WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

Etocelest: A Multi-Agent Aerospace Mission Planning System with LLM-Orchestrated Agents and Judge-Based Iterative Scoring

Mr. E. Subramanian[1], Mrs. M. Sabeetha[2], Abiraminayagi S[3], Akilnila B[3], Akshara I[3], Brinda P[3]

[1] (Assistant Professor, Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, Coimbatore, India)
[2] (Assistant professor, Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, Coimbatore, India)
[3] (Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, Coimbatore, India)
[a] Department of Computer Science and Engineering, India
[*] Corresponding Author Email: palanimuthubrinda@gmail.coms

ABSTRACT :

Aerospace mission planning requires reconciling trajectory design, payload selection, resource budgeting, and risk assessment, tasks traditionally handled by separate specialists and disconnected tools. This paper presents Etocelest, a multi-agent aerospace mission planning system in which specialised large-language-model (LLM) agents collaboratively draft, critique, and refine a mission plan. The platform incorporates: (1) a set of role-specialised planning agents coordinated through a LangGraph-style state-graph orchestration model with agent-to-agent (A2A) communication; (2) a Judge Agent that scores each candidate plan against a multi-criteria rubric and drives an iterative refinement loop until the plan converges to a score of 90 or above; (3) an asynchronous FastAPI service layer with MongoDB persistence through the Motor driver; and (4) a responsive React/Vite mission console for submitting mission requirements and inspecting agent reasoning and scores. Inference is served through the Groq API for low-latency generation. The working prototype demonstrates a Judge-driven scoring loop that converges to scores above 90. Quantitative benchmarks are reported in Section 6.

Keywords: Multi-Agent Systems, Aerospace Mission Planning, LLM Agents, LangGraph, Judge Agent, FastAPI, Agent-to-Agent Communication.

1. Introduction

Space and aerial missions involve tightly coupled decisions: the choice of orbit influences payload mass limits, which influence cost, which in turn influences risk tolerance. In practice, these decisions are made by different subject-matter experts working in separate documents and spreadsheets, so inconsistencies surface late and are expensive to fix. Recent advances in LLM-based agents make it feasible to model each specialist as an autonomous agent and to let them negotiate a consistent plan.

1.1 Background and Problem Formulation

Conventional mission planning workflows suffer from the following limitations: (1) Fragmented Expertise: trajectory, payload, resource, and safety analyses are performed in isolation; (2) Unquantified Plan Quality: plans are reviewed qualitatively, giving no repeatable measure of whether a plan is good enough; (3) Slow Iteration: each revision cycle requires manual coordination between teams; and (4) Opaque Reasoning: the rationale behind a decision is rarely captured in a form that can be audited.

1.2 System Contributions

The principal contributions of this work are summarized as follows:

- **Multi-Agent Planning Architecture:** specialised agents for mission planning, trajectory, payload, resources, and risk, coordinated through graph-based orchestration.
- **Judge Agent Scoring Loop:** an objective evaluator that scores candidate plans and feeds structured critique back to the planning agents until a score threshold is met.
- **Agent-to-Agent Communication:** a shared mission state and message protocol that lets agents query and revise one another's outputs.
- **Asynchronous Full-Stack Implementation:** a FastAPI and Motor backend with a React/Vite frontend, deployed as a web application.
- **Transparent Reasoning Trail:** every agent output, score, and critique is persisted for audit and replay.

1.3 Paper Organization

Section 2 surveys related work. Section 3 describes the architecture, workflow, data model, and scoring formulation. Section 4 details the functional modules. Section 5 covers deployment and reliability concerns. Section 6 presents results and discussion. Section 7 concludes and outlines future work.

1.4 Research Objectives

The work is guided by the following objectives: (1) to design a modular set of agents whose responsibilities mirror the disciplines involved in a real mission planning team; (2) to define a quantitative, repeatable criterion for plan quality so that refinement can be automated; (3) to expose the whole process through a web interface so that non-expert users can submit requirements and understand the resulting plan; and (4) to keep the system lightweight enough to be deployed on commodity cloud infrastructure and developed within the constraints of an academic project.

2. Related Work and Literature Survey

Classical mission design relies on physics-based tools and expert judgement, with optimisation methods such as genetic algorithms applied to trajectory and constellation design. More recently, LLM-based agent frameworks have shown that role-specialised agents can decompose complex tasks, and that a separate critic or judge model can improve output quality through iterative feedback (the "LLM-as-a-judge" pattern).

2.1 Existing Planning and Agent Frameworks

General-purpose agent frameworks provide tool use and memory but typically lack domain-specific evaluation criteria. Graph-based orchestration libraries such as LangGraph model an agent workflow as a state machine with explicit nodes, edges, and shared state, which suits mission planning because the control flow (plan, critique, revise) is cyclic and conditional. Domain mission-analysis software, in contrast, offers accurate physics but no natural-language reasoning or automated critique.

2.2 LLM-Based Multi-Agent Systems

The transformer architecture [3] enabled large language models capable of following natural-language instructions. Building on this, reasoning-and-acting prompting [4] showed that interleaving thought and action steps allows a model to use tools and revise its behaviour, while conversational multi-agent frameworks [5] demonstrated that several agents with distinct roles can solve tasks that a single prompt handles poorly. Simulation studies of generative agents [6] further showed that persistent memory and structured interaction produce coherent collective behaviour. These results motivate the role-specialised design adopted in Etocelest.

2.3 LLM-as-a-Judge and Iterative Refinement

Using a strong model to grade the outputs of another model has been shown to agree closely with human preferences on open-ended tasks [2]. Iterative refinement loops extend this idea by feeding the grader's critique back to the generator. In a planning setting this is attractive because the grader can enforce explicit criteria (completeness, consistency, feasibility) that a single generation pass does not reliably satisfy. A known risk is that the judge may reward fluent but incorrect plans; Etocelest therefore uses a structured rubric rather than a single holistic grade.
Table 1 summarises how the main strands of prior work relate to the present system.

Table 1 - Summary of Related Approaches.

Approach	Strength	Limitation for Mission Planning
Physics-based mission analysis [7]	Numerically accurate	No natural-language reasoning or automated critique
Reasoning-and-acting agents [4]	Tool use and self-correction	Single agent, no domain roles
Conversational multi-agent frameworks [5]	Role-based collaboration	Generic, no domain rubric
LLM-as-a-judge [2]	Scalable quality grading	Needs task-specific criteria

2.4 Research Gap

Existing mission-analysis tools focus on numerical accuracy, whereas existing agent frameworks focus on generic tasks. To the best of the authors' knowledge, there is limited work that combines domain-specialised planning agents with a rubric-based judge in a deployable web application for early-stage mission planning. Etocelest addresses this gap at the conceptual-design level, where rapid, explainable iteration is more valuable than high-fidelity simulation.

2.5 Comparative Feature Analysis

Table 2 compares Etocelest with representative approaches across key dimensions.

Table 2 - Comparative Feature Matrix of Mission Planning Approaches.

Dimension	Etocelest (Proposed)	Physics Tools	Single LLM	Manual
Specialised Agents	Full (multi-agent)	None	None	Human experts
Quantified Plan Score	Judge rubric	Domain metrics only	None	Ad-hoc
Iterative Refinement	Automated loop	Manual re-run	Single pass	Manual meetings
Reasoning Audit Trail	Persisted per agent	Limited	Chat log only	Scattered documents
Web Deployment	Yes	Desktop software	Yes	N/A

3. System Architecture and Design Methodology

Etocelest follows a decoupled three-tier design: a Presentation Layer, an Agent and Application Logic Layer, and a Persistence Layer.

3.1 Architectural Decomposition

The Client Tier is a React application built with Vite, providing a mission-requirements form, a live view of agent activity, and a score dashboard. Communication with the backend uses asynchronous REST calls.

The Application Tier is built on FastAPI. Route handlers accept mission requirements, launch the agent workflow, and expose results. The agent workflow is modelled as a state graph: each agent is a node that reads from and writes to a shared mission state, and conditional edges route the state to the Judge Agent and back to the planners based on the returned score. LLM inference is performed through the Groq API.

The Persistence Tier uses MongoDB accessed through the asynchronous Motor driver, storing missions, agent outputs, judge evaluations, and iteration history.

Table 3 - Technology Stack of Etocelest.

Layer	Technology	Purpose
Frontend	React, Vite	Mission console and score dashboard
API	FastAPI (Python)	Asynchronous REST endpoints
Orchestration	LangGraph-style state graph	Agent control flow and shared state
Inference	Groq API (hosted LLM)	Low-latency generation for agents and judge
Database	MongoDB with Motor	Missions, outputs, evaluations, history
Hosting	Vercel, GitHub	Deployment and source control

3.2 Agent Roles

Table 4 - Agent Roles in Etocelest.

Agent	Responsibility
Mission Planner	Interprets requirements and produces the overall plan outline
Trajectory Agent	Proposes orbit or flight path and timeline
Payload Agent	Selects instruments and checks mass and power constraints
Resource Agent	Estimates budget, fuel, and schedule
Risk Agent	Identifies hazards and mitigation measures
Judge Agent	Scores the assembled plan and returns structured critique

3.3 Mathematical Formulation of the Judge Score

The Judge Agent evaluates a candidate plan P across K criteria. The Plan Score S(P) in [0, 100] is defined in Equation (1):

$$S(P) = \Sigma\, w_k \cdot s_k(P),\ k = 1..K \quad (1)$$

where s_k in [0, 100] is the score assigned to criterion k (for example feasibility, completeness, consistency, and safety), w_k is its weight, and sum(w) = 1.0. The refinement loop terminates when the score reaches a threshold τ or a maximum iteration count N is reached, as in Equation (2):

$$\text{stop at iteration } t \text{ if } S(P_t) \geq \tau \text{ or } t = N \quad (2)$$

In the prototype, τ is set to 90, and the loop converges to scores of 90 or above.

3.4 Agent Workflow and State Graph

The orchestration layer treats the mission plan as a shared state object that is progressively enriched. The workflow proceeds as follows: (1) the Mission Planner parses the user requirements into objectives, constraints, and a plan outline; (2) the Trajectory, Payload, and Resource agents each read the outline and contribute their sections; (3) the Risk Agent reviews the assembled sections and records hazards and mitigations; (4) the Judge Agent scores the complete plan and writes a structured critique; and (5) a conditional edge either terminates the run, if the threshold is met, or routes the state back to the planners together with the critique.

Because every node reads and writes the same state, any agent can inspect the output of another before acting. This is the basis of the agent-to-agent communication described in Section 4.3, and it also gives the graph a natural checkpoint after each node.

3.5 Data Model

Persistence is organised into four collections, summarised in Table 5. Storing every iteration, rather than only the final plan, allows the reasoning trail to be replayed and compared.

Table 5 - MongoDB Collections.

Collection	Contents
missions	Requirements, constraints, status, final plan
agent_outputs	Per-agent text, prompt, latency, iteration index
evaluations	Judge scores per criterion, critique, total score
iterations	Plan snapshots and score history per run

3.6 Prompt Design

Each agent receives a role description, the relevant slice of the shared state, and an output format specification. Constraining the output format keeps responses compact, which reduces token consumption and makes the results easy to parse into state fields. The Judge prompt additionally includes the rubric definition and instructs the model to justify each criterion score before giving the number, which improves consistency across runs.

3.7 Design Rationale

Several design decisions follow from the intended use of the system. An asynchronous backend was chosen because each mission run involves many sequential LLM calls whose latency dominates the total time; asynchronous handlers allow other users to be served while a run is in progress. A document database was chosen because agent outputs are semi-structured text whose shape evolves as prompts are refined, and a fixed relational schema would have required frequent migrations. Finally, a hosted inference API was preferred over local model serving so that the project could be developed and demonstrated without dedicated GPU hardware.

4. Functional Implementation and Core Modules

4.1 Mission Input and Orchestration

Users describe the mission objective, constraints, and priorities in the mission console. The backend stores the request, initialises the shared state, and starts the agent graph. Each node call is logged with its prompt, output, and latency.

4.2 Judge-Driven Refinement Loop

After the specialist agents contribute, the Judge Agent returns a numeric score together with strengths, weaknesses, and concrete revision instructions. If the score is below the threshold, the state is routed back to the relevant agents with the critique attached, and the plan is regenerated. This continues until convergence.

4.3 Agent-to-Agent Communication

Agents exchange information through the shared state and explicit messages, for example the Payload Agent asking the Resource Agent to confirm a mass budget. This reduces contradictions between sections of the final plan.

4.4 Mission Console

The frontend presents the final plan, per-iteration score progression, and the reasoning trail of each agent, so that reviewers can see how and why the plan changed.

4.5 REST API Design

The backend exposes a small set of resource-oriented endpoints, summarised in Table 6. Long-running generation is started by a POST request and its progress is retrieved by polling the mission resource, so that the client is never blocked on a single long HTTP call.

Table 6 - Representative REST Endpoints.

Endpoint	Method	Function
/missions	POST	Create a mission and start the agent run
/missions/{id}	GET	Retrieve status, plan, and score
/missions/{id}/iterations	GET	List per-iteration plans and scores
/missions/{id}/agents	GET	Retrieve agent outputs and reasoning trail
/missions	GET	List previous missions

4.6 Judge Evaluation Rubric

The Judge Agent grades each plan against the criteria in Table 7. The criteria were chosen to cover both the internal quality of the document and its engineering plausibility.

Table 7 - Judge Rubric Criteria.

Criterion	Description
Feasibility	Whether the proposed trajectory, payload, and schedule are technically plausible
Completeness	Whether all required plan sections are present and sufficiently detailed
Consistency	Whether numbers and assumptions agree across agent sections
Safety and Risk	Whether hazards are identified with credible mitigation measures
Resource Realism	Whether budget, mass, and power estimates are reasonable

4.7 Illustrative Usage Scenario

Consider a user who submits the brief "plan a low-Earth-orbit Earth-observation mission with an optical camera payload and a limited budget". The Mission Planner extracts the objective (imaging), the orbit class, the payload type, and the cost constraint. The Trajectory Agent proposes an orbit and revisit strategy, the Payload Agent selects an instrument suite consistent with the mass and power limits, and the Resource Agent drafts a budget and schedule. The Risk Agent adds hazards such as launch delay, thermal stress, and communication loss.

The Judge Agent then scores the assembled plan. A typical first critique points out, for example, that the budget does not account for ground-station costs or that the payload power draw is not reconciled with the stated power budget. These critiques are returned to the relevant agents, whose revised sections are re-scored in the next iteration. The user can open the iteration history in the console and see which critique triggered which change.

5. Deployment and Reliability

5.1 Service Layer

The backend is fully asynchronous so that long-running LLM calls do not block other requests. Mission state is persisted after every iteration, allowing recovery from interrupted runs.

5.2 Handling LLM Rate and Token Limits

During development, token-limit constraints on the hosted inference API affected long multi-agent prompts. The system mitigates this by keeping the shared state compact, passing each agent only the sections it needs, and selecting a model suited to the workload.

5.3 Deployment

The frontend is deployed on Vercel, and the source is hosted on GitHub.

5.4 Security and Configuration

API keys for the inference provider are stored as server-side environment variables and are never exposed to the browser. Cross-Origin Resource Sharing is restricted to the deployed client origin, and request bodies are validated against typed schemas before they reach the agent layer, which rejects malformed input early.

5.5 Fault Tolerance

Calls to the inference API can fail because of rate limits or transient network errors. Each agent call is wrapped with a bounded retry and back-off policy, and the loop enforces a maximum iteration count so that a run cannot continue indefinitely if the score plateaus below the threshold. In such cases the best-scoring plan found so far is returned together with its critique.

5.6 Scalability Considerations

Since each mission run is independent, the service can be scaled horizontally by adding application instances behind a load balancer, with MongoDB as the shared store. The principal bottleneck is the rate limit of the inference provider rather than the application server, so request queuing and per-user concurrency limits are the first controls to introduce under heavier load.

6. Results, Performance Evaluation, and Discussion

The prototype was evaluated on a set of sample aerospace missions. The principal observed outcome is that the Judge Agent loop drives candidate plans to scores of 90 or above.

6.1 Experimental Setup

The prototype was run against a small set of sample mission briefs of varying complexity, such as an Earth-observation satellite, a lunar payload delivery, and a high-altitude research balloon. For each brief, the system was run with the score threshold fixed at 90 and a bounded iteration limit. The recorded quantities were the Judge score after every iteration, the number of iterations to convergence, wall-clock latency, and token usage per run.

6.2 Judge Score Convergence

Table 8 - Judge Score Progression per Scenario.

Mission Scenario	Initial Score	Final Score	Iterations
Scenario 1	[value]	[value]	[value]
Scenario 2	[value]	[value]	[value]
Scenario 3	[value]	[value]	[value]

6.3 Latency and Cost

Table 9 - End-to-End Latency and Token Usage.

Stage	Mean Latency (s)	Tokens per Run
Planning agents	[value]	[value]
Judge evaluation	[value]	[value]
End-to-end	[value]	[value]

6.4 Discussion

The results indicate that a dedicated judge improves plan quality through repeated critique, and that persisting reasoning trails makes outputs auditable. A limitation is that LLM-generated plans are not a substitute for physics-based verification; the Judge scores plausibility and consistency rather than numerically validating orbital mechanics.

6.5 Threats to Validity

Three limitations should be noted. First, the Judge and the planning agents may share the same underlying model family, so the Judge may be biased towards the style of the plans it evaluates. Second, the evaluation set is small and was selected by the authors, so the results should be read as a demonstration of feasibility rather than a statistical benchmark. Third, LLM outputs are non-deterministic, so scores and iteration counts vary between runs of the same brief.

6.6 Lessons Learned

Development surfaced several practical lessons. Token limits on the inference API were the main engineering obstacle and shaped the design of the shared state. Agents produced better results when their prompts were short and their output formats strict. Persisting every intermediate output greatly simplified debugging, because failures could be traced to a specific node and iteration rather than to the run as a whole.

6.7 Responsible Use

Etocelest is intended as a conceptual-design aid and learning tool. Its outputs are generated by language models and must not be used as the sole basis for real mission or safety decisions without independent engineering verification. The interface therefore labels generated plans as advisory, and the persisted reasoning trail is intended to support, not replace, expert review.

7. Conclusion and Future Scope

This paper presented Etocelest, a multi-agent aerospace mission planning system that combines role-specialised LLM agents, graph-based orchestration, and a Judge Agent scoring loop. The system produces consistent, auditable mission plans that converge to high judge scores.

Future research and engineering avenues will explore: (1) integrating physics-based simulators as verification tools callable by agents; (2) fine-tuning or calibrating the Judge against expert-labelled plans; and (3) retrieval of historical mission data to ground agent decisions. Additional directions include:

(4) using a different model family for the Judge to reduce self-preference bias; (5) adding human-in-the-loop review steps in which an engineer can accept or override agent decisions; and (6) exporting the final plan as a formatted mission report for use by other teams.

Acknowledgements

The authors express sincere gratitude to the faculty advisors and the Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, for their guidance and support.

REFERENCES :

1. Wooldridge, M. (2009). An Introduction to MultiAgent Systems (2nd ed.). Wiley.
2. Zheng, L., et al. (2023). Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. NeurIPS Datasets and Benchmarks Track.
3. Vaswani, A., et al. (2017). Attention is all you need. Advances in Neural Information Processing Systems, 30.
4. Yao, S., et al. (2023). ReAct: Synergizing reasoning and acting in language models. International Conference on Learning Representations.
5. Wu, Q., et al. (2023). AutoGen: Enabling next-gen LLM applications via multi-agent conversation. arXiv:2308.08155.
6. Park, J. S., et al. (2023). Generative agents: Interactive simulacra of human behavior. ACM UIST.
7. Wertz, J. R., & Larson, W. J. (1999). Space Mission Analysis and Design (3rd ed.). Microcosm Press.
8. LangChain, Inc. (n.d.). LangGraph documentation. https://langchain-ai.github.io/langgraph/
9. Ramírez, S. (n.d.). FastAPI documentation. https://fastapi.tiangolo.com/
10. MongoDB, Inc. (n.d.). Motor: asynchronous Python driver for MongoDB. https://motor.readthedocs.io/
11. Groq, Inc. (n.d.). Groq API documentation. https://console.groq.com/docs
12. React. (n.d.). React documentation. https://react.dev/