IFN11 Final Project Report

Coppercore AI

Student and Group Details			
Group number	22		
Student name	Maryam Jamshidi	**Student number**	N11672510
Student name	Sicheng Chen	**Student number**	N12205699
Student name	John Jude Kuzhivelil	**Student number**	N11870192
Student name	Dili Mariya Maichal	**Student number**	N11778300
Student name	Carla Chen	**Student number**	N11564628

Project Supervisor	Dr. Venkat Venkatachalam
Tutorial Time	Wednesday 9 AM – 12 PM
Project Name	Coppercore AI
Industry Partner	Orefox AI Limited

Project Description
The final artefact is a web-based application that detects and classifies porphyry copper anomalies from geophysical data using machine learning, and visualises the results on an interactive map.

Criteria	High Distinction	Distinction	Credit	Pass	Fail
Project Outcome and success • *Value* • *Useful* • *Complete* **25 marks**	Demonstrates a sophisticated understanding of the project context including the nature of the business or organization and the sector within which it operates The project goals are listed and explained so that their impact is clear. There is a clear and professional assessment of the extent to which the goals were achieved, and explanation provided as to how the assessment was achieved including any metrics use for evaluation.	Demonstrates a high understanding of the project context including the nature of the business or organization and the sector within which it operates The project goals are listed and explained clearly but lack sophisticated engagement and depth of understanding. An evaluation of effectiveness is clearly presented but lacking depth of engagement and inclusion of logical rationale or metric.	Demonstrates a good understanding of the project context including the nature of the business or organization and the sector within which it operates Project goals are listed and an explanation is provided but is missing clear articulation of their impact and value. A general assessment of success is included with limited depth and no inclusion of rationale or evaluative metric.	Demonstrates an understanding of some significant aspects of the project including the nature of the business or industry. The project goals are presented but not explained with depth or detail. Most information is pertinent to the project. A simplistic assessment of success is included with limited, or no rationale or evaluative metric included.	Does not display an understanding of the project correctly, significant aspects missing. Reasoning is deficient. Information is not relevant or flawed. Project goals are not articulated, incorrect or unprofessional in engagement
Project Progress & Reflections • Critical Analysis • Problem Solving • Efficiencies • Effectiveness **25 marks**	• The project increments were executed completely as outlined in the Assessment 2 Project Plan • All increment level evidence on the project progress is included using accepted tools (Trello images, burndown charts, Gantt Charts and retrospective records) • Revisions to the Assessment 2 Project Plan are discussed and justified professionally. • Relevant experiences related to issues your group faced are critically analysed, presented with insightful resolutions	• The project increments were executed mostly as outlined in the Assessment 2 Project Plan • Most increment level evidence on the project progress is included using accepted tools (Trello images, burndown charts, Gantt Charts and retrospective records) • Revisions to the Assessment 2 Project Plan are discussed in detail and justified • Relevant experiences related to group level issues are properly analysed, presented with original resolutions	• The project increments were executed partly as outlined in the Assessment 2 Project Plan • Some of the increment level evidence on the project progress is included using accepted tools (Trello images, burndown charts, Gantt Charts and retrospective records) • Revisions to the Assessment 2 Project Plan are described with limited justifications • Relevant experiences related to group level issues are analysed, presented with basic resolutions	• Most of the project increments did not deliver meaningful outcomes • The increment level evidence on the project progress is not relevant or not included from the tools (Trello images, burndown charts, Gantt Charts and retrospective records, "Done" lists) • Limited Revisions to the Assessment 2 Project Plan are mentioned • Relevant experiences related to group level issues are not discussed	• No useful outcomes were delivered in all increments • The increment level evidence on the project progress is not included from accepted tools (Trello images, burndown charts, Gantt Charts and retrospective records) • Revisions to the Assessment 2 Project Plan are not covered at all • Relevant experiences related to group level issues are not discussed (no group level reflections)

IT artefacts Delivered to the customer • *Quality* • *Professional Standards* **40 marks**	The artefacts and its subcomponents submission fully match with the scope agreed with the client Those are delivered to with high quality standards and well above normal expectations of the industry partner or tutor The IT artefacts are indistinguishable from one produced in a similar time frame by an experienced professional team.	The artefact submission encompasses much of the scope agreed with the client or tutor but falls short in several important areas, with key components either not attempted or completed, or completed well below a professional standard. Overall, the artefact reflects work of a good standard for the work successfully delivered, while remaining below the level as those in the higher-grade band.	The artefact submission broadly reflects the scope agreed with the client, but some aspects of the agreed deliverable may not have been completed or may not have been completed to a professional standard. Overall, the artefact reflects work of a near professional standard for the work successfully delivered, without perhaps reaching the same level as those in the higher-grade band.	The artefact submission encompasses some of the scope agreed with the client but falls well short of an acceptable deliverable for the project. Key components are either not attempted or completed or completed poorly. Overall, the artefact reflects work of a relatively weak standard for the work successfully delivered,	The artefact submission encompasses little or none of the scope agreed with the client or tutor None of the agreed components of the artefact have been delivered, and those mentioned are of poor quality. Key components are either not attempted or
Communications • ***Clarity*** • ***Conciseness*** • ***Correctness*** ***10 marks***	The report is consistently professional in tone and structure and addresses each of the listed requirements in great detail No errors of grammar or structure. Report is organized to aid understanding, and this is assisted by the layout and formatting. The standard of writing exceeds well above expectations	The report is generally professional in tone and structure and addresses each of the listed requirements in detail Very limited errors in grammar or structure. The report is well organized, and the layout and formatting are well chosen. The standard of writing is above expectations	The report is professional in tone and structure but lacks some detail in a small number of the listed requirements. There may be frequent and occasional errors of grammar or structure. Organization and layout remain good, The standard of writing meets the expectations of this level.	The report does not meet expectations, and the coverage is deficient in several the listed requirements. Grammar and structure are variable but are usually ok. The organization is deficient, but some effort has been made to structure and format the document. The standard of writing may lie below the expectations at this level	The report doesn't meet the requirements set out in the brief. Sections missing or poorly covered. Meaning unclear as grammar and/or spelling contain frequent errors Disorganised or incoherent writing Structure either absent or incoherent and the standard of writing may be well below the expectations at this level

Executive Summary

The CopperCore AI project was developed in collaboration with Orefox, a geoscience company focused on accelerating mineral exploration using artificial intelligence. Orefox's core problem was the time-consuming and expertise-heavy process of manually interpreting large magnetic datasets to locate potential porphyry copper deposits. They required a system capable of automatically detecting and classifying anomalous patterns in geophysical data, while remaining accessible to geologists over the age of 50 with limited digital literacy.

To address this, the team delivered an end-to-end AI-assisted platform that supports anomaly detection, classification, and result visualization. The frontend was designed with a simple, intuitive layout and minimal interaction complexity to meet the usability expectations of older professionals. Users can upload .tif files, view processed coordinates on an interactive map, and explore detected anomalies using integrated filtering and search tools. A built-in AI chatbot allows users to ask geological questions and retain chat history for ongoing reference.

On the machine learning side, an unsupervised autoencoder model was used for anomaly detection, trained on Total Magnetic Intensity (TMI) data from the Queensland government. This model identifies local patches with high reconstruction error as anomalous. The output is passed to a lightweight Random Forest classification model that labels patches as potentially ore-related or not, based on intensity and location features. The backend system, built in Django and hosted on Microsoft Azure, manages file uploads, ML processing, classification, chat threads, and data storage using PostgreSQL and Azure Blob Storage.

The project faced several key challenges, including the unexpected unavailability of training data from the client, which led the team to pivot from a supervised learning approach to an unsupervised one. Additional delays were caused by version incompatibilities during model integration. Despite these setbacks, the team adapted quickly using agile methods and successfully demonstrated the working system to the client. Although the Azure services were recently deactivated due to the expiry of the free student subscription, both the tutor and client were informed, and the full functionality had already been reviewed.

Table of Contents

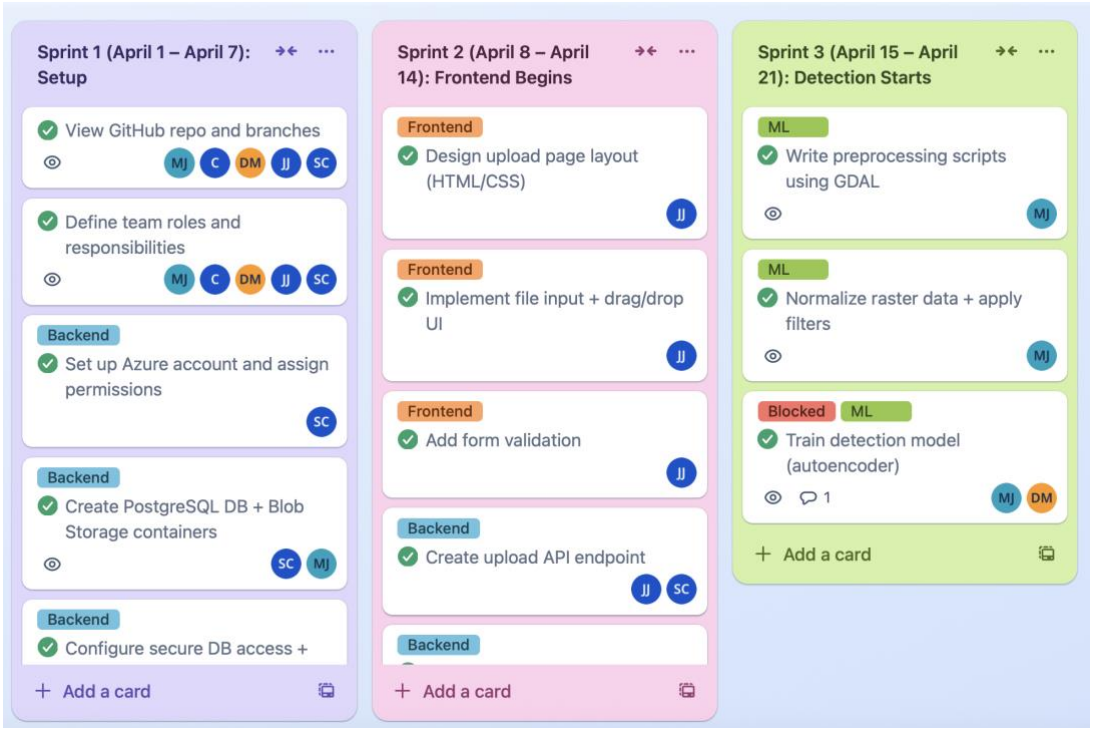
Analysis

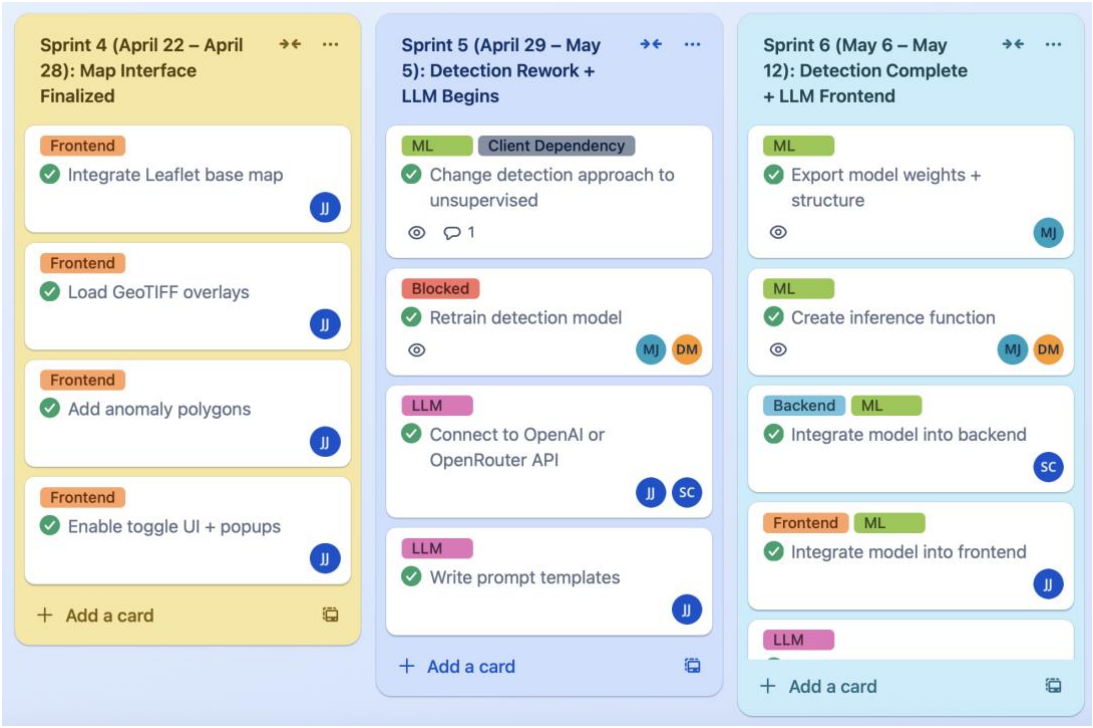
The CopperCore AI project aimed to deliver an intelligent, user-friendly system for identifying porphyry-style mineralization zones within large-scale geophysical datasets. Orefox operates in the mineral exploration sector and sought a solution to automate anomaly detection and classification to accelerate early-stage exploration. The client emphasized the need for a simple interface suitable for geologists over 50, many of whom had limited experience with modern digital tools. This project held real-world value and demanded robust analytical and collaborative problem-solving approaches to meet dynamic requirements.

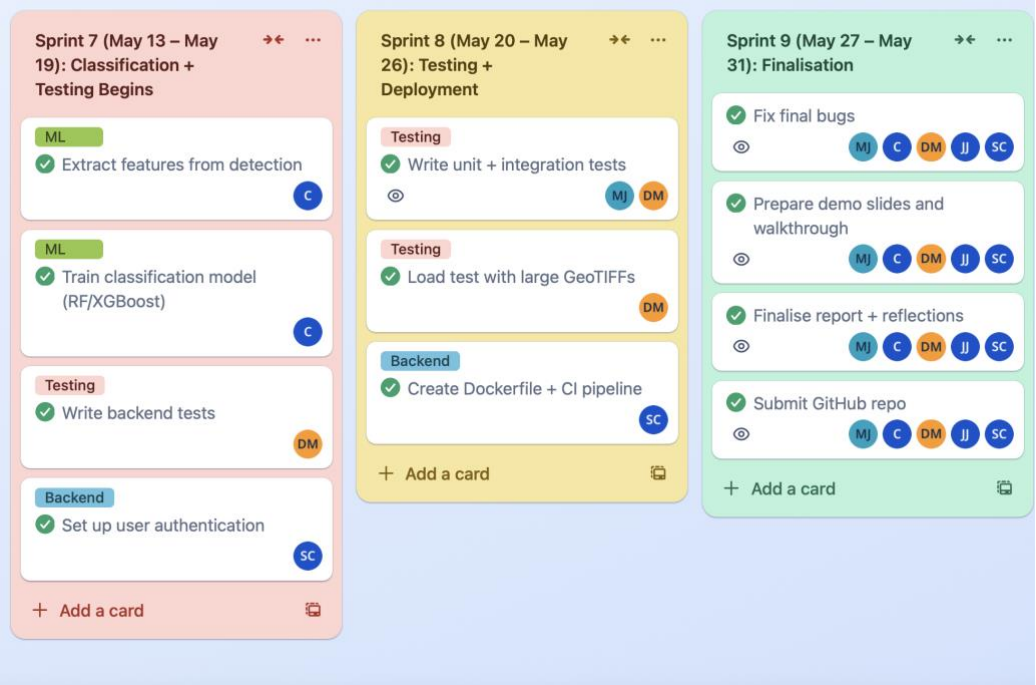
From the beginning, the project involved significant unknowns. A key challenge was that no one on the team had prior experience working with geophysical data. While the client gave access to a Queensland-wide magnetic dataset, it was unclear how to interpret the data or what preprocessing would be necessary. Through independent research and multiple client meetings, the team gradually understood the relevance of magnetic anomalies and identified Total Magnetic Intensity (TMI) as the most viable data layer for the system. This learning process was a critical step in forming our overall technical strategy.

A second major unknown was introduced when the client, after several weeks of development, informed us that they could not provide the CSV file containing labeled anomalies, which is a file that our supervised machine learning plan had relied on. This forced the team to pivot mid-project to an unsupervised learning model, disrupting our original plan and reordering team assignments. Everyone working on detection and classification had to rapidly acquire new machine learning knowledge to implement an alternative pipeline. This pivot required cross-functional collaboration and weekly meetings to reorganize priorities and reassign tasks.

Our agile methodology was tested during this shift. Originally, roles were clearly split, for example, Maryam was assigned to classification, Carla to frontend and detection, and Dili to model support. But as the crisis unfolded, these roles were adapted: Maryam shifted to assist with detection while Carla took over classification duties. Sicheng, originally assigned to backend development, also contributed to cloud environment setup and LLM work when deployment dependencies stalled progress. These decisions were made incrementally and logged using our Trello board, which was structured into 9 sprint lists (April 1 to May 31). Each card detailed tasks, assignments, and progress updates. The Trello board below illustrates how team adaptability and frequent reprioritization helped us stay aligned even as the plan evolved. It shows weekly sprint columns from April 1 to May 31, including task cards and assignments. Notable role changes and task swaps are visible from Sprint 4 onward, particularly around detection model rework and classification role reversal.


Sprint 1 (April 1 – April 7): Setup
View GitHub repo and branches
Define team roles and responsibilities
Backend
Set up Azure account and assign permissions
Backend
Create PostgreSQL DB + Blob Storage containers
Backend
Configure secure DB access +
Add a card
Sprint 2 (April 8 – April 14): Frontend Begins
Frontend
Design upload page layout (HTML/CSS)
Frontend
Implement file input + drag/drop UI
Frontend
Add form validation
Backend
Create upload API endpoint
Backend
Add a card
Sprint 3 (April 15 – April 21): Detection Starts
ML
Write preprocessing scripts using GDAL
ML
Normalize raster data + apply filters
Blocked
ML
Train detection model (autoencoder)
Add a card


Sprint 4 (April 22 – April 28): Map Interface Finalized
Frontend
Integrate Leaflet base map
Frontend
Load GeoTIFF overlays
Frontend
Add anomaly polygons
Frontend
Enable toggle UI + popups
Add a card
Sprint 5 (April 29 – May 5): Detection Rework + LLM Begins
ML
Client Dependency
Change detection approach to unsupervised
Blocked
Retrain detection model
LLM
Connect to OpenAI or OpenRouter API
LLM
Write prompt templates
Add a card
Sprint 6 (May 6 – May 12): Detection Complete + LLM Frontend
ML
Export model weights + structure
ML
Create inference function
Backend
ML
Integrate model into backend
Frontend
ML
Integrate model into frontend
LLM
Add a card



These adjustments were also captured in our burndown chart, which shows the difference between ideal and actual progress. From sprint 2 to 6, the actual story point line pivoted from the ideal due to frontend delays and mainly due to the loss of our planned label file which delayed our detection model by 2 weeks. A sharp drop occurs in Sprint 6 when the detection model was finalized and progress resumed quickly. By Sprint 9, the chart converges with the ideal line, indicating full completion.

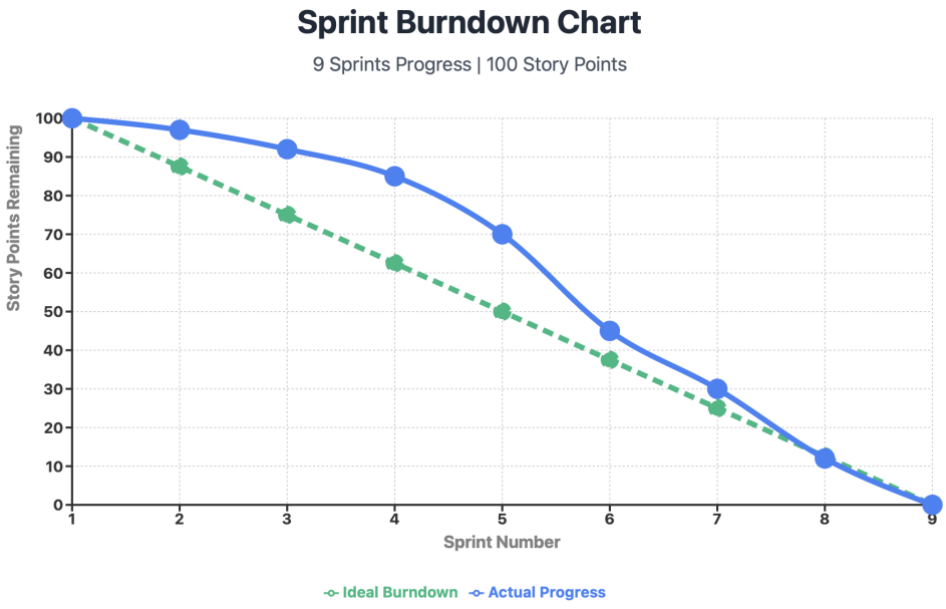


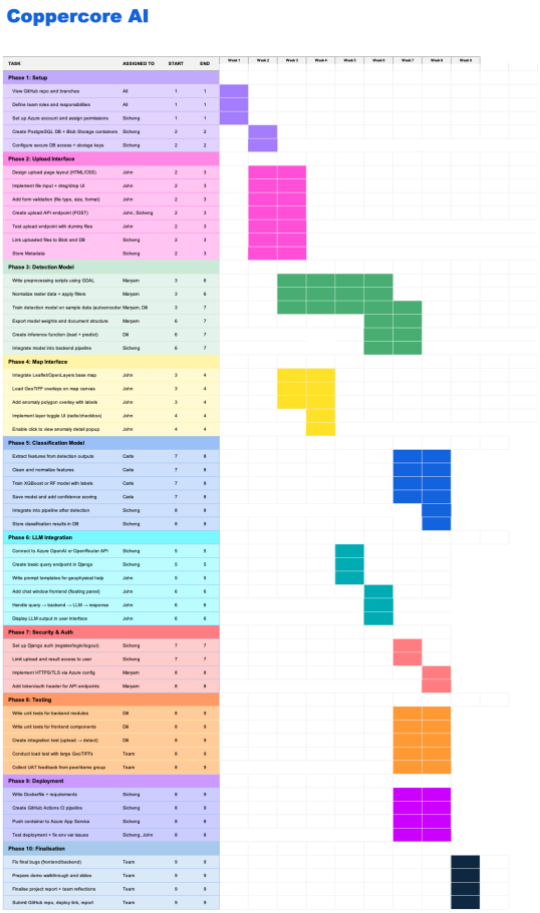
Figure 1: Sprint 1–3: Frontend setup delays. Sprint 4–5: ML model pivot with no CSV data. Sprint 6: Major breakthrough with detection model working. Sprint 7–9: Fast progress as frontend and classification came together.

The original Gantt chart assumed a linear timeline and fixed sprint-based goals with minimal interdependency. However, the actual project revealed several unanticipated technical and planning challenges that invalidated these assumptions. The missing CSV file derailed the original supervised pipeline entirely. Additionally, version incompatibilities in our machine learning environments (particularly between TensorFlow, Python, and deployment dependencies) created further delays and required rework.

In response, we modified our Gantt chart to reflect how timelines shifted based on real progress and inter-team dependency resolution. The classification model and frontend deployment were pushed to later sprints, while detection model training required more collaborative time investment. Team members swapped roles to support critical paths, and some tasks such as chatbot integration were deprioritized temporarily to ensure core functionality.

As the Gantt chart is quite large and detailed, the file has been uploaded to OneDrive to allow for easier viewing.

Please click here to view the chart.



We also kept weekly records, identifying blockers, solutions, and key learnings. These records are available in the appendix to view in full. Some recurring blockers included TensorFlow version issues in deployment environments, frontend-backend integration bugs, and missing documentation on GeoTIFF input handling. Most of these were resolved through collaborative debugging sessions and pair programming. These logs were crucial for refining our sprint plans and mitigating risk.

Overall, the analysis phase revealed critical lessons about dependency risk, agile adaptation, and technical resilience. The team's ability to pivot quickly, redistribute workload, and learn unfamiliar technical domains was essential to completing the project successfully. Despite the obstacles, we met all core deliverables and achieved our functional goals, preparing a robust foundation for implementation.

Design

The CopperCore AI platform was designed to address Orefox's need for a lightweight, easy-to-use, AI-assisted geoscientific tool to support mineral exploration for geologists with low technological literacy. This section describes the major components that form the overall solution architecture and how they interoperate to fulfill the project goals. The key components include the frontend interface, detection model, classification model, large language model (LLM) chatbot, and backend infrastructure. These are supported by cloud storage, APIs, and database integration, which form the underlying operational layer. A link to the prototype demonstration video can be found in the appendix, explaining all main components and their functionalities.

Frontend Interface

The frontend serves as the primary point of interaction for end-users, who are geologists tasked with uploading geospatial datasets and reviewing analytical outputs. Built with HTML/CSS and integrated with Django templates, it emphasizes a clean, minimal UI that abstracts technical complexity. Key features include:

- An **uploader** for .tif geophysical maps (e.g., Total Magnetic Intensity).
- A **map visualization panel** where outputs from the detection and classification models are overlaid.
- A **chatbot popup** enabling natural-language geological queries.

The interface was tested with older, non-technical users to ensure navigability. Through simple visual outputs, it enables users to upload datasets and review results without understanding machine learning intricacies.

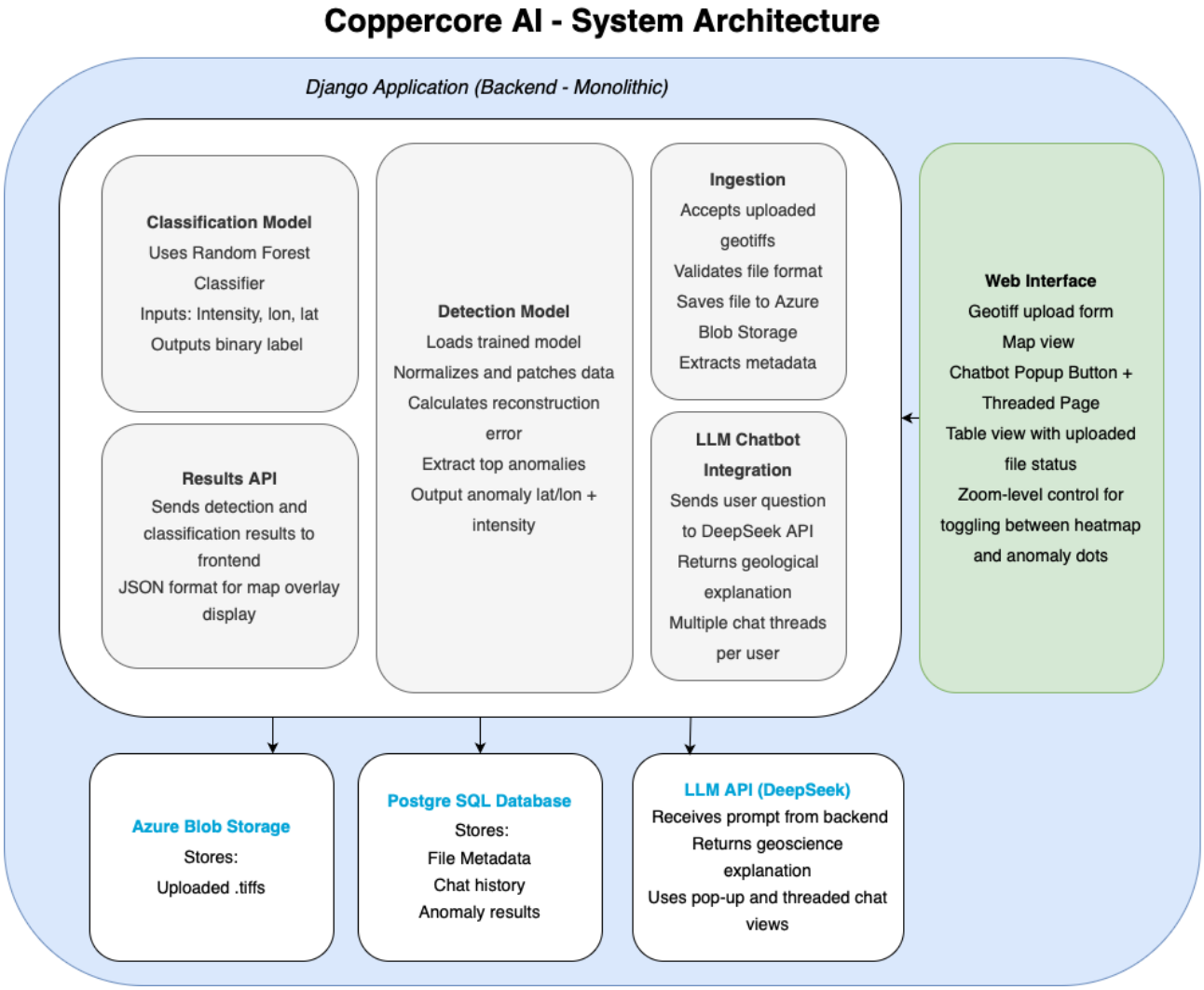


Figure 2:High-level architecture of CopperCore AI

Detection Model

The detection model is the system's first analytical layer, responsible for identifying potential geophysical anomalies in uploaded datasets. Users upload a GeoTIFF file, which is stored on Azure Blob Storage. Upon upload, the backend triggers a patching routine that divides the image into smaller tiles, normalizes pixel values, and feeds each patch into a pre-trained autoencoder neural network.

This unsupervised model reconstructs input patches and calculates reconstruction error. Patches with high reconstruction loss are flagged as anomalous. Coordinates (lat, lon) are extracted for the centroids of these patches, and each is assigned an intensity score based on anomaly magnitude.

Outputs are then passed to the classification model. All detections are saved in the database for retrieval.

Classification Model

The classification model forms the second layer, distinguishing between relevant and irrelevant anomalies. It takes as input the detection coordinates and intensity values, and assigns a binary classification (1 or 0) indicating whether a detected point is likely indicative of a mineral deposit.

This model was trained using labeled data extracted from prior datasets and uses a simple feedforward neural network. It complements the unsupervised nature of the detection stage with supervised learning, adding domain-driven refinement.

Classified points are returned to the frontend and visualized as heatmaps or pin overlays. This dual-model pipeline ensures that the system can both detect unusual signals and filter out noise based on known geological patterns.

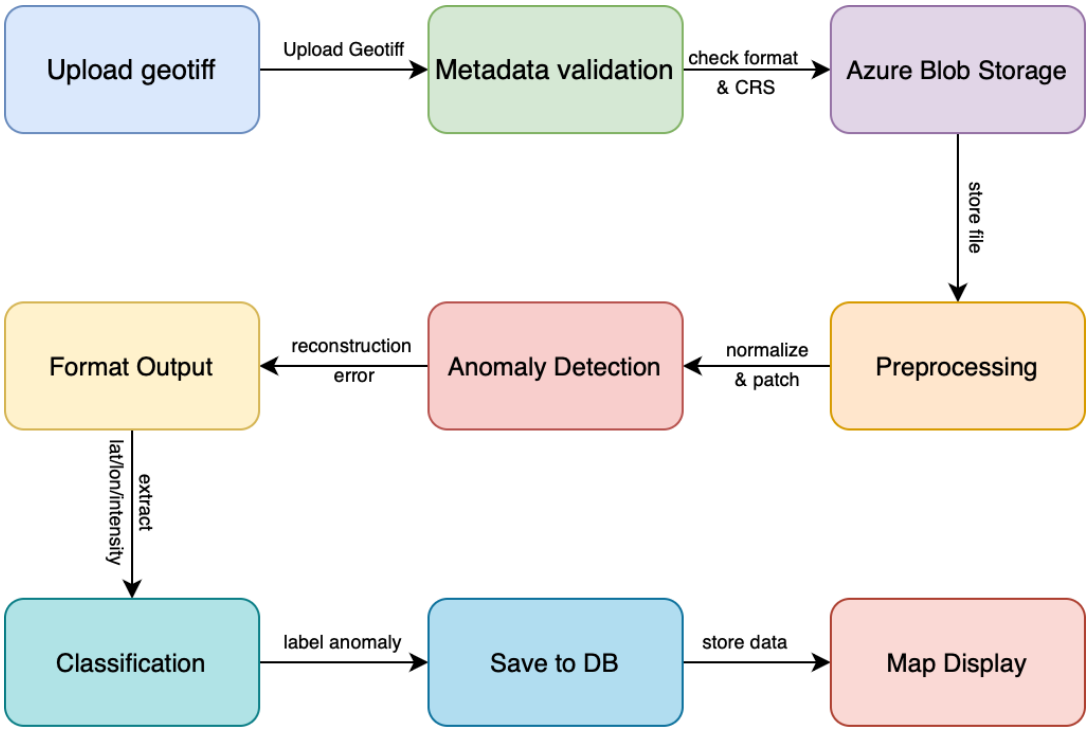
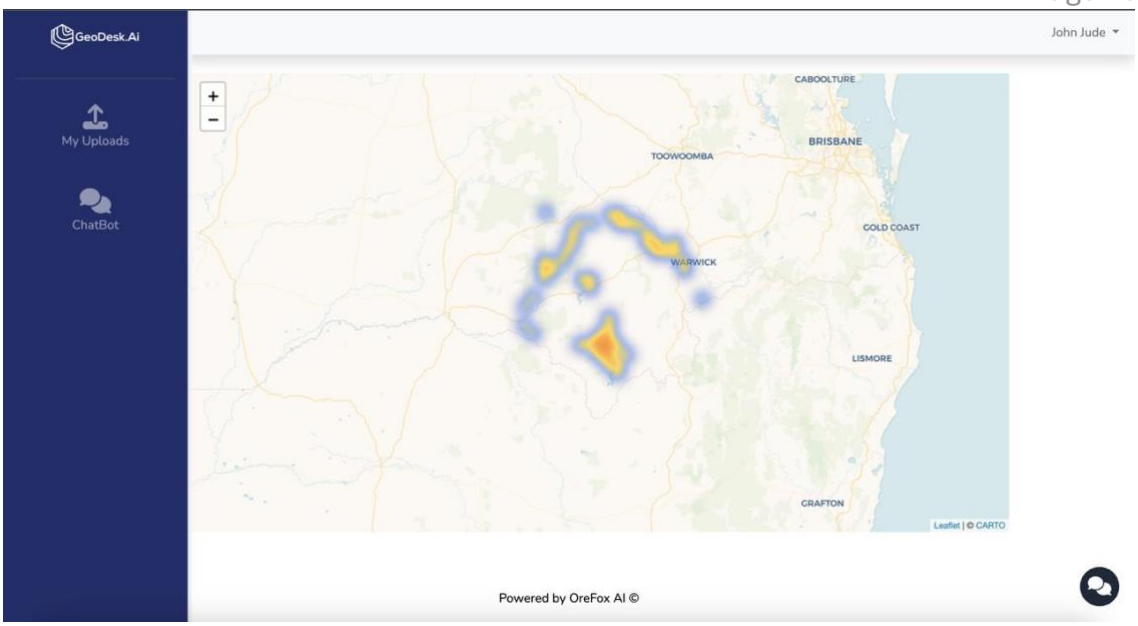
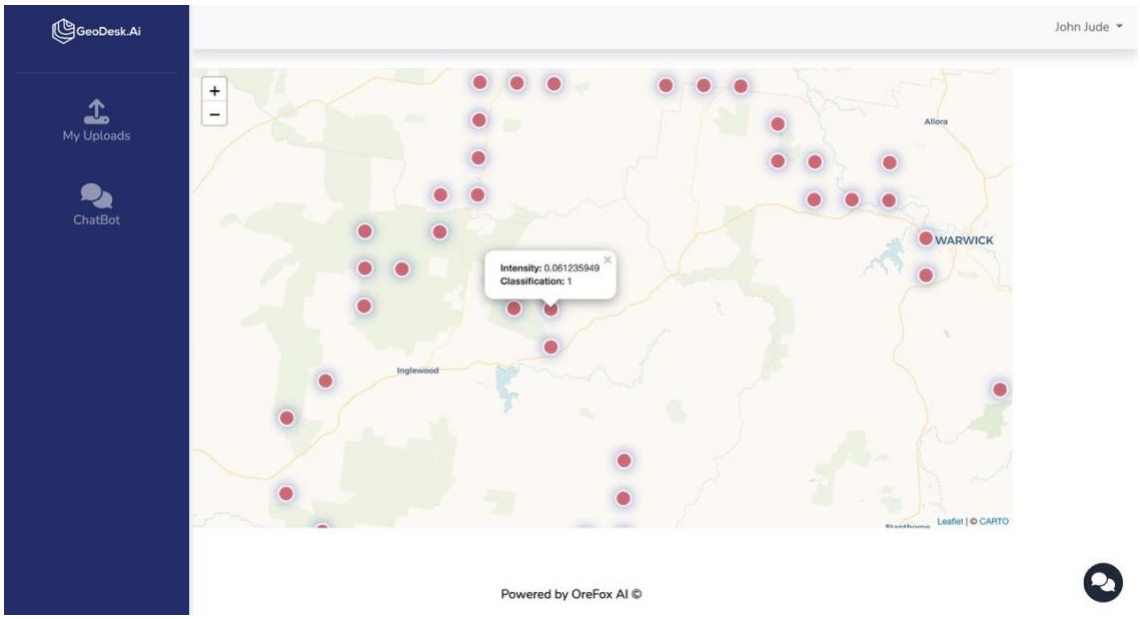


Figure 3: This diagram shows how data moves through CopperCore AI, from user upload through detection, classification, and frontend visualization.

The snapshots below show the output of the map after uploading is complete. Initially, the heat signatures for the anomaly coordinates are shown. If the user zooms in, the classification points are shown by red pulsing dots and upon clicking on it, a popup appears showing the coordinates and the intensity.



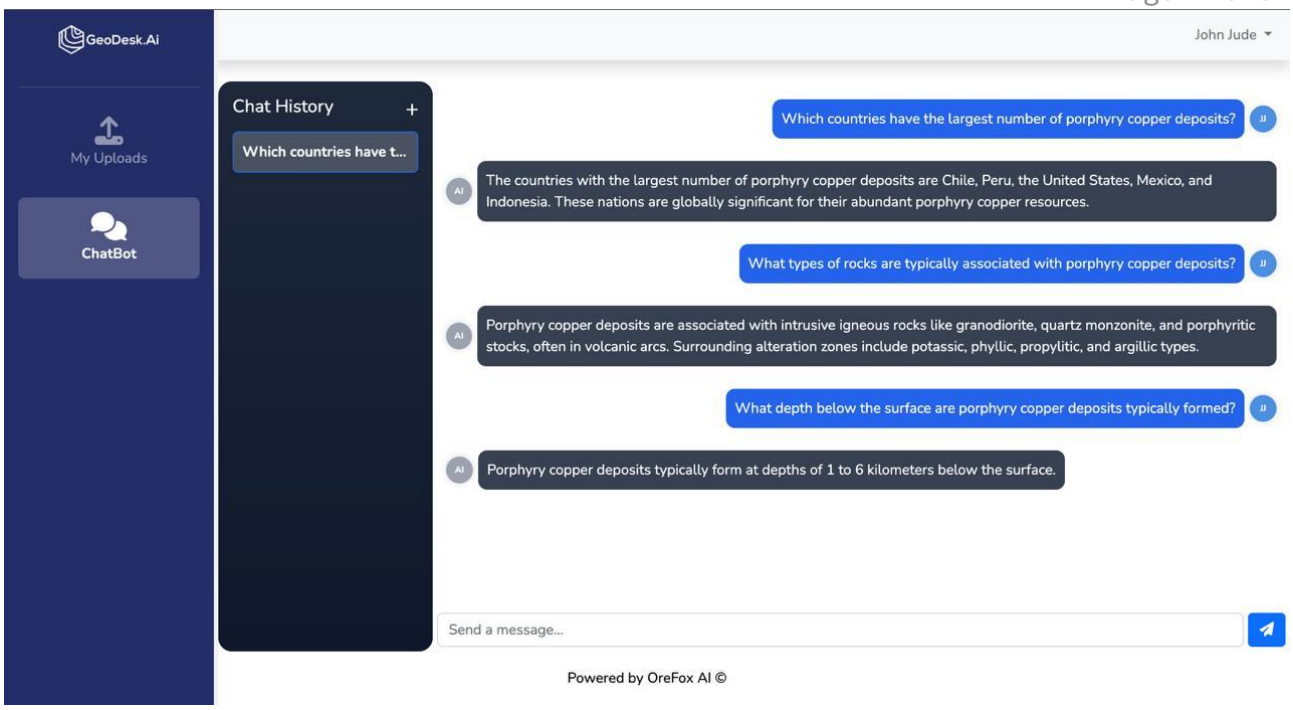


LLM Chatbot

To support non-technical users, an integrated large language model (LLM) chatbot powered by DeepSeek AI was added. Users can ask questions such as “What does this anomaly mean?” or “Is this pattern typical in Queensland?” The chatbot generates helpful responses with DeepSeek.

Chat history is stored in the PostgreSQL database, allowing users to revisit past conversations. While not a primary deliverable, the chatbot enhances usability and fosters exploratory engagement.

The snapshot below shows a sample interaction between the user and the chatbot.



Backend Infrastructure

The backend was developed using Django, with REST APIs that handle file ingestion, trigger model execution, and retrieve outputs. The backend coordinates the following:

- Accepting GeoTIFF uploads and linking to Azure Blob Storage.
- Running preprocessing and passing data to the detection model.
- Sending detected results to the classification model.
- Storing results in PostgreSQL.
- Serving endpoints for result queries, chatbot logs, and dataset history.

This component ensures coherence between the ML pipeline and user interaction.

The diagram below illustrates how data is stored in relation to each other, and the information stored. The detection results and chat history tables are connected with the upload history table through user ID/uploader ID.

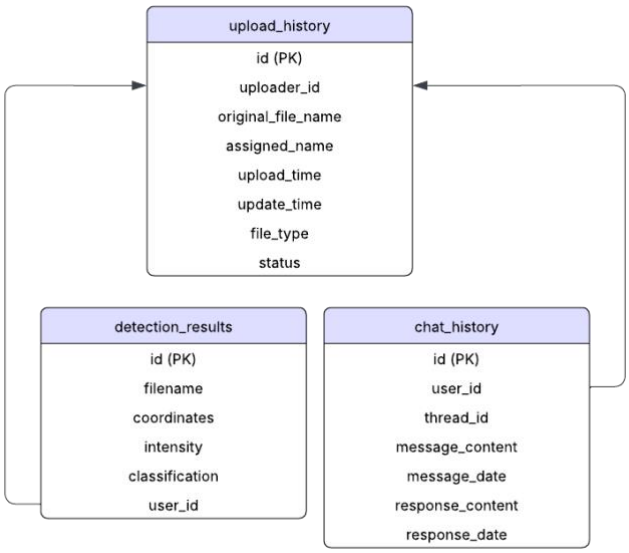


Figure 4: Diagram showing the structure of the chat_history and detection_results database schemas, which store uploaded files, ML outcomes, and chatbot conversations.

Integration and Flow

The system components are tightly integrated. Once a file is uploaded, the backend immediately initiates patch-based preprocessing and detection. Detected coordinates are stored and passed downstream to the classification model. Results are rendered on the frontend map view for intuitive interpretation. Meanwhile, users can query the chatbot, which accesses the stored results for contextual responses.

This seamless flow is what enables CopperCore AI to fulfill its project goals:

- **Simplified exploration** through an upload-to-visualization workflow.
- **Actionable results** via anomaly detection and classification.
- **User support** through the chatbot and clean UI.

API Endpoints

To support interaction between components, the following endpoints were developed:

Endpoint	Method	Purpose	Auth Required
/upload-dataset	POST	TIF file ingestion	Yes
/get_datasets	GET	Retrieve current user's uploaded files	Yes

/query	POST	Send geo-related queries to the LLM	Yes
/list_history_messages	GET	Retrieve user-specific chat history	Yes
/get_results_by_filename	GET	Fetch analysis results for a specific file	Yes
/delete_dataset	DELETE	Delete a selected upload record	Yes

Figure 5: Summary of backend endpoints enabling interaction between frontend, ML models, and database.

Evaluation of Design Success

The effectiveness of the design is evident in how closely it aligns with the project goals:

- The **intuitive frontend** fulfilled Orefox's requirement for a non-technical user interface.
- The **detection-classification pipeline** offered reliable analytical insights despite lacking labeled training data from the client.
- The **chatbot** served as a soft knowledge base, reducing the learning curve for new users.
- The **modular backend** ensured maintainability and a clean separation of concerns.

While the Azure cloud deployment was temporarily deactivated due to the free student subscription expiry, the system was successfully demonstrated to the client and evaluated by the tutor. All core functionalities, uploading, ML analysis, result display, and user interaction, were fully implemented and operational at the time of handover. The artefact can still be run locally with the available code and cloud deployment instructions will be handed over to the client.

Overall, the design leveraged core knowledge in ML, full-stack development, and cloud integration, reflecting the interdisciplinary application of skills developed throughout the degree. The components work cohesively to deliver a direct solution to Orefox's geoscientific exploration goals.

Outcomes

This section presents the testing and quality control activities used to validate the functionality of CopperCore AI. Given the multi-component nature of the system, unit tests were designed for each core module, including detection, classification, ingestion, and LLM integration, focusing on verifying functionality, input-output consistency, and error handling. These tests were executed using Python's unittest and pytest frameworks. All the tests and their outputs can be found in the appendix to view in detail.

Detection Module

The detection pipeline relies on an autoencoder model to flag anomalous geophysical patterns. A unit test was created for the run_autoencoder_pipeline() function. To isolate logic, all external dependencies (e.g., rasterio, pyproj, and TensorFlow) were mocked. A synthetic GeoTIFF band and a dummy model output were simulated.

The test checked that the function correctly returned a dictionary mapping geographic coordinates to anomaly intensity values. It verified output structure, ensuring types and data shapes aligned with project expectations. This test passed successfully, demonstrating the reliability of the core detection function independently of model files and disk I/O.

Classification Module

The classify_json_only() function, which is responsible for assigning anomaly labels using a trained Random Forest classifier, was tested using Python's unittest.mock. A mocked predict_proba output was used to simulate inference. The test confirmed that the output:

- Preserved original coordinates and intensity values
- Returned a binary classification field (0 or 1)
- Followed the expected format and type structure

This confirmed that the classification module correctly interprets detection output and returns consistent predictions for downstream use.

Ingestion and API Tests

Unit tests for ingestion covered multiple scenarios:

1. **QueryExistingDatasetsTests**:
    - Confirmed that a GET request to /query returns user-uploaded dataset metadata with a 200 status.
    - Validated that incorrect methods (POST for instance) trigger a 400 Bad Request.
2. **UploadDatasetNoDBTests**:
    - Tested successful file upload via POST, returning a valid assigned filename.
    - Simulated a missing file scenario, ensuring that the system returns an appropriate 400 error.
3. **DeleteDatasetMockedTests**:
    - Verified dataset deletion when a valid filename is present.

- Ensured appropriate errors are raised when a filename is missing or invalid.

4. **ValidateFileTests**:
    - Tested validation logic for uploaded files, including edge cases (multiple dots, wrong extension, or missing extension).

All 10 ingestion-related unit tests passed successfully, indicating stable API behavior and robust input validation.

LLM Integration Tests

Two unit tests were created to validate LLM-related API functions:

1. **Query API Test**: Ensured that submitting a query without an existing thread creates a new thread and returns a mock response and thread ID with a 200 status code.
2. **Chat History Test**: Verified that passing a thread ID returns all associated messages correctly from the database, ensuring thread-specific continuity.

These tests passed successfully, confirming the chatbot's backend logic works as intended, supporting seamless conversation flow and response logging.

Summary

A total of 20+ unit tests were executed across all core modules. Each test verified expected outputs, edge cases, and proper error handling. The testing efforts ensured system robustness and functional correctness across:

- Geospatial anomaly detection
- Classification workflows
- User-facing APIs (upload, delete, query)
- LLM query processing and history retrieval

These tests form the foundation of our project's quality assurance strategy, allowing confident delivery of a functional and maintainable system.

Group Reflection

This section provides a reflective analysis of our team's experiences during the CopperCore AI project. Our reflection is structured around three themes: (a) collaboration to address challenges, (b) new tools and technologies learned, and (c) how we handled changes and scope adjustments throughout the project lifecycle.

Team Collaboration in Problem Solving

From the outset, collaboration was essential to managing the complexity and scope of the CopperCore AI project. Our team brought together students from diverse academic backgrounds, including Maryam and Carla from cybersecurity, John and Dili from software development, and Sicheng from computer science. This range of expertise initially posed challenges in knowledge alignment, but ultimately became one of the team's greatest strengths.

A significant example of collaboration occurred during the detection and classification phase. Initially, Maryam was assigned to classification, while Carla and Dili led detection. However, due to a lack of labeled data and technical hurdles, the detection pipeline stalled. Maryam quickly pivoted to detection, assisting Carla and Dili in developing and debugging the unsupervised model. Carla, in turn, shifted to classification to ensure progress on both fronts. This mutual flexibility prevented project bottlenecks and maintained momentum.

John and Sicheng, responsible for frontend and backend development respectively, also had to collaborate closely with the detection and classification teams to visualize model outputs on interactive maps and integrate the ML models with backend pipelines. John learned how to integrate Leaflet map overlays with frontend components, and contributed to rendering anomaly intensities clearly for end-users. Shared working and brainstorming sessions in person every week were used to solve integration issues and ensure consistency across frontend and backend.

We conducted weekly sprint retrospectives that functioned as dynamic checkpoints to track progress, realign priorities, and redistribute workload. These stand-ups proved essential for responding to issues like client delays and detection model setbacks.

Learning New Tools and Technologies

The CopperCore AI project demanded that team members move well beyond their prior coursework and rapidly upskill in new technical areas. For most of the team, working with geophysical data, machine learning pipelines, and Azure infrastructure was a first-time experience.

Maryam learned autoencoder-based anomaly detection in TensorFlow, alongside pixel normalization and patch extraction from large GeoTIFFs. Carla and Dili, also unfamiliar with geophysical data, quickly developed preprocessing pipelines and validation tools to process magnetic survey inputs. These tasks were critical in ensuring the model could identify meaningful anomalies.

John, tasked with frontend responsibilities, self-learned how to build map overlays using Leaflet and connected them with Django APIs to display anomaly locations. He also collaborated on formatting the output from classification to ensure frontend compatibility.

Sicheng took the lead in backend development and Azure deployment, despite having no previous experience with Azure hosting. He configured Django with PostgreSQL, managed file storage in Azure Blob Containers, and implemented backend endpoints for dataset upload, detection output, and chatbot functionality.

The LLM-based chatbot also presented a steep learning curve. John and Sicheng both contributed to integrating the LLM, and managing history-based query sessions, all components not covered in previous units. Through this cross-domain exposure, each team member gained valuable experience in bridging theory and practice.

Handling Changes and Adjusting Scope

The most pivotal change was our mid-project pivot from a supervised to an unsupervised learning pipeline. Originally, the detection model was intended to be trained on a labeled CSV provided by Orefox. However, the client was unable to supply this data on time. Facing tight deadlines, the team decided during Sprint 5 to abandon the supervised approach and adopt an autoencoder-based unsupervised model instead.

This decision cascaded through our entire workflow. The detection model had to be redesigned, requiring fresh training and evaluation metrics. This, in turn, delayed the classification phase and necessitated reassigning tasks: Maryam joined detection efforts, Carla took over classification. The timeline was revised accordingly, and the Gantt chart was updated to reflect new dependencies and sprint priorities.

As a result, we had to shift from our standard agile methodology into a more adaptive crisis-management approach. Rather than following our original sprint plan and fixed task allocations, we reorganized each sprint around the most immediate roadblocks. Each week began with a discussion of what was blocking progress, followed by a strategic reassignment of roles to meet critical needs. For example, Maryam, originally responsible for classification, was reassigned to support detection alongside Carla and Dili, focusing the team's efforts on getting any viable model operational first. Once that goal was achieved, Carla resumed classification development.

Our weekly meetings evolved into active coordination points for this strategy, ensuring everyone was aligned on new priorities. This need-based restructuring reflected industry agile practices where sprint plans are fluid and respond to external constraints and blockers.

This shift in working approach significantly improved team responsiveness and kept the project on track despite considerable uncertainty. Our ability to reprioritize and reassign work dynamically enabled us to meet client expectations and deliver a complete solution within the deadline.

Throughout the project, our team continually adapted scope and deliverables in response to changing realities, from client constraints to technical learning curves. These experiences

mirror standard industry scenarios where agile teams must make decisions quickly, reprioritize on the fly, and deliver value under imperfect conditions.

Overall, this project was a meaningful learning experience that tested and strengthened our group's collaboration, technical agility, and resilience. The reflections presented above capture the evolution of our team practices and mindset in response to real-world project dynamics.

Appendix A: Prototype Video

We have recorded a prototype video to further demonstrate the functionalities of the system and their use. The video has been uploaded to OneDrive for better viewing.

Please click here to view the video.

Appendix B: Sprint Logs

This appendix documents the week-by-week sprint retrospectives conducted over 9 sprints from April 1 to May 31. Each entry details progress highlights, blockers, lessons learned, and resolutions made during each increment. These records were used to guide adjustments to our task assignments, Gantt chart, Trello board, and sprint goals.

Sprint 1 (April 1–7)

Highlights: Project setup began with clear task distribution across frontend, backend, and ML components. Initial client expectations clarified.
Challenges: Delay in frontend base layout. Some team members had to wait for dependent tasks.
Lessons & Resolutions: Identified the need for stronger cross-role awareness to avoid idle time.

Sprint 2 (April 8–14)

Highlights: Frontend structure with sidebar/topbar completed. Detection model file structure set up.
Challenges: Backend API and Azure setup blocked progress. Frontend not ready for integration.
Lessons & Resolutions: Need for early deployment testing and task reprioritization.

Sprint 3 (April 15–21)

Highlights: Detection preprocessing steps clarified, team experimented with patching strategies.
Challenges: CSV file from client was not received. Supervised ML strategy put on hold.
Lessons & Resolutions: Project scope highly dependent on client input; alternative unsupervised methods explored.

Sprint 4 (April 22–28)

Highlights: Pivot to autoencoder agreed upon. Cross-team discussion accelerated learning.
Challenges: Detection model rework caused confusion in task roles. Versioning issues began to emerge.
Lessons & Resolutions: Needed a clearer contingency plan for client dependencies.

Sprint 5 (April 29–May 5)

Highlights: Maryam and Carla successfully trained detection model with new patching method.
Challenges: Frontend integration stalled due to incompatible formats. Backend waiting on detection output.
Lessons & Resolutions: Model outputs must be designed with frontend/backend compatibility in mind.

Sprint 6 (May 6–12)

Highlights: Detection finalized; classification thresholding method decided. Major drop in burndown chart.
Challenges: Backend storage and Azure deployment failed due to free tier limits.
Lessons & Resolutions: Infrastructure planning must consider resource availability and limits.

Sprint 7 (May 13–19)

Highlights: Classification model implemented and predictions linked to frontend map display.
Challenges: Version mismatches caused deployment delays. Minor feature bugs in chatbot.
Lessons & Resolutions: Continuous integration testing should have started earlier.

Sprint 8 (May 20–26)

Highlights: Rapid progress in frontend UI, chatbot button popup, and classification overlays.
Challenges: Azure VM began to show instability. Team workload reached maximum capacity.
Lessons & Resolutions: Buffer time is essential in final sprints.

Sprint 9 (May 27–31)

Highlights: Final features tested and deployed. Trello board closed with all cards marked complete.
Challenges: Azure free subscription expired before final submission. Notified tutor and client.
Lessons & Resolutions: Always plan ahead for platform-based resource expiry. Saved all backend outputs locally.

Appendix C: Unit Tests and Outputs

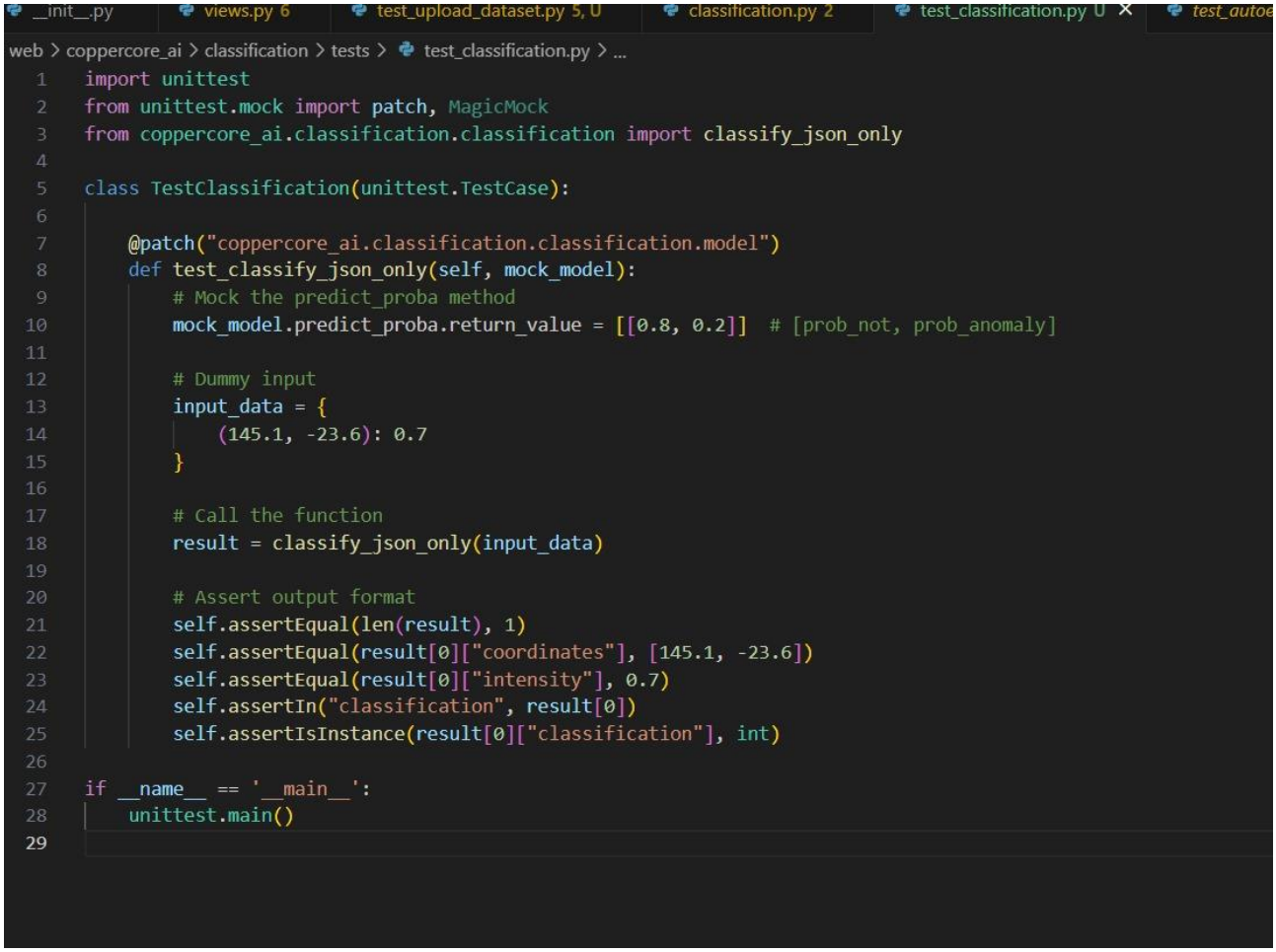
Detection Model Testing

b > coppercore_ai > detection > tests > test_autoencoder_pipeline.py > test_run_autoencoder_pipeline_returns_results

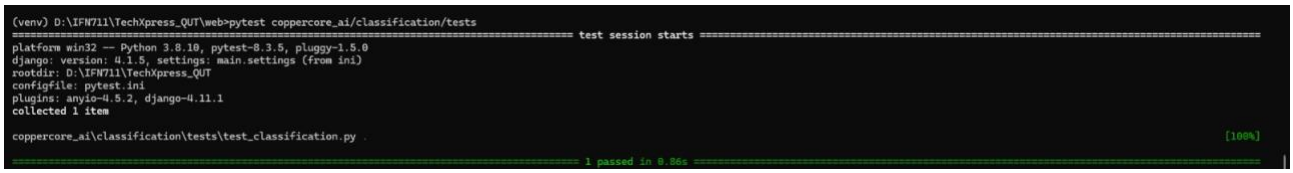
```python
import numpy as np
from unittest.mock import patch, MagicMock
from rasterio.transform import from_origin
from coppercore_ai.detection.autoencoder_pipeline import run_autoencoder_pipeline

@patch("coppercore_ai.detection.autoencoder_pipeline.tf.keras.models.load_model")
@patch("coppercore_ai.detection.autoencoder_pipeline.rasterio.open")
@patch("coppercore_ai.detection.autoencoder_pipeline.Transformer.from_crs")
def test_run_autoencoder_pipeline_returns_results(mock_transformer, mock_rasterio_open, mock_load_model):
    # Create dummy GeoTIFF band
    dummy_band = np.random.rand(128, 128).astype(np.float32)

    # Mock rasterio.open to simulate reading the band
    mock_src = MagicMock()
    mock_src.read.return_value = dummy_band
    mock_src.transform = from_origin(100.0, 100.0, 1.0, 1.0)  # Proper Affine transform
    mock_src.crs = "EPSG:3857"
    mock_rasterio_open.return_value.__enter__.return_value = mock_src

    # Mock pyproj Transformer
    transformer_mock = MagicMock()
    transformer_mock.transform.side_effect = lambda x, y: (x + 0.123, y + 0.456)
    mock_transformer.return_value = transformer_mock

    # Mock Keras autoencoder model
    mock_model = MagicMock()
    mock_model.predict.return_value = dummy_band.reshape(-1, 64, 64, 1) * 0.9
    mock_load_model.return_value = mock_model

    # Run the pipeline
    result = run_autoencoder_pipeline("dummy.tif", model_filename="autoencoder_tmi.h5", top_n=3, patch_size=64, stride=64)

    # Assert result format
    assert isinstance(result, dict)
    assert len(result) == 3
    for coords, intensity in result.items():
        assert isinstance(coords, tuple)
        assert len(coords) == 2
        assert all(isinstance(coord, float) for coord in coords)
        assert isinstance(intensity, float)
```

```
(venv) D:\IFN711\TechXpress_QUT\web>pytest coppercore_ai/detection/tests/
============================================== test session starts ==============================================
platform win32 -- Python 3.8.10, pytest-8.3.5, pluggy-1.5.0
rootdir: D:\IFN711\TechXpress_QUT\web
plugins: anyio-4.5.2
collected 1 item

coppercore_ai\detection\tests\test_autoencoder_pipeline.py .                                            [100%]

=============================================== 1 passed in 6.91s ===============================================
```

Classification Model Testing

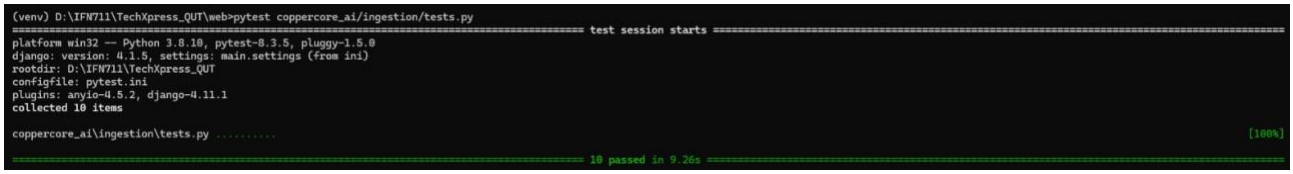


```python
import unittest
from unittest.mock import patch, MagicMock
from coppercore_ai.classification.classification import classify_json_only

class TestClassification(unittest.TestCase):

    @patch("coppercore_ai.classification.classification.model")
    def test_classify_json_only(self, mock_model):
        # Mock the predict_proba method
        mock_model.predict_proba.return_value = [[0.8, 0.2]]  # [prob_not, prob_anomaly]

        # Dummy input
        input_data = {
            (145.1, -23.6): 0.7
        }

        # Call the function
        result = classify_json_only(input_data)

        # Assert output format
        self.assertEqual(len(result), 1)
        self.assertEqual(result[0]["coordinates"], [145.1, -23.6])
        self.assertEqual(result[0]["intensity"], 0.7)
        self.assertIn("classification", result[0])
        self.assertIsInstance(result[0]["classification"], int)

if __name__ == '__main__':
    unittest.main()
```



```
(venv) D:\IFN711\TechXpress_QUT\web>pytest coppercore_ai/classification/tests
============================= test session starts =============================
platform win32 -- Python 3.8.10, pytest-8.3.5, pluggy-1.5.0
django: version: 4.1.5, settings: main.settings (from ini)
rootdir: D:\IFN711\TechXpress_QUT
configfile: pytest.ini
plugins: anyio-4.5.2, django-4.11.1
collected 1 item

coppercore_ai\classification\tests\test_classification.py .              [100%]

============================== 1 passed in 0.86s ==============================
```

Ingestion Testing




```
from django.test import SimpleTestCase
from django.core.files.uploadedfile import SimpleUploadedFile
from unittest import TestCase
from django.test import RequestFactory
from http import HTTPStatus
import json
from unittest.mock import patch, Mock, MagicMock
from django.http import HttpRequest
from coppercore_ai.ingestion.views import upload_dataset, delete_dataset,
query_existing_datasets, FILE_STATUS_UPLOADED, validate_file
from coppercore_ai.ingestion.helper import USER_ID_KEY_IN_SESSION


class QueryExistingDatasetsTests(TestCase):
    def setUp(self):
        self.factory = RequestFactory()


    @patch('coppercore_ai.ingestion.views.query_existing_datasets_from_db')
    @patch('coppercore_ai.ingestion.views.get_user_id_from_session')
    def test_query_existing_datasets_success(self, mock_get_user_id, mock_query_db):
        mock_get_user_id.return_value = 1


        mock_query_db.return_value = [
    Mock(original_file_name="file1", file_type="tif", upload_time="2023-01-01",
         status=FILE_STATUS_UPLOADED, assigned_name="assigned1.tif"),
    Mock(original_file_name="file2", file_type="tif", upload_time="2023-01-02",
```

```
        status=FILE_STATUS_UPLOADED, assigned_name="assigned2.tif"),
]


        request = self.factory.get('/query/')
        request.session = {USER_ID_KEY_IN_SESSION: "1"} # set user id key for session
        request.user = Mock(is_authenticated=True)

        response = query_existing_datasets(request)

        self.assertEqual(response.status_code, HTTPStatus.OK)
        data = json.loads(response.content)  # decode JSON
        self.assertEqual(data['message'], 'Success')
        self.assertEqual(len(data['data']), 2)
        self.assertEqual(data['data'][0]['filename'], "file1.tif")
        self.assertEqual(data['data'][0]['assigned_filename'], "assigned1.tif")
        # Check the status string matches your mapping in views.py
        self.assertEqual(data['data'][0]['status'], 'Uploaded')
        self.assertEqual(data['data'][1]['status'], 'Uploaded')

    def test_query_existing_datasets_invalid_method(self):
        request = self.factory.post('/query/')
        request.session = {USER_ID_KEY_IN_SESSION: "1"} # set user id key for session
        request.user = Mock(is_authenticated=True)

        response = query_existing_datasets(request)

        self.assertEqual(response.status_code, HTTPStatus.BAD_REQUEST)
        data = json.loads(response.content)  # decode JSON
        self.assertEqual(data['error'], 'Invalid Method')
```

```
class UploadDatasetNoDBTests(TestCase):
    def setUp(self):
        self.factory = RequestFactory()

    @patch('coppercore_ai.ingestion.views.get_user_id_from_session')
    @patch('coppercore_ai.ingestion.views.validate_file')
    @patch('coppercore_ai.ingestion.views.preprocess')
    @patch('coppercore_ai.ingestion.views.assign_new_name')
    @patch('coppercore_ai.ingestion.views.init_and_save_new_db_record')
    @patch('coppercore_ai.ingestion.views.run_autoencoder_pipeline')
    @patch('coppercore_ai.ingestion.views.classify_json_only')
    @patch('coppercore_ai.ingestion.views.bulk_insert_analysed_data')
    @patch('os.remove')
    def test_upload_success_no_db(self, mock_remove, mock_bulk_insert, mock_classify,
mock_run_ae,
                                  mock_init_save, mock_assign_name, mock_preprocess,
                                  mock_validate, mock_get_user):
        mock_get_user.return_value = 1
        mock_validate.return_value = (True, "")
        mock_assign_name.return_value = "assigned.tif"
        mock_run_ae.return_value = {"coords": "data"}
        mock_classify.return_value = [{"classified": "data"}]

        fake_file = SimpleUploadedFile("file.tif", b"fake content", content_type="image/tiff")
```

```
        request = self.factory.post('/upload/', {'file': fake_file})
        request.FILES['file'] = fake_file
        request.session = {}
        request.user = Mock(is_authenticated=True)

        response = upload_dataset(request)

        self.assertEqual(response.status_code, HTTPStatus.OK)
        data = json.loads(response.content)
        self.assertIn('message', data)
        self.assertEqual(data['message'], 'File uploaded successfully')
        self.assertEqual(data['assigned_filename'], 'assigned.tif')

    def test_upload_no_file(self):
        request = self.factory.post('/upload/', {})
        request.session = {}
        request.user = Mock(is_authenticated=True)

        response = upload_dataset(request)
        data = json.loads(response.content)
        self.assertEqual(response.status_code, HTTPStatus.BAD_REQUEST)
        self.assertEqual(data['error'], 'No file uploaded')

class DeleteDatasetMockedTests(TestCase):

    @patch("coppercore_ai.ingestion.views.query_existing_datasets_from_db")
    @patch("coppercore_ai.ingestion.views.DetectionResults.objects.filter")
    @patch("coppercore_ai.ingestion.views.FileUploadedDatasets.objects.filter")
    @patch("coppercore_ai.ingestion.views.get_user_id_from_session")
```

```
def test_successful_deletion(
    self, mock_get_user_id, mock_file_filter, mock_result_filter, mock_query
):
    mock_get_user_id.return_value = 1

    mock_file = MagicMock()
    mock_file.original_file_name = "example"
    mock_file.file_type = "tif"
    mock_file.assigned_name = "example123"
    mock_file.id = 99
    mock_query.return_value = [mock_file]

    mock_file_filter.return_value.delete.return_value = None
    mock_result_filter.return_value.delete.return_value = None

    request = HttpRequest()
    request.method = "GET"
    request.GET["filename"] = "example.tif"
    request.session = {}
    request.user = MagicMock(is_authenticated=True)

    response = delete_dataset(request)
    self.assertEqual(response.status_code, HTTPStatus.OK)
    self.assertEqual(json.loads(response.content), {"message": "Success"})

@patch("coppercore_ai.ingestion.views.query_existing_datasets_from_db")
@patch("coppercore_ai.ingestion.views.get_user_id_from_session")
def test_missing_filename(self, mock_get_user_id, mock_query_existing):
    mock_get_user_id.return_value = 1
```

```
        mock_query_existing.return_value = []

        request = HttpRequest()
        request.method = "GET"
        request.GET = {}  # No filename
        request.session = {}
        request.user = MagicMock(is_authenticated=True)

        response = delete_dataset(request)
        self.assertEqual(response.status_code, HTTPStatus.BAD_REQUEST)
        self.assertEqual(json.loads(response.content), {"error": "Invalid Filename"})

class ValidateFileTests(SimpleTestCase):

    def test_valid_file(self):
        file = SimpleUploadedFile("sample.tif", b"dummy content")
        valid, msg = validate_file(file, user_id=1)
        self.assertTrue(valid)
        self.assertIsNone(msg)

    def test_invalid_file_multiple_dots(self):
        file = SimpleUploadedFile("sample.invalid.tif", b"dummy content")
        valid, msg = validate_file(file, user_id=1)
        self.assertFalse(valid)
        self.assertEqual(msg, 'Invalid File Format')

    def test_invalid_file_wrong_extension(self):
        file = SimpleUploadedFile("sample.txt", b"dummy content")
        valid, msg = validate_file(file, user_id=1)
```

```
    self.assertFalse(valid)
    self.assertEqual(msg, 'Invalid File Format')

def test_invalid_file_no_extension(self):
    file = SimpleUploadedFile("sample", b"dummy content")
    valid, msg = validate_file(file, user_id=1)
    self.assertFalse(valid)
    self.assertEqual(msg, 'Invalid File Format')
```

LLM/Chatbot Testing

```python
import json
from unittest import TestCase
from unittest.mock import patch, MagicMock
from django.http import HttpRequest
from http import HTTPStatus
from coppercore_ai.llm_integration.views import query, get_chat_history


class MockedLLMViewTests(TestCase):

    @patch("coppercore_ai.llm_integration.views.get_user_current_thread_count")
    @patch("coppercore_ai.llm_integration.views.get_user_id_from_session")
    @patch("coppercore_ai.llm_integration.views.call_openai_api")
    @patch("coppercore_ai.llm_integration.views.ChatHistory.save")
    def test_query_success(self, mock_save, mock_call_openai, mock_get_user_id, mock_thread_count):
        mock_get_user_id.return_value = 1
        mock_call_openai.return_value = "Mocked response from LLM"
        mock_thread_count.return_value = 0  # Simulate zero threads

        request = HttpRequest()
        request.method = "POST"
        request._body = json.dumps({
            "query": "What is porphyry copper?",
            "thread_id": None
        }).encode("utf-8")
        request.user = MagicMock(is_authenticated=True)
        request.session = {}

        response = query(request)
        self.assertEqual(response.status_code, HTTPStatus.OK)
        data = json.loads(response.content)
        self.assertEqual(data["response_content"], "Mocked response from LLM")
        self.assertEqual(data["thread_id"], 1)
        self.assertIn("response_time", data)

    @patch("coppercore_ai.llm_integration.views.get_user_id_from_session")
    @patch("coppercore_ai.llm_integration.views.ChatHistory.objects")
    def test_get_chat_history_success(self, mock_objects, mock_get_user_id):
        mock_get_user_id.return_value = 1

        mock_msg = MagicMock(
            message_date="2024-01-01",
            message_content="mock query",
            response_content="mock response",
            response_date="2024-01-01",
            thread_id=1
        )

        # Mock chain: filter().only().order_by().__getitem__() -> [mock_msg]
        mock_qs = MagicMock()
        mock_qs.only.return_value.order_by.return_value.__getitem__.return_value = [mock_msg]
        mock_objects.filter.return_value = mock_qs

        request = HttpRequest()
        request.method = "GET"
        request.user = MagicMock(is_authenticated=True)
        request.session = {}

        response = get_chat_history(request)
        self.assertEqual(response.status_code, HTTPStatus.OK)
        data = json.loads(response.content)
        self.assertEqual(len(data["chat_history"]), 1)
        self.assertEqual(data["chat_history"][0]["query"], "mock query")
        self.assertEqual(data["chat_history"][0]["response"], "mock response")
```

```
(venv) D:\IFN711\TechXpress_QUT\web>pytest coppercore_ai/llm_integration/tests/test_llm.py
============================= test session starts =============================
platform win32 -- Python 3.8.10, pytest-8.3.5, pluggy-1.5.0
django: version: 4.1.5, settings: main.settings (from ini)
rootdir: D:\IFN711\TechXpress_QUT
configfile: pytest.ini
plugins: anyio-4.5.2, django-4.11.1
collected 2 items

coppercore_ai\llm_integration\tests\test_llm.py ..                       [100%]

============================== 2 passed in 1.00s ==============================
```