work that requires deep understanding of both the physical and computational layers of complex systems. Our mission is to create an AI-powered ground control station capable of processing real time

video feeds from autonomous drones, performing computer vision analysis to detect persons of interest, and providing tactical decision support to SAR operators in the field. This requires seamless integration between edge computing hardware, flight controllers like the Pixhawk, wireless telemetry systems, and sophisticated machine learning models, a challenge that spans my entire technical education.

My technical preparation for this role began well before VITALS. Working as an undergraduate researcher in the DRACO Lab, I gained hands-on experience with GPU computing, CUDA programming, and hardware signal analysis. My work on benchmarking post-quantum cryptography algorithms, specifically ML-KEM implementations on GPU architectures, taught me how to optimize performance at the hardware-software boundary. I learned to analyze power consumption, thermal management, and computational efficiency in resource-constrained environments. These skills translate directly to VITALS, where we must deploy AI inference on low-power edge devices capable of field operation. Understanding how to squeeze maximum performance from limited hardware resources has become one of my key contributions to the team.

Beyond research, my coursework in embedded systems, requirements engineering, and hardware design provided the theoretical foundation I needed to tackle VITALS' technical challenges. But theory alone doesn't prepare you for the reality of inherited hardware, inconsistent documentation, and the inevitable troubleshooting that comes with integration work. When our team first received the SOAR drones and ground station equipment from the previous cohort, we quickly discovered that much of the system was undocumented or only partially functional. Radio frequency modules were mismatched models that wouldn't pair. Flight controller firmware needed updating. MAVLink communication protocols, the backbone of drone-to-ground-station coordination, required extensive debugging before we could achieve reliable telemetry.

Rather than viewing these obstacles as setbacks, I recognized them as invaluable learning opportunities. Working alongside my teammates, Justin, Giorgio, Tyler, Alex, and Stephen, and

with mentorship from industry professionals Josh and Aaron at Duke Energy, as well as AI specialist Issam, I developed a systematic approach to hardware integration and troubleshooting. We established a collaborative workflow using Jira for sprint planning and task tracking, Confluence for technical documentation, and Git for version control. Each challenge we solved became documented knowledge for future teams. Each firmware flash and successful radio pairing represented not just technical progress, but growth in our ability to work as a cohesive engineering team.

The interdisciplinary nature of VITALS has been one of its greatest strengths. As the hardware specialist working alongside computer science students focused on software architecture, AI model development, and user interface design, I've learned the critical importance of clear communication across technical domains. When I'm selecting a single-board computer for edge AI inference, I need to consider not just processing power and thermal constraints, but also how it interfaces with the software stack my CS teammates are building. When we're planning autonomous flight missions in Mission Planner, I need to understand both the hardware limitations of our Pixhawk flight controller and the data requirements of our computer vision pipeline. This constant translation between hardware and software perspectives has made me a more versatile engineer.

I am particularly motivated by the prospect of building an integrated drone platform that leverages both my technical strengths and my curiosity for AI-driven technology. The VITALS project pushes me to synthesize knowledge from multiple courses, such as, digital systems, embedded programming, wireless communication, machine learning, and apply it in concert. Designing the ground control station requires considering power distribution, sensor integration, thermal management, and electromagnetic compatibility. Implementing autonomous flight requires understanding MAVLink protocols, GPS navigation, flight dynamics, and fail-safe mechanisms. Deploying real time computer vision demands knowledge of neural network architectures, inference optimization, and edge computing constraints. No single course prepared me for all of this; VITALS required me to integrate everything.

What makes this project particularly meaningful is its alignment with my long-term career aspirations. I don't want to be an engineer who specializes narrowly in either hardware or software. I want to work at the intersection of building intelligent systems where the boundary between physical and computational becomes blurred. Whether that's developing autonomous vehicles, designing robotics platforms, creating IoT security devices, or building next-generation embedded AI systems, the future of technology lies in seamless integration across domains. VITALS is training me for exactly that future.

The opportunity to collaborate with a multidisciplinary team, tackle complex system integration, and contribute to a project with meaningful, real-world applications in search and rescue is precisely the type of challenge that excites me at this stage of my academic and professional journey. Every time we successfully test a new autonomous flight pattern, every time our AI model correctly identifies a target in the video feed, every time we solve an integration bug that's been blocking progress, these moments reinforce why I chose engineering. They remind me that behind every line of code and every circuit is the potential to make a tangible difference in people's lives.

Looking ahead to Spring 2026 and the completion of VITALS, I'm focused on achieving full autonomous operation with integrated AI decision-making. We're working toward a system where drones can be deployed with minimal human intervention, automatically survey designated areas, process video in real time, and alert operators to potential persons of interest. The path from our current state to that goal involves solving numerous technical challenges: improving GPS navigation accuracy, optimizing our neural network for edge deployment, ensuring reliable wireless communication at extended ranges, and implementing robust failure recovery mechanisms. Each of these challenges represents an opportunity to deepen my expertise and prove that our team can deliver a production-ready system.

Ultimately, I joined the VITALS team because I believe this project will allow me to learn new technologies, enhance my collaboration skills, and prepare for a career where the fusion of hardware, software, and machine intelligence will be critical. As we approach

graduation in Spring 2026, I'm confident that the skills I've developed, technical troubleshooting, system integration, project management, cross-functional collaboration, and leadership under uncertainty, will serve me well in whatever engineering challenges come next. VITALS has transformed me from a student who learns from textbooks into an engineer who solves real problems. That transformation, more than any grade or credential, is the true measure of my education at UCF.

Stephen An

I chose to be part of the VITALS senior design project because I wanted to step outside of my comfort zone and challenge myself with something completely new. Most of my academic and personal work has centered around analog and digital hardware design, so working on a project rooted in artificial intelligence, drone autonomy, and interdisciplinary collaboration is a unique opportunity for growth. I was also drawn to the chance to work closely with team members from other departments, since I believe diverse perspectives drive more innovative and practical solutions. By participating in VITALS, I expect to broaden my skill set, strengthen my ability to collaborate across disciplines, and prepare myself for future engineering roles that increasingly require integration of hardware, software, and AI.

Experience in Group Projects

I have participated in multiple group projects throughout my coursework. In COP 4710 (Database Systems), I worked on a multi-phase project that required designing and implementing a relational database, from conceptual E-R diagrams to SQL scripts and a final implementation report. This experience gave me hands-on practice in breaking down large projects into deliverables, coordinating roles within a team, and integrating individual contributions into a cohesive system. I also worked on projects in COP 4331 (Object-Oriented Software Development), which emphasized UML diagramming, requirements gathering, and managing the full software development lifecycle. These projects required clear communication, version control discipline, and accountability to deadlines. Through both courses, I learned how to manage the challenges of collaborative design and how to balance technical detail with broader project goals.

Experience Related to VITALS Technology

My strongest background is in analog and digital hardware design, built through courses in circuits, digital systems, and embedded systems. I am comfortable with designing and troubleshooting hardware at the component level and understand how to optimize performance, reliability, and power efficiency. However, I have little direct experience with AI or machine learning technologies. This lack of exposure is precisely why I wanted to join VITALS. The project requires integrating low-power AI agents on platforms such as the NVIDIA Jetson Orin to control drone swarms for search-and-rescue. While the AI software stack is new to me, I bring complementary expertise in hardware integration, power management, and systems reliability. Areas that are critical for the success of the project. By contributing my hardware knowledge while learning AI frameworks and tools along with my teammates, I hope to bridge the gap between embedded hardware and modern AI applications.

David Jackson

My motivations for joining the project stem from several key factors that have been building up to my acceptance into this project. To begin, I have a deep fascination with drone light shows, mainly the idea of a drone swarm being controlled from a base station to execute a task. I have often thought about the various applications of drone swarms and have been extremely fascinated by the extensive use of commercial drones in modern-day warfare settings.

When I entered UCF in 2023, I was doing independent research on drone applications in emergency SAR and swarm applications just to see what the modern drone research was being focused around. The research that I saw at the time was heavily focused around drone delivery systems that are intended to deliver supplies and other needed materials to disaster areas affected by natural disasters such as forest fires and earthquakes. As the growth of AI continues in every industry, a question continues to grow in my mind: how are you able to combine a drone's flight capabilities with AI's powerful computing capabilities.

The question of how Artificial Intelligence is going to impact these fields I was interested in continued to linger within my mind and guided my studies at the University of Central Florida for the next 3 years. Within these past 3 years I have taken elective courses such as Artificial Intelligence, Advanced Artificial Intelligence, and Machine Learning. Furthermore, as these

fields are very math intensive I have also focused my studies with this in mind by acquiring a math minor and physics minor alongside my computer science major. Within these minors I have taken classes such as Numerical Computing 1 & 2, Linear Algebra, and Mathematical Foundations for Machine Learning. I am doing my best to aim my studies and career at the inspirational and challenging field of Machine Learning and Artificial Intelligence development and I am confident with my background in this field for this project and my future career.

At the beginning of August 2024, I started an autonomous drone project called Project Kestrel, which allowed me to conduct independent research into autonomous drone flight using an onboard Jetson computer. This project has since been able to achieve Blue Origin sponsorship and finalize the designs, implementations, and started test flights. Through this project I gained research experience with complex autonomous tracking architectures such as DeepSORT which utilizies YOLO as its object detection model. Furthermore, I gained experience with embedded programming through the Jetson, motors, and other various onboard components. Finally, the most important thing that will aid my in this project is the experience I gained with flight simulations and flight path planning. Rather quickly we were able to get a fully working simulation created within SITL, ROS2, and Gazebo that simulated all of our sensors and cameras allowing us to test our entire software suite. This highlighted the importance of simulation as once we got the simulation running we heavily adapted many of our methods as we continued to see the faults in our original plans. Additionally, I recruited many robotics focused computer science students to aid with the pathing for this project. With their aid, I began to realize how heavily focused robotics pathing is within my AI class. Within the project we used primarily D*, which is a refined version of A* focused towards robotics, both these algorithms were taught to me within my AI classes. The hardest challenge within this project was the sensor fusion between the lidars, ultrasonic, and camera.

I continued my research, learning about object detection, tracking, flight pathing, and more, leading me to the beginning of August 2025. I heard about this project and was immediately intrigued and wanted to join as this projects technical applications and implementations directly align with Kestrel. This project has research in several different fields that I am interested in, including: Drone Swarm Control, LLM configuration, YOLO, Jetson

Hardware, and more. One aspect of this project I am extremely interested in is the use of Ollama on the jetson hardware and the creation of agents to control the swarm. Currently the plan we have setup within the MVP is to create a 4 agent architecture assigning each agent to a specific task, mine being the pathing and planning agent. With the extreme innovation shown through LLMs I am very excited to learn how to implement these within a mission ready real time scenario such as VITALS. Something that I am excited to learn is how to make the raw data, such as the camera or geo spatial information, usable for the LLMs to make correct and accurate mission critical decisions. When reviewing the prior teams work on path planning they do not have any obstacle avoidance or real path planning, just a way point generation. This is going to be a major hurdle I need to tackle as having drones crash due to incorrect path planning is a very big problem. Within Kestrel we had the assistance of many sensors to aid with local path planning however the drones given to us don't have any other sensors other than a camera attached using a zip tie. This will be a difficult challenge to tackle but a very interesting challenge to try and accomplish primarily using swarm coordination and relying solely on CV.

I believe that the most fascinating thing to me about deep learning research is learning how to make accurate predictions for futures based on a given dataset. For example, given the position of the drone, current sensor data, and next given path waypoints are you able to make the next best move in order to keep the drone in the air moving towards its path. I would like to learn how to apply this school of thought to other fields such as quantitative development and research or continuing research and development at a military/government contractor. I believe the hardest part of this field is data science, learning how to clearly format your data and parse your data so that your algorithm (or in VITALS case the agent) can properly understand the raw data provided. I have often thought about how we are able to quantify information such as political decisions to predict outcomes on stocks or businesses and how we can get this to provide a more clear outcome for market predictions but this is a very unclear train of thought. Having raw quantifiable data such as sensor, camera, or other numerical data is the first step to learning how to get into the field of predicting futures.

Alex Gershfeld

My drive to work on this project comes from a newly founded appreciation for how digital systems can translate real world sensory data into information that can be analyzed and for this project specifically, maybe even save lives. The link between this appreciation, my skills in working with technology, and the involvement in something greater than myself keeps me motivated to transcend the challenges of taking academic knowledge and applying it to industry level work and standards.

All throughout my years studying computer science, I found it difficult to see how learning about niche parts of computers would help me in industry. For the longest time I believed that the knowledge I have attained would be better suited in the research track of my major. Now that I am reaching the end of my studies I'm beginning to find where all of these parts of a whole combine, not just as a developer, but as a well-rounded Computer Scientist. The root of my greater understanding of the process stems from my elective robot vision class. Seeing and actually creating a product capable of solving real world issues made it more clear what I would be doing in this field. Although my exposure to this genre of computer science is somewhat limited in comparison to others that I would be working with, I know that my motivation to succeed in this project will guide me in the right direction.

I hope to build my skills with AI and computer vision in order to provide society with more examples of AI being used for the greater good. This project seems to be a good fit for the grand scheme of things. I have a good understanding of how LLM models are run and built and have experience with several Python modules that allow me to fully develop or integrate models into our design.

This project holds a place dear to me, as it has potential to do great things for our society. While there are a plethora of polarizing opinions on AI, it brings me great pride to think that I am helping to balance out the bad implications of AI on society by creating for the objectively good. With a project of this significance, a part of me thinks that we may be in too deep, or that we may not create a finished product that can be shipped immediately in disaster situations, but yet another part of me feels extra motivated to even make some contribution to this greater effort.

My thoughts on the current state of the project make me believe that our group would be near the end, or the end of the building of this multi-year project. I have high hopes that our group would be the group to put this project into production, especially with coordination with our ECE students on the project. Based on documentation, research and code from previous groups it seems that the main goal is to optimize the current object detection software, flight controls, and a high priority on drone physical hardware. With that in mind I would hope to work on the connectivity and foundation of the project, making sure the drone swarm and ground control station can effectively and reliably communicate with each other.

Giorgio Torregrosa

The convergence of artificial intelligence, autonomous systems, and humanitarian technology represents one of the most compelling frontiers in modern computer science. The VITALS project embodies everything that drew me to this field. As a computer science student with minors in mathematics and physics, focusing my coursework on AI/ML and robot vision, I see this project as the perfect synthesis of theoretical knowledge and practical application. The challenge of developing autonomous drone swarm systems for search and rescue operations goes beyond traditional academic boundaries, requiring not just technical expertise but a deep understanding of how technology can serve humanity's most critical needs. My role as the primary GUI developer for this project has given me unique insight into how human operators interact with complex autonomous systems, and how thoughtful interface design can mean the difference between a successful rescue and a missed opportunity to save lives. My fascination with defense technology and companies like Lockheed Martin comes from their commitment to pushing the boundaries of what's technically possible while maintaining the highest standards of reliability and safety. Working on a project sponsored by Lockheed Martin provides an invaluable window into the rigorous engineering practices and innovative thinking that characterize the defense industry. The VITALS project exemplifies the type of dual use technology that excites me most about this sector, where capabilities developed for defense applications can directly translate into life saving civilian tools. The discipline required to meet aerospace and defense standards has sharpened my approach to software development, particularly in designing interfaces that must function flawlessly under pressure. Every line of

code I write, every interface element I design, and every system integration I implement must meet the exacting standards that human lives depend upon, whether in military operations or civilian search and rescue missions. The technical challenges I've tackled in developing and refactoring the VITALS GUI have deepened my appreciation for the complexity of human machine interaction in critical systems.

When I inherited a gigantic 1,262-line GUI file, I saw not just a coding challenge but an opportunity to fundamentally reimagine how operators interact with autonomous systems under stress. The comprehensive refactoring I undertook, transforming this unwieldy codebase into a modular component based architecture with clear separation between entities, widgets, and pages, represents more than just good software engineering practice. It reflects my understanding that in high stakes scenarios, cognitive load must be minimized, information must be presented with absolute clarity, and system responses must be instantaneous and predictable. Through extensive testing and refinement, I've developed interfaces that allow operators to define search areas through intuitive polygon drawing tools, monitor multiple drone telemetry streams simultaneously, and interact with AI systems through natural language while maintaining full situational awareness. Each enhancement, from optimizing map rendering performance to implementing real time MAVLink communication visualization, directly impacts operational effectiveness when seconds count. The transition from simulation to hardware implementation represents the most exciting phase of this project and aligns perfectly with my vision for practical, deployable autonomous systems. While the previous team built an impressive theoretical framework with sophisticated coordinate estimation algorithms and LangGraph integration, the true test lies in adapting these systems to the harsh realities of field deployment.

Working with Jetson Nano processors and actual drone hardware introduces constraints that simulations cannot fully capture, including computational limitations, network latency, GPS drift, and power management considerations. My approach focuses on creating robust interfaces that gracefully handle these real world challenges, implementing fallback mechanisms for partial system failures, and ensuring that operators maintain control even when autonomous systems encounter edge cases. The GUI must not only display information but actively assist operators in

making rapid decisions, flagging anomalies, suggesting alternative strategies, and maintaining clear communication channels between human judgment and machine capability.

The integration of Large Language Models into the command and control interface represents a paradigm shift in how we interact with autonomous systems, and my work on the ChatSidebar component and LLM with MAVLink translation pipeline positions me at the forefront of this evolution. By enabling operators to issue complex multidrone commands through natural language, we're removing the traditional barriers between human intent and machine execution. However, this capability brings tremendous responsibility to ensure that the system correctly interprets commands, validates them against safety parameters, and provides clear feedback about what actions will be taken.

My implementation includes sophisticated context management to maintain conversation state across multiple commands, visual confirmation of interpreted instructions before execution, and real time feedback showing how natural language translates into specific MAVLink commands and waypoint sequences. This work requires deep understanding not just of user interface design but also of the underlying drone communication protocols, path planning algorithms, and the critical importance of maintaining human oversight in autonomous operations. Looking forward, my vision for the VITALS project extends beyond current capabilities to embrace emerging technologies and operational paradigms. The modular architecture I've implemented provides a foundation for integrating real hardware, machine learning models that adapt search patterns based on terrain and historical success rates, and distributed command structures that allow multiple operators to coordinate large scale search operations. The potential for this technology to save lives motivates every technical decision I make, from optimizing class components for faster POI retrieval to implementing intuitive UI designs that anticipate operator needs.

Working on VITALS has reinforced my commitment to developing technology that matters, where academic excellence meets real world impact, and where the principles learned from defense applications can be applied to humanitarian purposes. This project represents not just a senior capstone but a defining moment in my career trajectory, demonstrating that with

rigorous engineering, thoughtful design, and unwavering focus on the mission, we can create autonomous systems that amplify human capability rather than replace human judgment, ultimately saving lives that might otherwise be lost.

Justin Gamboa

When I first decided to take on this project, I viewed it as a convenient bridge, something that could tie my current CWEP position into a senior design effort and, in the process, help me build even stronger connections with my employer. While that's still true, the project has grown into something far more compelling for me on a technical level than I originally expected. My motivations have stayed consistent with what I laid out in my team contract: I want to graduate, I want to keep learning, and I want to set myself up to succeed long after my degree. But with everything I've learned and continue to learn from VITALS, those goals no longer feel distant or theoretical. They feel close, tangible, almost within reach. At this point, the only remaining step is to actually stretch out my hand and take them.

Before senior design, we took a web development course that ultimately left me pretty disappointed. I simply didn't enjoy web development, and I still don't consider many of the systems or tools used in that domain to be interesting from a mathematical or practical standpoint. The one part that did grab my attention was the security and server administration side of things, which was essentially off the table for us. We were told not to worry about it, to skip over the real infrastructure questions, to just use plain HTTP instead of HTTPS "for convenience." And while I technically could have gone off on my own and dug into those topics anyway, I was busy with another class that captured my interest a lot more: Professor Mell's AI for Game Programming course.

That class was only six weeks long, but in those six weeks my team planned, built, and polished a functional 2D action roguelike packed with several AI-driven enemies. I was the one responsible for imagining how those enemies should think and behave, and then turning that imagined behavior into actual in-game logic. That process was exciting in a way that web development never was. I enjoyed learning about pathfinding algorithms, state machines, behavior trees, whatever tools the AI needed and then watching those enemies react exactly the

way I had planned. It created a satisfying feedback loop where my ideas weren't just abstract; they were visible, reactive, and alive on the screen. Seeing that immediate, tangible result from my own decisions was motivating, and it reminded me what kind of work energizes me the most.

VITALS gives me that same feeling again. The project's use of an AI system to command a drone swarm through computer vision feels like a natural extension of everything that fascinates me: intelligence, autonomy, emergent behavior, and systems that respond in real time. It's the kind of work where you can point to something concrete and say, "I built that and it works."

By coincidence, my time as a CWEP has put me right in the middle of the object detection model training pipeline. I've spent months becoming familiar with how those models behave, how they fail, how they can be improved, and how small adjustments in training can produce major downstream differences. That experience is now guiding me through VITALS and reinforcing the decisions I'm making along the way. It feels, in a phrase, like my worlds are finally aligning. The skills I build at work strengthen my work on VITALS, and the skills I build on VITALS strengthen what I bring back to my job. And on top of that, I now have the opportunity to pick up new knowledge in the LLM space through the Ollama integration we're aiming to complete.

The previous senior design team made good progress and hit several important milestones, but one area that still feels hollow is the agentic behavior of VITALS. Most of the chat messages shown in the GUI are hardcoded responses rather than genuine model-driven reasoning. It was disappointing to discover that, but in a strange way it was also encouraging. Everything else surrounding that module is robust enough that our team doesn't need to waste time fixing fundamentals. We can go straight to addressing the LLM gap and start building the actual intelligence the system deserves.

The vision model will require a similar amount of care. Right now, it's more of a placeholder than a reliable component. But because LLaVA is essentially a multimodal LLM with built-in visual reasoning, it feels like the vision problem and the language problem belong

to the same orbit. That makes both of them feel more approachable and more exciting to dive into. Overall, I'm hopeful and energized about the future of this project. We've laid out a clear path forward, we understand the parts that need work, and we're finally in a position to begin executing the timeline we've built. I'm ready to get started.