5.1 Development Methodology Overview

The Study in Woods project follows an Agile software development methodology, specifically implementing a hybrid approach combining elements of Scrum for sprint management and Kanban for continuous flow of features. This methodology was chosen to enable rapid iteration, frequent stakeholder feedback, and the ability to adapt to changing requirements during the academic project timeline. The iterative nature of Agile aligns perfectly with educational software development where user needs emerge through testing and feedback rather than comprehensive upfront specification.

The development process spans 12 major phases executed over a 6-month period from June 2024 to December 2024. Each phase represents a major functional milestone delivering working software increments that can be demonstrated and tested. Phases are not strictly sequential; rather, they overlap with continuous integration and testing running throughout the development lifecycle. This approach enabled early detection of integration issues and maintained a deployable application state at all times.

5.2 Agile Implementation Framework

5.2.1 Sprint Structure

Development work is organized into two-week sprints, each beginning with sprint planning and ending with sprint review and retrospective. Sprint planning sessions allocate user stories to the sprint backlog based on priority (MoSCoW method: Must have, Should have, Could have, Won't have) and estimated effort (story points using Fibonacci sequence). The team commits to completing 20-25 story points per sprint, calculated based on historical velocity metrics.

Daily standup meetings (15 minutes maximum) maintain team alignment through three questions: What did I accomplish yesterday? What will I work on today? What blockers am I facing? These standups occur asynchronously via project management tools given the singledeveloper context, with blockers documented in a dedicated channel for resolution. Sprint reviews demonstrate completed features to stakeholders (project guide, peers), gathering feedback for backlog refinement. Sprint retrospectives identify process improvements, with action items tracked and implemented in subsequent sprints.

5.2.2 User Story Driven Development

Requirements are expressed as user stories following the template: "As a [user role], I want [feature] so that [benefit]." Each story includes acceptance criteria defining "done" in testable terms. For example: "As a student, I want to upload my syllabus PDF so that I can chat with AI about course topics. Acceptance Criteria: (1) Upload form accepts PDF files up to 10MB, (2) Progress bar shows upload percentage, (3) Success message displays upon completion, (4) Document appears in subject's document list within 30 seconds."

Stories are decomposed into technical tasks assigned to specific architecture layers (frontend, backend API, database, AI integration). Task breakdowns ensure no single task exceeds 4 hours of work, enabling completion within a single development session. Dependencies between tasks are mapped in a task graph, with parallel tracks for frontend and backend development converging at integration points.

5.2.3 Continuous Integration and Deployment

The CI/CD pipeline automates code quality checks, testing, and deployment through GitHub Actions workflows triggered on every git push. The continuous integration workflow executes in parallel stages: (1) Linting stage runs golangci-lint for Go code and ESLint for TypeScript, failing the build on style violations or potential bugs, (2) Unit testing stage executes go test with coverage reporting, requiring minimum 70% coverage for critical services, (3) Integration testing stage launches Docker Compose stack and runs API tests verifying end-to-end scenarios, (4) Build verification stage compiles production Docker images ensuring no build-time errors.

The continuous deployment workflow activates on merges to the main branch, executing zero-downtime deployment to production environment. The workflow builds optimized Docker images with multi-stage compilation, pushes images to DigitalOcean Container Registry with semantic version tags (v1.2.3), SSHs into production droplets, pulls new images, starts new containers with health checks, waits for health check success, drains connections from old containers, stops old containers, and posts deployment notifications to monitoring channels. Automatic rollback triggers if health checks fail after 60 seconds.

Sprint Activity	**Frequency**	**Duration**	**Participants**	**Deliverables**
Sprint Planning	Every 2 weeks	2 hours	Developer, Guide	Sprint backlog, Story estimates
Daily Standup	Daily	15 minutes	Developer	Progress updates, Blocker list
Sprint Review	Every 2 weeks	1 hour	Developer, Guide, Peers	Working software demo
Sprint Retrospective	Every 2 weeks	1 hour	Developer, Guide	Process improvement action items
Backlog Refinement	Weekly	1 hour	Developer	Refined user stories, Updated priorities

5.3 Development Phases

5.3.1 Phase 1-3: Foundation (Weeks 1-6)

Phase 1: Project Setup & Infrastructure (Week 1-2)

Initialized project repository with monorepo structure (apps/api, apps/web) using Turbo for build orchestration. Set up development environment with Docker Compose defining services for PostgreSQL, Redis, backend API, and frontend. Created initial database schema with GORM models for User, University, Course, Semester, Subject tables. Configured environment variables for database connections, JWT secrets, and API keys. Established Git workflow with main and develop branches, pull request templates, and branch protection rules.

Phase 2: Authentication System (Week 3-4)

Implemented complete authentication system including user registration with email validation and password strength requirements, login endpoint generating JWT access tokens (24-hour expiration) and refresh tokens (7-day expiration), password reset flow with email-based token verification, JWT middleware validating tokens on protected routes, and token blacklist mechanism for logout functionality. Integrated bcrypt password hashing with cost factor 12, added Redis-based brute force protection limiting failed login attempts to 5 per 15 minutes, and created frontend authentication forms with validation using React Hook Form and Zod schemas.

Phase 3: Academic Hierarchy (Week 5-6)

Built CRUD endpoints for Universities (create, read, update, delete with soft delete support), Courses (with parent University association), and Subjects (with parent Semester association). Implemented automatic semester generation based on course duration configuration. Created hierarchical navigation UI showing University → Course → Semester → Subject drill-down. Added search and filtering capabilities on each entity with debounced search input. Implemented pagination with limit/offset parameters for large result sets.

5.3.2 Phase 4-6: Core Features (Weeks 7-12)

Phase 4: Document Upload System (Week 7-8)

Integrated DigitalOcean Spaces for document storage using AWS S3-compatible SDK. Implemented multipart upload for files larger than 5MB with 5MB chunk size. Created upload endpoint validating file type (PDF only), size (max 10MB), and generating unique filenames (UUID + original name). Built frontend upload interface with drag-and-drop zone using react-dropzone, progress bar showing upload percentage, and upload queue supporting multiple concurrent uploads. Added document list view displaying filename, upload date, size, and download button generating pre-signed URLs with 1-hour expiration.

Phase 5: AI Knowledge Base Integration (Week 9-10)

Integrated DigitalOcean GradientAI Knowledge Base API for document indexing. Implemented subject-specific knowledge base creation on first document upload, document indexing workflow uploading files to knowledge base and polling for completion status, and indexing status tracking (pending, in_progress, completed, failed) stored in Document table. Created background job using robfig/cron checking for pending documents every 5 minutes and initiating indexing. Built progress tracking system storing extraction state in Redis and streaming updates to frontend via Server-Sent Events (SSE).

Phase 6: Syllabus Extraction (Week 11-12)

Developed AI-powered syllabus extraction using Llama 3.3 70B with structured output prompting. Created extraction prompt template requesting JSON output with units array containing unit number, title, topics array (topic name, subtopics, teaching hours), and learning outcomes. Implemented chunked extraction for large syllabi (>8000 tokens) splitting into unit ranges and merging results. Added extraction validation checking for required fields, data type correctness, and logical consistency (total hours > 0, unit numbers sequential). Built syllabus display UI showing units as accordion sections, topics as nested lists, and edit capability for manual corrections. Stored extracted syllabus as JSONB in Subject table enabling fast queries on syllabus content.

5.3.3 Phase 7-9: AI Chat Features (Weeks 13-18)

Phase 7: Chat Session Management (Week 13-14)

Created chat session model storing session ID, subject ID, user ID, title (auto-generated from first message), created timestamp, and last message timestamp. Implemented session CRUD operations including create new session, list user's sessions sorted by last message timestamp, get session details with message history, update session title, and delete session with cascade to messages. Built session UI with sidebar showing session list, new session button, session deletion with confirmation dialog, and automatic session title generation ("Chat about Subject Name - Date").

Phase 8: AI Chat Implementation (Week 15-16)

Integrated DigitalOcean GradientAI Chat Completion API with knowledge base context. Implemented streaming response handling using Server-Sent Events (SSE), message persistence storing user prompt and AI response in ChatMessage table, context assembly including last 10 messages for conversation continuity, and token usage tracking for cost monitoring. Created chat UI with message list showing user messages (right-aligned) and AI responses (left-aligned), markdown rendering for formatted responses using react-markdown, streaming text display animating token-by-token appearance, typing indicator during AI processing, and error handling with retry capability.

Phase 9: Citation Support (Week 17-18)

Added citation extraction from AI responses capturing document references included in knowledge base responses. Implemented citation storage as JSONB array containing citation ID, page content snippet, relevance score, filename, and data source ID. Created citation display UI showing citation cards below AI messages, click-to-expand functionality revealing full citation text, and links to source documents. Enhanced response quality by increasing knowledge base retrieval count to top 5 documents (previously 3) and adjusting retrieval threshold to cosine similarity > 0.7.

5.3.4 Phase 10-12: Advanced Features (Weeks 19-24)

Phase 10: PYQ Extraction (Week 19-20)

Implemented AI-powered question extraction from exam paper PDFs using structured prompts requesting question number, question text, marks allocation, question type (MCQ, short answer, long answer, numerical), and estimated difficulty level. Created PYQ model storing question details, linking to parent subject and syllabus unit/topic, and extraction metadata (source document, page number, extraction confidence). Built PYQ display UI grouping questions by unit, filtering by difficulty and type, and search functionality on question text. Added manual categorization interface enabling users to link extracted questions to correct syllabus topics when AI categorization is incorrect.

Phase 11: Analytics Dashboard (Week 21-22)

Developed comprehensive analytics tracking user activity (logins, document uploads, chat messages, subject accesses), API usage (endpoint calls, response times, error rates), and system health (database connections, Redis memory, API response times). Implemented activity logging middleware capturing request details (user ID, endpoint, timestamp, response status, duration) and storing in UserActivity table. Created analytics aggregation queries calculating daily/weekly/monthly statistics using PostgreSQL date_trunc function and GROUP BY clauses. Built analytics dashboard UI with time-series charts showing usage trends using Recharts library, statistics cards displaying key metrics (total users, documents, chat messages), and user leaderboard ranking by activity level.

Phase 12: Admin Panel & Deployment (Week 23-24)

Created admin-only routes protected by role-based authorization middleware checking user.role === 'admin'. Implemented admin features including user management (list, search, edit role, reset password, delete), system settings (API key configuration, feature flags, rate limits), audit log viewer showing admin actions with timestamps and details, and database statistics (table row counts, storage usage). Built production deployment pipeline with multi-stage Docker builds, DigitalOcean Droplet provisioning using Terraform, automated database migrations on deployment, SSL certificate configuration using Let's Encrypt, and health monitoring with automatic alerting on failures.

Phase	Weeks	Key Deliverables	Lines of Code
1-3: Foundation	1-6	Authentication, Academic Hierarchy	~3,500
4-6: Core Features	7-12	Document Upload, AI Integration, Syllabus Extraction	~4,200
7-9: AI Chat	13-18	Chat Sessions, AI Chat, Citations	~3,800
10-12: Advanced	19-24	PYQ Extraction, Analytics, Admin Panel	~3,000

5.4 Testing Strategy

5.4.1 Unit Testing

Unit tests validate individual functions and methods in isolation using the standard Go testing package (testing) and testify assertion library. Critical business logic in services layer maintains minimum 80% code coverage measured by go test -cover. Test files follow naming convention *_test.go and are co-located with source files. Tests use table-driven testing pattern for comprehensive input coverage, with each test case defining input parameters, expected output, and error expectations.

Mocking strategies include interface-based mocking for database repositories (implementing repository interfaces with in-memory storage), HTTP mocking for external API calls (using httptest package to record and replay responses), and time mocking for datedependent logic (injecting time.Now() as dependency). Test fixtures provide reusable test data including sample users, documents, and chat messages loaded from JSON files. Continuous coverage tracking reports coverage percentage on each pull request, blocking merges that reduce overall coverage.

5.4.2 Integration Testing

Integration tests verify interactions between system components using real database and Redis instances launched via Docker Compose test configuration. Test suite setup scripts create test database schema, seed initial data (test user accounts, sample universities/courses), and configure test environment variables. Each test runs in a transaction that rolls back after completion, ensuring test isolation and repeatability.

API integration tests use Fiber's testing utilities to send HTTP requests to running application and assert response status codes, headers, and body content. Tests cover authentication flows (register, login, token refresh, logout), CRUD operations on all entities (create, read, update, delete with permission checks), document upload and retrieval (multipart form handling, presigned URL generation), and AI chat interactions (session creation, message sending, streaming responses). Database integration tests verify GORM model relationships (cascade deletes, preloading), transaction handling (rollback on errors), and constraint enforcement (unique indexes, foreign keys).

5.4.3 End-to-End Testing

End-to-end tests simulate real user workflows from browser interaction to database persistence using Playwright test framework. Tests launch full application stack (frontend, backend, database, Redis) and automate browser actions (clicking, typing, navigation) while asserting expected UI states and backend data changes. Critical user journeys tested include user registration and email verification, logging in and accessing dashboard, creating university and enrolling in course, uploading syllabus PDF and viewing extracted content, starting chat session and asking questions, and viewing analytics dashboard.

E2E tests run on pull requests to main branch and scheduled nightly builds, catching integration issues and regressions before production deployment. Test reports include screenshots and videos of test execution for debugging failures. Flaky test detection automatically retries failed tests up to 3 times, marking consistently failing tests for investigation.

5.5 Version Control & Branching Strategy

5.5.1 Git Workflow

The project follows Git Flow branching model with main branch containing production-ready code, develop branch for integration of completed features, feature branches for individual features (feature/syllabus-extraction), bugfix branches for bug fixes (bugfix/fix-login-validation), and release branches for release preparation (release/v1.2.0). Branch protection rules on main require pull request reviews, passing CI checks (linting, testing, build), and up-to-date branch before merging.

Commit messages follow Conventional Commits specification with type prefixes (feat:, fix:, docs:, refactor:, test:, chore:), scope indicating affected component (api, web, db), and imperative mood description. Example: "feat(api): add syllabus extraction endpoint with streaming progress". Semantic versioning (MAJOR.MINOR.PATCH) tags releases based on changes: MAJOR for breaking API changes, MINOR for new features maintaining backward compatibility, and PATCH for bug fixes.

5.5.2 Code Review Process

All code changes require pull request review before merging to develop or main branches. Pull request template includes description of changes, related user stories/issues, testing performed, and checklist for reviewer (code quality, test coverage, documentation, security considerations). Automated checks run on every pull request including linting (ensuring code style compliance), unit tests (verifying logic correctness), integration tests (checking component interactions), and build verification (confirming no compilation errors).

Review guidelines emphasize code readability (clear variable names, appropriate comments), performance considerations (avoiding N+1 queries, minimizing allocations), security best practices (input validation, SQL injection prevention), and maintainability (avoiding code duplication, clear separation of concerns). Approved pull requests are squash-merged to develop, creating single commit per feature for clean history.

5.6 Deployment Process

5.6.1 Deployment Environments

The project maintains three deployment environments: development (local Docker Compose with hot reload), staging (DigitalOcean Droplet mirroring production configuration), and production (DigitalOcean Droplet with load balancer and CDN). Development environment uses latest code from develop branch, staging deploys release candidates for final testing, and production deploys tagged releases from main branch.

Environment configuration uses separate .env files with environment-specific values including database connection strings (dev uses local PostgreSQL, production uses managed database), API endpoints (dev uses localhost, production uses custom domain), AI API keys (dev uses test credentials with rate limits, production uses production credentials), and feature flags (enabling experimental features only in development). Environment variables are injected at container runtime, never committed to version control.

5.6.2 Deployment Automation

GitHub Actions workflow automates production deployment on pushes to main branch. Deployment steps include building backend Docker image with multi-stage compilation (builder stage compiling Go binary, production stage using distroless base), building frontend Docker image with static export (next build && next export), pushing images to DigitalOcean Container Registry with semantic version tags (latest, v1.2.3), SSHing into production droplet, pulling latest images, running database migrations (make migrate-up), starting new containers with health checks enabled, waiting for health endpoint to return 200 OK, draining connections from old containers (30-second grace period), stopping old containers, and posting deployment notification to Slack channel.

Zero-downtime deployment is achieved through rolling update strategy where new containers start and pass health checks before old containers stop. Database migrations run before container updates to ensure new code finds expected schema. Rollback procedure involves tagging current commit (rollback-point), reverting commit on main branch, triggering deployment workflow, and running migration rollback if schema changes were made. Deployment metrics track deployment frequency (averaging 3 per week), lead time (commit to production in under 10 minutes), and failure rate (3% of deployments require rollback).