9.1 Testing Strategy

The Study in Woods project implements a comprehensive testing strategy encompassing unit tests, integration tests, end-to-end tests, security tests, and performance tests. Testing occurs continuously throughout development via automated CI/CD pipeline, with all tests executing on every pull request before merge approval. The testing pyramid approach prioritizes unit tests (70% of tests) for fast feedback, integration tests (20%) for component interaction verification, and end-to-end tests (10%) for critical user journeys. Test coverage targets minimum 70% for backend services and 60% for frontend components, measured automatically and reported on each commit.

9.2 Unit Testing

9.2.1 Backend Unit Tests

Go backend tests use the standard testing package with testify/assert library for readable assertions. Critical service layer functions maintain 80%+ coverage. Tests employ table-driven testing pattern enabling comprehensive input coverage with minimal code duplication. Mock interfaces replace external dependencies (database, AI API, storage) using testify/mock package. Test fixtures provide reusable test data loaded from JSON files. Example test cases include password hashing verification (bcrypt cost factor 12), JWT token generation and validation (RS256 signing, expiration checks), syllabus extraction parsing (valid JSON, missing fields, invalid structure), and document upload validation (file type, size limits, duplicate detection).

Test Execution: go test -v -cover ./...

Coverage Target: 70% minimum, 80% for critical services

Test Count: 156 unit tests covering services, handlers, utilities

9.2.2 Frontend Unit Tests

Frontend tests use Jest testing framework with React Testing Library for component testing. Component tests verify rendering with different props, user interactions (clicks, typing, form submission), state updates and re-rendering, error handling and error boundaries, and accessibility features (ARIA labels, keyboard navigation). Hook tests validate custom React hooks including useAuth (login, logout, token refresh), useChat (message sending, streaming responses), and useUpload (file selection, progress tracking, error handling).

Test Execution: npm test -- --coverage

Coverage Target: 60% minimum for components

Test Count: 89 component and hook tests

9.3 Integration Testing

9.3.1 API Integration Tests

Integration tests verify interactions between API handlers, services, database, and external services using real PostgreSQL and Redis instances launched via Docker Compose. Tests execute within database transactions that rollback after completion ensuring test isolation. Test suite covers authentication flows (register → login → access protected route → logout), academic hierarchy CRUD (create university → course → semester → subject → verify relationships), document workflow (upload → validate → save metadata → extract syllabus → index to KB), and chat interactions (create session → send message → verify AI response → check citations stored).

Test Suite	Test Cases	Coverage Area
Authentication	12	Register, Login, JWT validation,

Test Suite	Test Cases	Coverage Area
		Password reset
Universities	8	CRUD operations, Authorization checks
Courses & Subjects	15	Hierarchical relationships, Cascade deletes
Documents	10	Upload, Validation, Storage, Retrieval
Syllabus	18	Extraction, Parsing, Storage, Retrieval
Chat	14	Sessions, Messages, Streaming, Citations
Admin	9	User management, Settings, Audit logs

9.3.2 Database Integration Tests

Database tests verify GORM model relationships, constraints, and migrations. Tests cover foreign key cascade deletes (deleting university cascades to courses, semesters, subjects), unique constraints (duplicate email prevents user creation), JSONB operations (citations array storage and retrieval), and migration idempotency (running migration twice produces same schema).

9.4 End-to-End Testing

9.4.1 Critical User Journeys

End-to-end tests use Playwright to automate browser interactions simulating real user workflows. Tests launch full application stack (frontend, backend, database, Redis) in isolated Docker environment. Critical journeys tested include complete onboarding (register account →

verify email → login → create university → enroll in course), document processing (navigate to subject → upload PDF → wait for extraction → verify syllabus displayed), AI interaction (open chat → send question → verify streaming response → check citations → verify message saved), and admin operations (login as admin → view users → change role → verify permission change).

User Journey	Steps	Duration	Pass Rate
User Registration & Login	7	15s	98%
Course Enrollment	10	22s	96%
Document Upload & Processing	12	45s	94%
AI Chat Interaction	8	18s	97%
Admin User Management	9	20s	99%

Test Execution: Scheduled nightly + on PR to main

Environment: Isolated Docker stack with test database

Reporting: Screenshots and videos for failures

9.5 Security Testing

9.5.1 Authentication & Authorization Tests

Security tests verify authentication mechanisms and authorization enforcement. Test cases include invalid token rejection (expired, malformed, wrong signature), password security (minimum length, complexity requirements, hashing verification), brute force protection (rate limiting after 5 failed attempts), and authorization checks (students cannot access admin routes, users cannot access other users' data).

9.5.2 Input Validation Tests

Validation tests attempt malicious inputs including SQL injection payloads ('; DROP TABLE users; --), XSS attacks (), path traversal (../../etc/passwd), and oversized inputs (files exceeding 10MB, strings exceeding max length). All tests verify proper sanitization and rejection with appropriate error messages.

9.5.3 Dependency Scanning

Automated dependency scanning runs weekly via GitHub Actions using npm audit for Node.js packages and go mod verify for Go modules. High and critical severity vulnerabilities block deployment until patched. Dependency updates automated via Dependabot with automated testing before auto-merge.

9.6 Performance Testing

9.6.1 Load Testing

Load tests simulate concurrent users to verify system performance under stress. Tests use k6 load testing tool to generate traffic patterns. Baseline test simulates 100 concurrent users making varied requests (login, browse subjects, upload documents, chat) over 5-minute duration. Spike test suddenly increases load from 50 to 500 users testing auto-scaling. Soak test maintains 200 concurrent users for 30 minutes detecting memory leaks and resource exhaustion.

Test Type	Virtual Users	Duration	Success Criteria
Baseline	100	5 min	95% requests < 2s, 0% errors
Spike	50→500	10 min	No crashes, degradation acceptable

Test Type	Virtual Users	Duration	Success Criteria
Soak	200	30 min	Stable memory, no degradation
Stress	100→1000	15 min	Identify breaking point

9.6.2 API Response Time Tests

Performance benchmarks verify API endpoints meet response time requirements. Benchmarks execute 1000 requests per endpoint measuring p50, p95, p99 latencies. Target metrics include authentication (login < 500ms p95), subject list (< 200ms p95), document upload initiation (< 1s p95), chat message send (< 500ms p95 excluding AI processing), and AI streaming start (< 2s to first token p95).

9.6.3 Database Query Performance

Query performance tests measure database query execution times using EXPLAIN ANALYZE. Tests identify slow queries (> 500ms), missing indexes, and N+1 query problems. Optimizations include adding indexes on frequently queried columns, using GORM preloading to eliminate N+1 queries, and implementing Redis caching for frequently accessed data.

9.7 Test Automation & CI/CD

9.7.1 Continuous Integration Pipeline

GitHub Actions workflow executes automated tests on every push and pull request. Pipeline stages run in parallel for speed: Lint stage (golangci-lint, ESLint) - 2 minutes, Unit test stage (Go + JavaScript) - 5 minutes, Integration test stage (API tests with Docker services) - 8 minutes, and Build verification stage (Docker image build) - 4 minutes. Total pipeline duration

approximately 10 minutes. Failed checks block PR merge maintaining code quality.

9.7.2 Test Coverage Reporting

Coverage reports generated automatically on each test run using go test -cover and Jest --coverage. Reports uploaded to Codecov providing coverage trends visualization, pull request coverage diffs, and coverage badges for README. Coverage requirements enforced: no PR decreases overall coverage, new code must have minimum 70% coverage.

9.7.3 Test Data Management

Test data managed through seeding scripts creating consistent test datasets. Seed data includes 3 universities, 10 courses, 40 subjects, 5 test users (various roles), sample documents, and pre-extracted syllabus data. Database reset script drops all tables and re-seeds ensuring clean state. Docker Compose test configuration uses separate database volumes preventing test data pollution of development database.

9.8 Test Metrics & Results

Metric	Target	Current	Status
Unit Test Coverage	70%	76%	✓ Pass
Integration Test Coverage	60%	68%	✓ Pass
E2E Test Coverage (Critical Paths)	100%	100%	✓ Pass
API Response Time (p95)	< 2s	1.2s	✓ Pass
Zero Security Vulnerabilities (High/Critical)	0	0	✓ Pass
Test Execution Time (CI)	< 15 min	10 min	✓ Pass

Metric	Target	Current	Status
Test Pass Rate	> 95%	97.8%	✓ Pass