THE ICT UNIVERSITY

Faculty/School	ICT Faculty
Course Code/Title	CS 4122 Distributed Systems and Cloud Computing
Full Name	NZEKUI NZOUDJIO ALICE CRESSENCE
Matricule	ICTU20241151
Instructor	Daniel Moune
Github	https://github.com/AliceCressence/Cloud-based-distributed-system
E-mail address	nzekuinzoudjio.alice@ictu.edu.cm
Phone	653936899

Project Proposal: A Unified, Cloud-Based Distributed System for University Management

October 15, 2025

Abstract

This document presents a comprehensive proposal for the re-architecture of the university's Learning Management System (LMS) infrastructure. Currently, the institution operates multiple, disparate Moodle instances for academics, finance, and human resources, creating significant operational inefficiencies, data integrity risks, and a disjointed user experience. These isolated systems, or "data silos," necessitate slow, error-prone manual data transfer processes, which negatively impact students, faculty, and administrative staff. The reliance on manual data handling introduces critical business risks, including incorrect billing, delayed access to course materials, and an inability to leverage institutional data for strategic decision-making.

To address these challenges, we propose the development of a unified, cloud-native ecosystem built upon a microservices architecture. This initiative will dismantle the existing data silos by integrating the core Moodle instances through a central, event-driven communication backbone. The solution involves strategically refactoring monolithic Moodle functions, exposing them via internal APIs, and developing a suite of independent, scalable microservices for core business capabilities such as Enrollment, Billing, and Grades.

The proposed system will be deployed on a modern cloud infrastructure (e.g., AWS, GCP, or Azure) and orchestrated by Kubernetes, ensuring high availability, fault tolerance, and elastic scalability. Communication between services will be primarily asynchronous, managed by a RabbitMQ message broker using the Saga pattern to ensure data consistency across distributed transactions. This modern, decoupled architecture will eliminate manual data entry, provide a real-time, 360-degree view of student data, and enable greater organizational agility.

The project is projected to span six months, following an Agile (Scrum) methodology to ensure iterative development and stakeholder alignment. Success will be measured by a combination of technical KPIs (e.g., 99.95% uptime, <200ms API latency) and business KPIs (e.g., 80% reduction in manual data entry, 50% reduction in related support tickets). This project represents a critical investment in the institution's digital infrastructure, promising substantial returns in operational efficiency, student satisfaction, and long-term strategic flexibility, positioning the university as a leader in educational technology.

Contents

1 Introduction and Problem Statement **2**
- 1.1 The Digital Imperative in Higher Education . . . 2
- 1.2 The Core Problem: Data Silos and Manual Integration . . . 2
  - 1.2.1 Impact on Stakeholders . . . 3
  - 1.2.2 Strategic Implications . . . 3

2 Project Scope **4**
- 2.1 In-Scope Activities . . . 4
- 2.2 Out-of-Scope Activities . . . 5

3 Proposed Solution: A Distributed Systems Approach **6**
- 3.1 Scalability & Performance: Achieving True Elasticity . . . 6
- 3.2 Fault Tolerance & Resilience: Designing for Failure . . . 6
- 3.3 Decoupling & Collaboration: Enabling Organizational Agility . . . 7

4 System Design and Architecture **8**
- 4.1 Architectural Model: Microservices and API Gateway . . . 8
  - 4.1.1 Core Microservices Responsibilities . . . 8
- 4.2 Data Model Distributed Transactions . . . 9
  - 4.2.1 Example: Student Enrollment Saga . . . 9
- 4.3 Communication Model . . . 10

5 Technology Stack **11**

6 Project Plan and Timeline **12**
- 6.1 Calendar of Activities . . . 12
- 6.2 Risk Management . . . 13

7 Implementation Strategy **15**
- 7.1 Development Workflow Methodology . . . 15
- 7.2 CI/CD (Continuous Integration/Continuous Deployment) Pipeline . . . 15
- 7.3 Observability Strategy . . . 16

8 Evaluation and Success Metrics **17**
- 8.1 Success Metrics & Key Performance Indicators (KPIs) . . . 17
  - 8.1.1 Technical KPIs . . . 17
  - 8.1.2 Business KPIs . . . 17
- 8.2 Testing Strategy . . . 17
- 8.3 User Acceptance Testing (UAT) Plan . . . 18

9 Conclusion **20**

Chapter 1

Introduction and Problem Statement

1.1 The Digital Imperative in Higher Education

Educational institutions today operate within a highly competitive and complex digital landscape where efficiency, data integrity, and user experience are no longer optional but are fundamental drivers of success. The modern student, accustomed to the seamless, integrated digital ecosystems of consumer technology giants, expects a similarly frictionless experience from their university. The Learning Management System (LMS) is the heart of this experience, serving as the digital campus for virtually all academic activities. However, the organic growth of IT systems has led many institutions, including our own, to adopt a siloed approach to digital services.

The common practice of deploying separate, specialized Moodle instances for different administrative functions—specifically academics, finance, and human resources—has inadvertently created a fractured and profoundly inefficient operational environment. While each instance may be well-suited for its specific domain, they exist as isolated islands of data, leading to a host of cascading problems that accrue significant technical and organizational debt.

1.2 The Core Problem: Data Silos and Manual Integration

The central problem this project addresses is the absence of an automated, real-time communication fabric between these critical systems. The consequences of this disconnect are severe and permeate every level of the university's operations, impacting all stakeholders. Consider the quintessential journey of a new student:

1. A student enrolls in a course via the academic Moodle instance. This single digital action should trigger a seamless chain of automated events.

2. Instead, it initiates a series of slow, brittle, and error-prone manual processes. An administrator must perform a manual data export from the academic system.

3. This data is then manually imported into the finance Moodle to generate a tuition invoice.

4. Concurrently, another staff member might need to add the student to the library's access control system, and yet another to the HR system if the student is also a campus employee.

This reliance on "human middleware" for system integration is not merely inefficient; it is a critical business risk. A simple data entry error, a misplaced file, or a delay in processing can lead to a student being incorrectly billed, denied access to essential course materials, or, in a worst-case scenario, being barred from a final exam due to a payment flag that was not cleared in time.

1.2.1 Impact on Stakeholders

The negative consequences of this fragmented system are felt across the institution:

- **For Students:** The experience is disjointed and frustrating. They are forced to navigate a maze of administrative bureaucracy to resolve issues that, in a properly integrated system, would not exist. This friction detracts from their primary focus: learning.
- **For Faculty:** There is no unified, real-time view of a student. It becomes difficult to ascertain a student's official status (e.g., "Is this student officially enrolled or have they dropped the course? Has their payment been processed?").
- **For Administrators:** It translates into thousands of wasted hours spent on repetitive data entry, complex manual reconciliation tasks, and correcting preventable errors. This is low-value work that prevents skilled staff from focusing on higher-impact activities.

1.2.2 Strategic Implications

Beyond the daily operational friction, this data fragmentation severely hampers the institution's ability to leverage its most valuable asset—its data—for strategic planning. Generating a simple, accurate, real-time report on the correlation between student enrollment, course demand, and revenue becomes a monumental task requiring data from multiple systems to be manually collated and cleaned. This inability to generate timely business intelligence puts the university at a competitive disadvantage.

This proposal outlines a plan to methodically dismantle these data silos and engineer a cohesive, integrated digital ecosystem that is resilient, scalable, and built for the future.

Chapter 2

Project Scope

This project is designed to methodically dismantle the aforementioned data silos and create a cohesive, integrated digital ecosystem. The scope is carefully defined to ensure focus on the most critical integration points while managing project complexity and risk. The objective is to deliver maximum value by targeting the core student lifecycle processes.

2.1 In-Scope Activities

Integration of Core University Systems The project will focus exclusively on integrating the three most critical Moodle instances that govern the student lifecycle: **Academics** (course management, enrollment, grades), **Finance** (billing, payments), and **Human Resources** (student employment, faculty data). This initial focus ensures the highest impact on operational efficiency and student experience.

Real-Time, Event-Driven Data Synchronization The primary technical objective is to enable real-time data synchronization for a defined set of key business processes. This includes, but is not limited to, new student registration, course enrollment, course withdrawal, grade submission, fee payment confirmation, and updates to student personal information. For the purposes of this project, "real-time" is defined as a sub-second propagation time for critical events across the integrated system.

Development of a Central Communication Backbone We will design and build the "central nervous system" for the university's digital operations. This backbone will consist of an API Gateway for managing and securing external traffic, a service mesh for secure and observable inter-service communication, and a robust event bus to handle all asynchronous data exchange.

Deployment on a Cloud-Native Infrastructure The entire integrated solution will be architected and deployed on a modern, scalable cloud infrastructure (e.g., Amazon Web Services (AWS), Google Cloud Platform (GCP), or Microsoft Azure). This is a non-negotiable requirement for achieving the required levels of elasticity, resilience, and maintainability that are impossible with on-premise, monolithic deployments.

Architectural Refactoring of Monolith Functions The project involves a strategic refactoring of the existing Moodle monoliths. It is crucial to understand that this is **not a complete rewrite**. Instead, we will identify key, well-defined business capabilities (bounded contexts in Domain-Driven Design) within Moodle, such as "Course Roster Management" or "Invoice Generation," and expose them via secure, internal RESTful APIs. The new microservices will then interact with these APIs, allowing for gradual modernization without the immense risk and cost of a "big bang" replacement.

2.2 Out-of-Scope Activities

To maintain focus and ensure timely project delivery, the following activities are explicitly excluded from the scope of this project:

Development of New End-User Features in Moodle This project's mandate is strictly focused on the backend integration layer. The development of new features within the Moodle UIs (e.g., a new forum module, enhanced quiz functionality) is explicitly out of scope. This prevents "scope creep" and ensures the core integration goals are met on time and within budget.

Large-Scale Data Migration from Legacy Systems The project assumes the existing Moodle databases are the current sources of truth for their respective domains. It will not involve migrating historical data from other, non-Moodle legacy systems (e.g., old SIS platforms). The focus is squarely on the integration of live, operational processes moving forward.

Complete Replacement of the Moodle Platform We are augmenting, integrating, and modernizing the existing Moodle ecosystem, not replacing it. A "rip and replace" strategy for a system as embedded as an LMS would be a multi-year, high-risk, and prohibitively expensive endeavor. Our approach delivers significant value at a fraction of the cost and risk by building on the existing investment in the Moodle platform.

Chapter 3

Proposed Solution: A Distributed Systems Approach

To address the profound challenges of scalability, resilience, and inter-system collaboration inherent in the current architecture, we propose a solution architected on the principles of modern distributed systems. This paradigm shift moves away from a fragile, centralized monolithic application towards a resilient and flexible collection of small, independent services that communicate over a network. This approach is not merely a technical choice; it is a strategic decision that enables future agility and growth.

3.1 Scalability & Performance: Achieving True Elasticity

A distributed architecture enables **horizontal scaling**, a concept far superior to the traditional vertical scaling of monoliths.

- **Vertical Scaling (The Monolith Way):** This involves increasing the resources (CPU, RAM) of a single server. It is analogous to training one cashier to be marginally faster. This approach is expensive, has a hard physical limit, and involves downtime for upgrades.
- **Horizontal Scaling (The Distributed Way):** This involves adding more server instances to the pool as needed. It is analogous to opening more checkout lanes during peak hours. This approach is cost-effective, virtually limitless, and can be done with zero downtime.

Using a container orchestrator like Kubernetes, we can configure a **Horizontal Pod Autoscaler (HPA)** to automatically increase the number of ‘Enrollment‘ and ‘Billing‘ service instances during peak registration periods and then scale them back down during quiet periods to optimize costs. This elastic scaling ensures that the system remains responsive under any load.

Performance will be further enhanced through distributed caching strategies. A solution like Redis, an in-memory data store, can be used to cache frequently accessed, read-heavy data, such as course catalog information or user session data. This dramatically reduces the load on the primary PostgreSQL databases and lowers latency for end-users, resulting in a faster, more responsive experience.

3.2 Fault Tolerance & Resilience: Designing for Failure

A core principle of distributed systems is to expect and design for failure. By distributing services across multiple servers and even multiple physical data centers (known as Availability Zones in cloud terminology), we eliminate single points of failure. The failure of a single component will not bring down the entire system. We will design for failure by implementing several key patterns:

- **Database Resilience:** We will achieve high availability for our data tier through a multi-AZ PostgreSQL configuration. This setup involves a primary "master" database that handles writes and multiple "replica" servers in different physical locations that handle reads and can be promoted to master in case of a failure. An automated failover process will promote a replica to become the new master in under a minute with zero data loss (Recovery Point Objective of zero).
- **The Circuit Breaker Pattern:** At the application layer, we will implement this crucial pattern. For example, if the ‘Grades Service‘ tries to call a slow or unresponsive ‘Notification Service‘, the circuit breaker will "trip" after a few failed attempts. This prevents the ‘Grades Service‘ from being bogged down by retries, consuming resources, and potentially failing itself. Instead, it allows the service to fail gracefully, perhaps by queueing the notification to be sent later. This pattern prevents localized failures from causing a catastrophic system-wide cascade.

3.3 Decoupling & Collaboration: Enabling Organizational Agility

The cornerstone of the proposed solution is enabling seamless collaboration through **asynchronous, event-driven communication**. This design adheres to the "Smart Endpoints and Dumb Pipes" philosophy: the business logic and intelligence reside within the microservices themselves, not within the communication bus.

1. When a significant business event occurs (e.g., a student enrolls in a course), the responsible service (the ‘Enrollment Service‘) publishes a message to a central event bus (like RabbitMQ). The message contains data about the event, such as ‘"studentId": "123", "courseId": "CS101"‘.
2. Other services can then subscribe to these events and react accordingly. The ‘Billing Service‘ subscribes to ‘ENROLLMENT$_{C}REATED‘eventstogenerateaninvoice.The‘CourseManagementSer$

This completely decouples the systems; the ‘Academic Moodle‘ (via the ‘Enrollment Service‘) does not need to know anything about the internal workings of the ‘Finance Moodle‘ (via the ‘Billing Service‘). This loose coupling makes the entire ecosystem more flexible, maintainable, and adaptable to future changes. If a new ‘Alumni Relations‘ system is added in the future, it can simply subscribe to the ‘STUDENT$_{G}RADUATED‘eventwithoutrequiringanychangestotheexistingservices.Thisarchitectur$

Chapter 4

System Design and Architecture

The system will be meticulously designed as a cloud-native, microservices-based platform, prioritizing separation of concerns, data integrity, and clear communication patterns. The architecture is based on industry best practices for building scalable and resilient distributed systems.

4.1 Architectural Model: Microservices and API Gateway

The design is centered around a **microservices architecture**. Each core business capability is encapsulated as a small, autonomous service. These services are independently deployable, scalable, and maintainable.

An **API Gateway** (e.g., Kong) will serve as the single, managed entry point for all external requests originating from the Moodle frontends or other clients. This provides a critical layer for:

- **Security:** Authentication and authorization will be centralized at the gateway. It will validate JSON Web Tokens (JWTs) on all incoming requests.
- **Rate Limiting & Throttling:** Protects backend services from being overwhelmed by traffic spikes or malicious actors.
- **Routing:** Intelligently routes requests to the appropriate backend microservice (e.g., '/api/courses/*' goes to the 'Course Management Service').
- **Decoupling:** Clients interact with a stable gateway endpoint, while the internal service architecture can be refactored or updated without affecting the clients.

4.1.1 Core Microservices Responsibilities

The following is an initial breakdown of the core microservices to be developed:

Authentication Service: A centralized service for user identity and access control. It will handle user logins, issue and validate JWTs, and manage role-based access control (RBAC) policies. It will serve as the single source of truth for user identity across the ecosystem.

Course Management Service: Manages the lifecycle of courses, including creation, scheduling, and cataloging. It is responsible for maintaining the official course list and class rosters. Its API will provide endpoints like 'GET /courses' and 'GET /courses/id/roster'.

Enrollment Service: Orchestrates the complex business logic of student enrollment. This includes checking for prerequisites, managing class capacity and waitlists, and handling course withdrawals. It is the primary service that initiates the distributed transaction for enrollment.

Billing Service: Handles all financial transactions. It will calculate tuition fees based on enrollment data, generate invoices, and integrate with a third-party payment gateway (e.g., Stripe, PayPal) to process payments. It listens for enrollment events to trigger billing processes.

Grades Service: Manages the submission, storage, and calculation of student grades. It will handle the logic for calculating Grade Point Averages (GPAs) and provide the necessary data for generating official transcripts.

Notification Service: A centralized service for dispatching communications to users. It will support multiple channels (Email, SMS, Push Notifications) and use a templating engine for consistent messaging. Other services will publish events like 'INVOICE$_{G}ENERATED$'or'$GRADE_{P}OSTED$', $an

4.2 Data Model Distributed Transactions

Each microservice will adhere to the **"database per service"** pattern. This means each service will own its own private PostgreSQL database and will be the sole writer to that database. This is essential for achieving true service autonomy and loose coupling.

However, this introduces the challenge of maintaining data consistency across services for business processes that span multiple domains (e.g., enrollment requires creating records in both the Enrollment and Billing systems). To manage these distributed transactions, we will implement the **Saga pattern** using a choreography approach.

A saga is a sequence of local transactions where each transaction updates the database in a single service and then publishes an event to trigger the next local transaction in the sequence.

4.2.1 Example: Student Enrollment Saga

The enrollment process perfectly illustrates the saga pattern in action:

1. **Initiation:** A user's request to enroll hits the 'Enrollment Service'. It begins the 'CreateEnrollment' saga by creating an enrollment record in its local database with a 'PENDING' status.

2. **Event Publication:** It then publishes an 'ENROLLMENT$_{C}REATED$'$eventtotheRabbitMQmessagebro

2. **Billing Service Reaction:** The 'Billing Service', which subscribes to this event, consumes the message. It performs its local transaction: creating a corresponding invoice in its own database.

3. **Success Event:** Upon successful creation of the invoice, the 'Billing Service' publishes a 'BILLING$_{S}UCCESSFUL$'$event$.

3. **Confirmation:** The 'Enrollment Service' consumes the 'BILLING$_{S}UCCESSFUL$'$eventandcompletest

3. **Failure and Compensation:** If the 'Billing Service' fails for any reason (e.g., invalid student financial data), it would instead publish a 'BILLING$_{F}AILED$'$event.The$'$EnrollmentService$'$wouldconsu

4.3 Communication Model

The system will employ a hybrid communication model, defaulting to asynchronous communication for maximum resilience and scalability.

Asynchronous (Default): The vast majority of inter-service communication will be asynchronous via the RabbitMQ message broker. This publish-subscribe model is resilient to transient failures and highly scalable, as services do not need to be available simultaneously to communicate and do not block while waiting for a response. This is the preferred method for all event-based collaboration.

Synchronous (By Exception): Synchronous communication (direct REST API calls over HTTP) will be used sparingly and only when an immediate response is absolutely required. For example, the API Gateway will make a synchronous call to the ‘Authentication Service‘ during a user login process because it must wait for the validation response before proceeding. To manage the complexity of this internal "east-west" traffic, a **Service Mesh** like Istio or Linkerd will be implemented. A service mesh provides advanced features like traffic management (retries, timeouts), mutual TLS encryption for security, and detailed observability (metrics, tracing) without requiring any changes to the application code itself.

Chapter 5

Technology Stack

The technology stack for this project has been carefully selected to build a high-performance, scalable, and maintainable system. The choices prioritize best-in-class open-source tools and managed cloud services to accelerate development and reduce operational overhead.

Category	**Technology**	**In-Depth Justification**
Language/Framework	Python/FastAPI	Python's rich ecosystem, readability, and extensive libraries (e.g., Py
Database	PostgreSQL	PostgreSQL is a battle-tested, open-source relational database renow
Messaging	RabbitMQ	As a mature and robust implementation of the Advanced Message Q
Containerization	Docker	Docker is the de facto industry standard for containerization. It allow
Orchestration	Kubernetes	Kubernetes is the leading container orchestration platform for autom
API Gateway	Kong	A robust and high-performance open-source API gateway is needed
Infrastructure	Terraform	Infrastructure as Code (IaC) is a critical practice for managing cloud

Chapter 6

Project Plan and Timeline

The project is estimated to take approximately six months and is broken down into distinct, manageable phases. The timeline is ambitious but achievable with a dedicated project team operating under an Agile framework.

6.1 Calendar of Activities

The project will be executed in seven distinct phases, from initial planning to post-launch support.

Phase	Granular Activities	Duration	Deadline
Phase 1: Research & Planning	- Conduct detailed stakeholder interviews with heads of Academics, Finance, and HR. - Perform user story mapping workshops with administrative staff to capture requirements. - Finalize detailed technical requirements and non-functional requirements (NFRs). - Perform a comprehensive risk assessment and create a mitigation plan (see Section 6.2).	1 Week	Oct 26, 2025
Phase 2: System Design & Architecture	- Create detailed C4 models (Context, Containers, Components) of the architecture. - Design OpenAPI 3.0 specifications for all microservice APIs (API-first design). - Model database schemas for each service and define inter-service event payloads. - Design the CI/CD pipeline architecture and a comprehensive observability strategy.	5 days	Oct 31, 2025
Phase 3: Core Infrastructure Setup	- Use Terraform scripts to provision the cloud VPC, subnets, and security groups. - Set up a managed Kubernetes cluster (e.g., AWS EKS). - Implement the CI/CD pipeline using Jenkins or GitLab CI. - Deploy a monitoring stack (Prometheus, Grafana) and a centralized logging solution (ELK Stack).	1 Week	Nov 7, 2025

Table 6.1 – continued from previous page

Phase	Granular Activities	Duration	Deadline
Phase 4: Microservice Development	- **Sprint 1-2:** Build Authentication Service & Course Management Service. - **Sprint 3-4:** Build Enrollment Service, implementing the 'CreateEnrollment' saga. - **Sprint 5-6:** Build Billing Service with payment gateway integration. - **Sprint 7-8:** Build Grades Service and Notification Service. (Development will follow a two-week sprint cycle)	8 Weeks	Jan 2, 2026
Phase 5: Integration & Testing	- Write and automate end-to-end (E2E) tests using a framework like Cypress to validate user journeys. - Conduct extensive performance and load testing using a tool like Locust or k6. - Execute chaos engineering experiments (e.g., terminating pods) to validate system resilience.	2 Weeks	Jan 16, 2026
Phase 6: UAT & Deployment	- Conduct formal User Acceptance Testing (UAT) with a pilot group of administrative staff. - Prepare and secure the production environment, including final security audits. - Execute a phased rollout using a Blue-Green deployment strategy to minimize risk and downtime.	1 Week	Jan 23, 2026
Phase 7: Post-Launch Support	- Actively monitor system health and performance dashboards. - Establish on-call rotations and create operational runbooks for incident response. - Provide training sessions and ongoing support for administrative staff.	Ongoing	-

Table 6.1: Project Timeline

6.2 Risk Management

A proactive risk management strategy is essential for project success. The following table identifies potential risks and their mitigation strategies.

Risk Category	Risk Description	Mitigation Strategy
Technical	Unforeseen complexity in refactoring and creating APIs for the legacy Moodle monoliths.	Allocate specific "spike" stories in early sprints to investigate Moodle's codebase. Adopt a strangler fig pattern to gradually replace functionality rather than a big bang refactor.
Project Management	Scope creep from stakeholders requesting new user-facing features.	Strictly enforce the "Out-of-Scope" definitions. Maintain a separate backlog for future feature requests to be addressed after this project is complete. Regular stakeholder communication to manage expectations.
Resource	Lack of available personnel with expertise in the selected cloud-native technology stack (Kubernetes, Terraform).	Invest in targeted training for the existing team. Engage with a certified cloud partner for initial setup and consultation. Prioritize hiring for key DevOps roles early in the project.
Security	A data breach in one of the microservices or databases.	Implement a "defense in depth" strategy: secure coding practices, regular vulnerability scanning, principle of least privilege for IAM roles, network segmentation, and encryption of data at rest and in transit.

Table 6.2: Risk Assessment and Mitigation Plan

Chapter 7

Implementation Strategy

The implementation phase will be guided by Agile principles, focusing on iterative development, continuous feedback, and a high degree of automation to ensure both quality and velocity.

7.1 Development Workflow Methodology

We will adopt the **Scrum framework** for Agile project management, operating in **two-week sprints**.

- **Sprint Ceremonies:** Each sprint will begin with a *Sprint Planning* meeting to select a set of user stories from the product backlog. Daily *Stand-ups* will ensure team alignment. Each sprint will conclude with a *Sprint Review* to demonstrate the completed work to stakeholders and a *Sprint Retrospective* for the team to reflect and improve its processes.
- **Source Control:** A **GitFlow** branching strategy will be used for source control in Git. This provides a robust model for managing feature development (‘feature/‘ branches), releases (‘release/‘ branches), and hotfixes (‘hotfix/‘ branches), ensuring the ‘main‘ branch is always stable and deployable.
- **Code Quality:** All code will be subject to mandatory peer review via pull requests. No code will be merged into the main development branch without at least one approval from another engineer. This maintains high code quality and facilitates crucial knowledge sharing within the team.

7.2 CI/CD (Continuous Integration/Continuous Deployment) Pipeline

Automation is the bedrock of this project’s implementation strategy. A comprehensive CI/CD pipeline will be established to automate the entire process from code commit to production deployment. This minimizes manual errors and enables rapid, reliable releases. The pipeline stages will include:

1. **Commit:** A developer pushes code to a feature branch in the Git repository.
2. **Static Analysis:** Automated tools (linters like Flake8, static security scanners like SonarQube) check the code for quality, style violations, and common vulnerabilities.
3. **Unit & Integration Tests:** A comprehensive suite of automated tests is run to validate the correctness of the new code in isolation and with its direct dependencies (e.g., the database). A high test coverage percentage will be enforced.
4. **Build:** A Docker image for the microservice is built.

5. **Publish:** The new, versioned Docker image is pushed to a private container registry (e.g., Amazon ECR or Google Artifact Registry).

6. **Deploy to Staging:** The new version is automatically deployed to a production-like staging environment.

7. **End-to-End Testing:** Automated E2E tests are run against the staging environment to validate that entire user workflows function correctly across multiple services.

8. **Manual Approval (Optional):** For production deployments, a final manual approval gate can be configured to allow for a last-minute sanity check by a senior engineer or project lead.

9. **Deploy to Production:** The change is rolled out to production using a **Blue-Green deployment** strategy. Traffic is shifted from the old version (Blue) to the new version (Green). This allows for instant, zero-downtime rollback by simply shifting traffic back to Blue if any issues are detected.

7.3 Observability Strategy

In a complex distributed system, understanding the internal state and performance is paramount for maintenance and debugging. We will implement the three pillars of observability:

Logging: All services will output structured JSON logs. These logs will be collected by an agent (like Fluentd) and forwarded to a centralized logging platform like the **ELK Stack (Elasticsearch, Logstash, Kibana)**. This allows for powerful, searchable aggregation of log data from across the entire system, making it possible to trace a single user request through multiple services.

Metrics: We will use **Prometheus** to scrape time-series metrics from all services and infrastructure components. Key application metrics (request rates, error rates, latencies—the RED method) and system metrics (CPU, memory, disk usage) will be collected. These metrics will be visualized in **Grafana dashboards** to provide a real-time, at-a-glance view of system health and to configure automated alerting for anomalies.

Tracing: Distributed tracing, using an open standard like **OpenTelemetry** and a backend like **Jaeger** or Zipkin, will be implemented. This allows us to trace the lifecycle of a single request as it flows through multiple services. This is indispensable for debugging performance bottlenecks and understanding complex service interactions, providing a visual representation of the entire call graph for a given transaction.

Chapter 8

Evaluation and Success Metrics

The success of this project will not be measured merely by its completion, but by a set of clear, quantifiable metrics that span both technical performance and tangible business impact.

8.1 Success Metrics & Key Performance Indicators (KPIs)

We will track two categories of KPIs to evaluate the project's effectiveness.

8.1.1 Technical KPIs

These metrics will be continuously monitored through our observability stack (Prometheus/Grafana).

8.1.2 Business KPIs

These metrics measure the direct impact of the project on university operations.

8.2 Testing Strategy

A multi-layered testing strategy is essential for ensuring the quality and reliability of the system. We will employ the "testing pyramid" model:

Unit Tests: These form the base of the pyramid. Each individual function and class will be covered by unit tests (using frameworks like ‘pytest‘) to ensure its correctness in isolation. They are fast, cheap to write, and provide rapid feedback to developers.

Integration Tests: These tests verify the interactions between a service and its external dependencies, like its database or other services it calls directly. We will write tests that confirm a service can correctly write to and read from its PostgreSQL database.

Contract Tests: To manage dependencies in a microservices world, we will use a framework like **Pact**. Pact allows a "consumer" service to define a contract for how it will interact with a "provider" service's API. This ensures that as services evolve, they do not introduce breaking changes that would affect their dependents.

End-to-End (E2E) Tests: At the top of the pyramid, these automated tests will simulate real user workflows from the perspective of the end-user (e.g., logging in, enrolling in a course, and verifying the invoice). We will use a tool like Cypress to automate these tests against our staging environment.

KPI	Target	Measurement Method
System Availability	99.95% up-time	Measured by Prometheus blackbox probing of critical API endpoints.
API Latency	95th per-centile re-sponse time < 200ms	Measured from API Gateway and service mesh metrics.
Error Rate	Production er-ror rate < 0.1%	Measured from API Gateway and service logs (HTTP 5xx responses).
Deployment Frequency	Daily de-ploy-ments	Measured by CI/CD pipeline execution logs.
Lead Time for Changes	< 1 hour	Measured as the time from code commit to successful production deployment.

Table 8.1: Technical Key Performance Indicators

Performance & Load Tests: We will use a tool like Locust to simulate thousands of concurrent users to identify performance bottlenecks and ensure the system can handle peak loads, such as the first day of registration.

8.3 User Acceptance Testing (UAT) Plan

Before the final production rollout, a formal UAT phase will be conducted with a selected group of administrative staff who represent the end-users of the system's data.

- **Scenarios:** The UAT group will be provided with a set of real-world test scenarios (e.g., "Enroll a new student who is also a campus employee," "Process a tuition refund for a withdrawn course").
- **Environment:** They will execute these scenarios in the staging environment, which is a production replica.
- **Feedback:** A structured feedback mechanism (e.g., a dedicated Jira board) will be used to collect and prioritize any issues or suggestions.

KPI	Target	Measurement Method
Reduction in Manual Data Entry	80% reduction in person-hours	Pre- and post-implementation surveys and interviews with administrative
Decrease in Support Tickets	50% reduction in tickets	Analysis of help desk ticketing system data for categories related to enro
Time-to-Report Generation	From days to seconds	Timed execution of generating a comprehensive enrollment and revenue

Table 8.2: Business Key Performance Indicators

- **Exit Criteria:** The UAT phase will be considered complete only when all critical and major bugs have been resolved and the key stakeholders formally sign off on the system's readiness.

Chapter 9

Conclusion

The current fragmented Moodle ecosystem, characterized by entrenched data silos and inefficient manual processes, represents a significant operational bottleneck and a fundamental barrier to providing a modern, seamless experience for our students and staff. This system, a product of organic growth rather than intentional design, accrues technical debt daily, stifles innovation, and puts the institution at a competitive disadvantage in an increasingly digital higher education landscape.

This proposal outlines a strategic and transformative initiative to engineer a unified, resilient, and scalable cloud-native platform. By adopting a distributed, microservices-based architecture, we will not only solve the immediate and pressing problems of data inconsistency and operational inefficiency but also lay a robust and flexible foundation for future innovation. The proposed solution, built on a modern, open-source technology stack, will empower the institution to become more agile, data-driven, and responsive to the evolving needs of the academic community.

This project is more than a technical upgrade; it is a critical investment in the digital transformation of the institution. The successful implementation of this proposal will yield substantial and measurable returns in operational efficiency, student and staff satisfaction, and long-term strategic flexibility. By dismantling our data silos and creating a cohesive digital nervous system, we will unlock the true value of our data, enabling real-time insights and data-informed decision-making. We are confident that this initiative will position the university as a leader in leveraging technology to enhance the educational experience and secure its success for years to come.