MEMORANDUM

Abstract / Executive Summary

This memorandum outlines the strategic and technical proposal for the Casual Academic Time Allocation Management System (CATAMS) Capstone Project.

Key Highlights:

- **Project Goal:** Establish a reliable, intuitive timesheet system for lecturers, tutors, and HR.
- **Core Philosophy:** Build a long-term, maintainable internal asset, guided by User-Centric, Internalized Workflow, and Future-Proof design principles.
- **Architectural Strategy:** Pragmatically start with a **Modular Monolith**, designed for future evolution towards **Microservices**.
- **Technology Selection:** Candidate options for the backend technology stack are presented for consideration.
- **Project Management:** Utilizes the **MoSCoW method** for MVP scope definition, coupled with phased delivery and robust risk management.

This proposal details a high-quality, future-extensible Minimum Viable Product (MVP) and lays a solid foundation for its subsequent evolution.

1. Project Context & Objectives

Summary:

This section outlines the necessity and core functionalities of the CATAMS project, and specifies that the COMP5709 Capstone Project aims to establish a reliable timesheet management system, adhering to industry best practices, and laying the groundwork for future architectural evolution.

1.1 Project Overview

The University of Sydney's Master of Computer Science program success has increased casual tutor management needs. Existing tools lack efficiency-enhancing features. CATAMS aims to provide a clear, straightforward system for lecturers, tutors, and HR to enter and review work hours.

Key Features Needed (from Project Description):

- Lecturers enter hours (description, quantity, pay rate).
- Tutors view their hours, request changes (lecturer approval).
- Dashboard summarizes hours, budget used/remaining.

- Approved hours notified to HR for final review.

This system is expected to significantly impact the teaching team and be utilized long-term.

1.2 Proposed Project Objectives

- **Establish a Reliable Timesheet System:** Construct a stable, clear system for timesheet entry, approval, and summarization.
- **Adhere to Industry Best Practices:** Adopt DDD, CI/CD, and containerization for code-business alignment and automated deployment.
- **Lay Foundation for Future Evolution:** Deliver MVP with clear Bounded Contexts, prepared for Microservices Architecture evolution.

2. Core Design Philosophy

Summary:

Project goal: a **long-term, reliable, and maintainable internal asset**. Design philosophies: **User-Centric, Internalized Workflow, and Future-Proof**, ensuring adaptability and longevity.

2.1 User-Centric

Provide users (Coordinators, Tutors, HR) a stable, easy-to-use, predictable platform, freeing them from manual processes.

Design Implications:

- **Minimal Cognitive Load:** Simple UI, relevant info only.
- **Clear Feedback & Predictability:** Immediate visual feedback; predictable operation outcomes.
- **Fault-Tolerant Design:** Undo capabilities; secondary confirmation for critical actions.

2.2 Internalized Workflow

Build an **"Opinionated System."** Excellent business logic should be internalized within the project's workflow. Through **"Positive Bureaucracy,"** guide users to best practices, reducing error-prone discretion.

Design Implications:

- **State-Driven UI:** UI element availability determined by business object state.
- **Enforced Business Rules:** System enforces core rules (e.g., no future

timesheets, budget checks).
- **Single Source of Truth:** Centralized management of business states and history, preventing inconsistencies.

2.3 Future-Proof

System must have high **flexibility and scalability** to adapt to future changes and needs, considering potential paradigm shifts in university education.

Design Implications:

- **Evolutionary Architecture:** Modular Monolith lays groundwork for smooth Microservices transition.
- **Configurable Business Logic:** Volatile rules decoupled from code (config files/database) for easy policy adjustments.
- **API-First Design:** Clear internal APIs for core functions, enabling future integration with other university systems.

3. Core Architectural Strategy & Evolution

Summary:

Long-term vision: **Microservices**. Pragmatic start: **Modular Monolith** (Microservices Ready), deployed with **cloud-native strategy**, enabling incremental evolution.

3.1 Microservices as the Ideal Long-Term Vision

Microservices architecture is the ideal "final form" for CATAMS (serving thousands), based on:

- **Independent Evolution Rhythm:** Decoupling stable (timesheet) from flexible (reporting) modules.
- **Differing Scalability Needs:** Independent scaling of high-load components (e.g., timesheet submission) for cost savings.
- **High Reliability & Fault Isolation:** Non-core service failures do not impact core operations.

3.2 Capstone Project Trade-off: Why We Start with a Modular Monolith

Directly building microservices for a single-person, single-semester project is high-risk. A mature engineering trade-off is essential.

- **Architectural Style: Modular Monolith:** Represents **"pragmatism and foresight."** Achieves high decoupling at code level, ensuring business logic independence and maintainability. **Crucially, early application of**

Domain-Driven Design (DDD) principles ensures internal modules are designed with clear boundaries (Bounded Contexts), making future refactoring and extraction into microservices significantly more straightforward. At the deployment level, it maintains the simplicity of a single application, avoiding excessive operational complexity for a single-developer project.

- **Evolutionary Path (Upgrade & Downgrade Routes):** This architecture is **"Microservices Ready."**
    - **Upgrade Path (Gradual Evolution to Microservices):** If time and project scope allow, future evolution will follow the **Strangler Fig Pattern** for **incremental extraction** of modules into independent microservices. This involves **gradually redirecting traffic** from the monolithic application to new service implementations, enabling a smooth transition and continuous delivery of value.
    - **Downgrade Path (Contingency for Partial Transformation):** This **incremental refactoring** approach ensures that even if a full microservices transformation is not completed within the semester due to time or complexity constraints, a usable and stable **hybrid version** will be delivered. This hybrid system will have core functionalities already benefiting from the modular design, providing a solid foundation for future work.

3.3 Deployment Strategy: Cloud-Native

Cloud-native deployment is the most cost-effective given fluctuating usage patterns.

- **Containerization:** Application packaged as a **Docker** image.
- **Continuous Integration/Continuous Deployment (CI/CD):** Basic CI/CD pipeline (e.g., GitHub Actions).
- **Cloud Platform & Services:** Deployment on **AWS** using **AWS Fargate** (serverless container service) and **Amazon RDS for PostgreSQL** (managed database service).

4. Technology Stack Selection & Rationale

Summary:

Technology selection serves long-term goals. Evaluated based on **Robustness & Maintainability, AI-Friendliness, and Ecosystem Maturity & Developer Productivity**. Candidate options for the backend technology stack are presented for discussion.

4.1 Principle 1: Robustness & Maintainability

Prioritize code robustness and maintainability for a long-term, financial data-handling internal asset.

- **Rationale:** Choose **Strongly-typed Languages** (e.g., Java) to catch errors at compile-time (critical for payroll), enhancing code readability and safe future modifications.

4.2 Principle 2: LLM-Friendliness

Future maintenance will highly benefit from AI-assisted programming tools.

- **Rationale:** Select mainstream languages with vast, high-quality open-source corpora (e.g., Python, JavaScript/TypeScript, Java) for better AI code suggestions. **The integration of practices like Spec-oriented Development and Test-Driven Development (TDD) can further enhance the effectiveness of AI code generation by providing clear, executable specifications.**

4.3 Principle 3: Ecosystem Maturity & Developer Productivity

Deliver a feature-complete system within limited time by leveraging mature ecosystems.

- **Rationale:** Choose frameworks with comprehensive documentation, an active community, and rich support for production-proven third-party libraries (e.g., Spring Boot, NestJS) to accelerate development.

4.4 Candidate Stack Evaluation & Decision

Two strong backend technology stack candidates are considered: **Java with Spring Boot 3+** and **TypeScript with NestJS**.

- **Java with Spring Boot 3+:**
  - **Pros:** Extreme robustness, mature enterprise ecosystem, high performance (comparable to Node.js in I/O-intensive scenarios with modern features), alignment with my experience, strong modular boundary governance (Spring Modulith), systematic microservices evolution ladder (Spring Cloud), long-term maintainability, predictability, and effective cold start mitigation (Spring Native + GraalVM).
  - **Potential Trade-offs:** Potentially slower initial development velocity, larger container footprint if native images are not used, and additional native image build time.
- **TypeScript with NestJS:**

    - **Pros:** Unified language for front-end and back-end, modern and productive development experience (rapid scaffolding), top-tier AI-friendliness, fast startup, low memory footprint, and efficient for I/O-intensive applications.
    - **Potential Trade-offs:** Ecosystem stability may vary, modular boundary governance relies more on manual enforcement, microservices governance requires manual assembly of third-party packages, potential long-term maintenance challenges due to frequent updates, and less optimal performance for CPU-intensive tasks.

Decision Point:

Both technology stacks offer compelling advantages. My preliminary recommendation leans towards **Java with Spring Boot 3+** due to its paramount strengths in **stability, predictable evolution paths, and architectural governance**, critical for a long-term internal asset. The final choice of technology stack will be determined in consultation with the supervisor, considering broader perspectives and team skill sets.

5. Stakeholder Requirements & Design Considerations

Summary:

This section analyzes requirements from **Primary Users, Key Decision-Makers, and Secondary Stakeholders**, proposing corresponding system design considerations.

Note: This analysis is based on a **preliminary understanding of potential needs**, derived from the project description and common enterprise system requirements. **Formal, in-depth requirements gathering and validation will be a critical early phase of the project, involving meetings with the lecturer and an internal representative to fully understand what is needed.**

5.1 Tier 1: Primary Users (Lecturers, Tutors, HR)

- **User Lifecycle Management:** Seamless onboarding/offboarding; data archiving/anonymization.
- **Data Integrity:** Flawless calculations; **database transactions** for **atomicity**.
- **Auditability, Transparency & System Resilience:** Implement **Immutable Audit Logs** for transparency, auditability, and system recovery (as a "Re-do Log").

5.2 Tier 2: Key Decision-Makers (Supervisor, Internal Representative)

- **Maintainability & Low Coupling:** Deliver **comprehensive technical documentation** (architecture diagrams, API docs, deployment guides) and well-commented code.

- **Requirements Clarification:** Establish clear communication (user story interviews, prototype demos) with the "Internal Representative" to avoid rework.

5.3 Tier 3: Secondary Stakeholders (The School, HR/Finance)

- **Business Logic Replaceability:** Encapsulate core business rules (e.g., approval hierarchies, pay rates) in configurable modules (e.g., using **Strategy Pattern**) for easy adaptation to policy changes.

6. Risk Management, Scope Control & Delivery Strategy

Summary:

This section aims to identify potential project risks and formulate pragmatic strategies for scope control and successful delivery. Risks include **Scope Creep, DevOps Complexity, Compliance & Security Gaps, and Evolution Drift**. Strategies involve **MoSCoW method** for MVP scope, phased delivery, compliance baselines, and architectural grounding.

6.1 Identifying Core Risks

- **Scope Creep:** Feature list may expand beyond feasible work, delaying core delivery.
- **DevOps Complexity:** Cloud-native stack configuration may consume excessive time.
- **Compliance & Security Gaps:** Risk of violating Australian Privacy Act with PII/financial data.
- **Evolution Drift:** Microservices-ready architecture may remain a prototype without clear baselines/documentation.

6.2 Defining the MVP Scope

Use **MoSCoW method** to strictly define delivery scope for this semester, mitigating scope creep.

- **Must Have:** Core timesheet CRUD, two-level approval, user authentication/RBAC, basic dashboard, immutable audit log.
- **Should Have:** Automated email notifications, full tutor lifecycle management.
- **Could Have:** CSV exports, budget-overrun alerts.
- **Won't Have this time:** External HR-system APIs, complex AI predictions, advanced custom reports.

6.3 Pragmatic Delivery & DevOps Strategy

Adopt a phased strategy to manage DevOps complexity.

- **Phase 1: Local-First Development:** Initial development in a **local containerized environment** (e.g., Docker-Compose) using a lightweight database (e.g., local PostgreSQL instance) for rapid iteration.
- **Phase 2: Cloud Deployment & Demo:** Later-stage deployment to a **serverless container service** (e.g., AWS Fargate) for final demonstration.
- **Observability Baseline:** Integrate basic logging (e.g., standardized JSON format outputting to CloudWatch) and monitoring (e.g., exposing Prometheus metrics for core APIs) for troubleshooting.

6.4 Compliance & Security Baseline

Implement minimal but critical measures to address compliance gaps.

- **Data Retention Policy:** Define explicit policy (e.g., 7-year retention for audit logs/personal data, then anonymization).
- **Sensitive Information Encryption: Field-Level Encryption** for sensitive data (e.g., pay rates).
- **Secrets Management:** Manage credentials via a **secure secrets management service** (e.g., AWS Secrets Manager), injected securely via CI/CD.

6.5 Grounding the Evolutionary Architecture

Adopt industry best practices for documenting and planning the architecture to prevent "evolution drift."

- **Architecture Decision Records (ADRs):** Maintain docs/adr/ for documenting significant architectural decisions, providing context for future maintainers.
- **Defining Splitting Criteria:** Explicitly define triggers for future module decomposition into microservices (e.g., performance thresholds, team ownership changes).

7. Project Milestones & Deliverables

Summary:

This section outlines the project's phased development schedule, core deliverables, success metrics, and contingency plan for addressing uncertainties, aiming to ensure efficient and high-quality project completion within the semester.

7.1 Project Milestones & Development Schedule

The project development will follow a phased roadmap designed to ensure MVP delivery within the semester and lay the groundwork for future evolution. This schedule acknowledges the iterative nature of software development and the need for

flexibility in a single-developer Capstone project.

Phase	Key Activities/Outputs	Primary Focus
0. Initial Setup & Discovery	Project planning, environment setup, **formal requirements gathering** with internal stakeholders, and initial domain modeling.	**Understanding Needs & Foundation**
1. Core MVP Development	Design and implementation of core timesheet CRUD, approval workflows, user authentication, and basic dashboard functionalities.	**Building Core System**
2. Hardening & Refinement	Comprehensive testing, performance tuning, security review, and user documentation.	**Ensuring Quality & Reliability**
3. Evolution & Presentation	Incremental microservice extraction (if feasible via **Strangler Fig Pattern**), final polish, project presentation, and thesis submission.	**Demonstrating Vision & Completion**

*Note: This schedule provides a flexible roadmap for a single developer. Agile adjustments and continuous communication with the supervisor will be crucial to manage potential challenges and ensure successful delivery within the semester. The application of **Domain-Driven Design (DDD)** principles from the outset will facilitate a smoother transition towards microservices, allowing for a usable **hybrid version** even if the full microservices transformation is not completed within the semester.*

7.2 Core Deliverables

The project will ultimately deliver the following key artifacts:

- **Source Code Repository:** Utilizing a Mono Repo + sub-repository management format.
- **Documentation:** Including Context Map, OpenAPI specification, Architecture Decision Records (ADR), and User Manual. **Emphasis will be placed on comprehensive documentation to facilitate long-term maintenance and future AI-driven development.**

- **Continuous Integration/Continuous Deployment (CI/CD):** Including GitHub Actions Workflow and Docker Compose files.
- **Test Reports:** Covering Unit Test, Integration Test, Pact Test, and Load Test.
- **Thesis:** Detailing project methodology, performance comparison, and future work.

7.3 Success Metrics

Project success will be assessed based on the following **proposed metrics**, which will be refined upon formal requirements gathering:

- **Feature Coverage:** Achieving high coverage of formally defined user stories.
- **User Acceptance Testing (UAT) Satisfaction:** Achieving strong user satisfaction.
- **Performance Comparison:** Demonstrating minimal performance degradation during evolution from monolithic to microservices architecture (if Phase 2 is undertaken).
- **Documentation Completeness:** Achieving high completeness of ADR, API specification, and testing documentation.

7.4 Contingency Plan

To address uncertainties during project implementation, especially under time constraints, the following contingency plan has been developed:

- **If a full microservices transformation is not achievable within the semester:**
  - A thoroughly tested Modular Monolith will be submitted as the final deliverable.
  - A detailed microservices decomposition blueprint, performance predictions, and experimental design will be provided in the thesis.
  - Comprehensive handover documentation will be prepared to enable subsequent Capstone students to continue the project's evolution.