StepHabit Capstone Report

A 30-page technical and product deep dive

Prepared by: Engineering Team

Date: December 12, 2025

Abstract

StepHabit is a full-stack habit building and productivity platform that combines goal tracking, task management, smart scheduling, and AI-assisted coaching. This report provides a detailed examination of the system architecture, codebase, and implementation decisions, serving as both a design document and a reference for future contributors.

Contents

1 Introduction

StepHabit is designed to help users build sustainable routines through a blend of structured scheduling, community accountability, and AI-powered guidance. The project is implemented as a Node.js and Express backend with a React front end built on the CoreUI design system. It integrates relational data modeling via Sequelize, secure authentication flows, and a suite of productivity-oriented features (habits, tasks, calendars, notifications, achievements, and group challenges).

This capstone provides the narrative and technical depth typically expected from a multi-chapter thesis: an overview of motivations, platform architecture, implementation specifics, data structures, deployment considerations, testing approaches, and future roadmap. Throughout, the document references concrete code paths to anchor discussion in the current repository state.

To reach a thesis-scale depth, each chapter includes expanded rationale, design trade-offs, and operational runbooks. Readers can treat the report as a combined architecture decision record (ADR) collection, onboarding guide, and roadmap for future feature work.

1.1 Problem Statement and Goals

Individuals struggle to translate aspirational goals into daily action. StepHabit aims to solve this by combining three pillars:

1. **Clarity**: capture habits, tasks, and schedules in a single source of truth.
2. **Momentum**: surface progress, achievements, and reminders to keep users engaged.
3. **Coaching**: leverage AI prompts and habit plan generation to guide users toward actionable steps.

1.2 Scope of This Report

The following chapters dissect backend services, data models, REST APIs, frontend UI flows, and supporting utilities. Each section blends descriptive text with rationale, patterns, and future-proofing guidance to help maintainers extend the system with confidence.

2 System Overview

2.1 Technology Stack

- **Backend**: Node.js with Express and Sequelize ORM for PostgreSQL. Core entry point: `Backend/server.js`.
- **Frontend**: React (Vite) using CoreUI components for rapid admin-style layouts and navigation, defined in `Frontend/src`.
- **AI Services**: LangChain clients configured for Anthropic models to support habit plan generation and rewriting.

- **Infrastructure**: Environment-driven configuration via `dotenv`, with static uploads served from `/uploads`.
- **Tooling**: ESLint and Prettier for style consistency, nodemon for local hot reloads, and Vite for fast module bundling in the frontend.
- **Documentation**: This LaTeX artifact, inline JSDoc-style comments in controllers, and CoreUI component stories to make UI interactions clear.

2.2 High-Level Capabilities

- Habit creation, categorization, daily goals, and progress logging.
- Task management with durations, color labels, status transitions, and scheduling constraints.
- Calendar integration and busy-block tracking to avoid conflicts.
- Achievements, group challenges, and community messaging to enhance motivation.
- Notifications and direct messaging for real-time engagement.
- AI habit coaching: plan suggestions and idea rewrites to improve habit quality.
- Persistent assistant memories to retain per-user context when generating future coaching responses.
- Calendar ingestion for external events to prevent conflicting habit or task plans.

2.3 Non-Functional Requirements

Non-functional considerations shape implementation constraints and quality bars:

- **Reliability**: Idempotent cleanup routines ensure a consistent database state even after deployments or crashes.
- **Security**: JWT authentication, encrypted password storage, and scoped resource ownership checks protect user data.
- **Performance**: Lazy-loaded frontend routes and pagination-friendly API endpoints keep perceived latency low on modest hardware.
- **Maintainability**: Centralized model associations and reusable middleware reduce duplication as new features land.
- **Extensibility**: AI connectors and calendar integrations are pluggable to support future providers.
- **Observability**: Structured logging and consistent error responses make incident triage and monitoring simpler.

3 Backend Architecture

3.1 Express Application Entry Point

The `server.js` file configures middlewares, route mounts, database sync, and health checks. On startup, the server authenticates the database connection, performs cleanup on legacy tables, enforces optional column additions, and then synchronizes models. It mounts API routers for users, habits, progress, schedules, notifications, group challenges, achievements, friends, analytics, tasks, avatars, daily challenges, smart scheduling, library content, AI chat, assistant memories, calendar data, and messaging.

The startup pipeline reflects a pragmatic choice: use `sequelize.sync` with `alter` rather than bespoke migrations. In small teams, this accelerates delivery but requires guardrails such as preflight integrity checks and backups. Future iterations could add `umzug` migration scripts to codify schema changes for production-grade rollouts.

3.2 Error Handling Strategy

A global error handler logs unexpected exceptions and returns a generic HTTP 500 response. Individual controllers validate inputs and return 400 or 404 responses for user-facing errors (e.g., missing identifiers, invalid statuses).

3.3 Database Synchronization and Migration Safety

During startup, the application:

1. Authenticates the database connection.
2. Drops legacy tables (e.g., `user_settings`) when found.
3. Cleans orphaned records across assistant memories, calendar artifacts, notifications, progress logs, tasks, habits, and friendships.
4. Adds optional columns (e.g., `avatar` on users) if absent.
5. Executes `sequelize.sync` { `alter:  true` } to align schemas.

This pattern balances resilience with forward compatibility for incremental schema evolution.

3.4 Request Lifecycle

1. **Inbound**: Requests enter through Express, passing JSON body parsing and CORS middleware.
2. **Authentication**: Protected routes validate JWTs and attach user context. Public routes (e.g., registration) skip this step.
3. **Validation**: Controllers verify payload shape and accepted enums.
4. **Business Logic**: Service functions compute derived values, guard against conflicts, and orchestrate ORM operations.

5. **Persistence**: Sequelize executes queries, emitting events for logging and returning hydrated models.
6. **Response**: Payloads are normalized; errors funnel to the global handler for consistent structure.

3.5 Observability

Operational visibility keeps the platform trustworthy:

- **Logging**: Structured logs around startup steps, cleanup actions, and controller branches aid debugging.
- **Metrics**: While not yet implemented, hooks exist to layer in request timing, DB latency, and AI token usage metrics.
- **Tracing**: A future improvement would wrap Sequelize calls with distributed trace IDs to follow user actions end-to-end.

4 Data Modeling

Sequelize models define relational structures for user-centric productivity data. Associations live in `Backend/models/index.js`, ensuring a single source of truth for relationships.

4.1 Core Entities

- **User**: owns habits, tasks, calendar events, notifications, settings, friendships, achievements, assistant memories, and more.
- **Habit**: tied to a user, with schedules and progress logs to capture recurring actions.
- **Task**: user-owned with duration metadata, status, scheduling bounds, and aesthetic attributes.
- **Schedule** and **BusySchedule**: define time blocks for habits and calendar conflicts.
- **Progress**: links users to habits with dated completion metrics.
- **Achievement** and **UserAchievement**: milestone tracking via many- to-many relationships.
- **Friend**: self-referential pivot for bidirectional user friendships.
- **GroupChallenge** and **UserGroupChallenge**: collaborative habit competitions and participation tracking.
- **Notification**: user-targeted alerts.

- **AssistantMemory**: AI assistant context tied to users.
- **CalendarIntegration** and **CalendarEvent**: synchronize external calendar data with user accounts.
- **ChatMessage** and **GroupChallengeMessage**: messaging constructs for direct and challenge-specific chats.
- **RegistrationVerification**: support for email verification flows.
- **LibraryContent**: curated learning materials that accompany habit plans.
- **DailyChallenge**: a rotating mini-habit to increase engagement and provide quick wins.

4.2 Association Patterns

Associations emphasize:

- Clear ownership (e.g., `User.hasMany(Habit)`, `Task.belongsTo(User)`).
- Many-to-many bridges for achievements and group challenges via explicit pivot tables.
- Self-referential friendships through a `Friend` join model with requester and recipient roles.
- Hierarchical calendar structures linking integrations to events and users.

4.3 Entity Attributes

Model	**Key attributes**
User	id, name, email, password, avatar, timezone, preferences, verification status
Habit	id, user_id, title, description, category, daily_goal, schedule metadata, color
Task	id, user_id, name, status, duration, schedule_after, due_date, color, priority
BusySchedule	id, user_id, start, end, description, source integration
Progress	id, user_id, habit_id, date, completion count, notes
Achievement	id, name, description, threshold, icon
Notification	id, user_id, type, payload, read flag
AssistantMemory	id, user_id, role, content, metadata, token counts
CalendarEvent	id, user_id, integration_id, title, start, end, status

4.4 Data Lifecycle

Data flows through a predictable lifecycle: creation (via user action or AI suggestion), validation (schema and business rules), persistence (Sequelize), consumption (UI queries and notifications), and archival (cleanup scripts). For long-term compliance, a deletion pipeline should anonymize PII while retaining aggregate analytics.

5 API Layer

5.1 Habit Controller

The habit controller handles CRUD operations and AI-assisted idea refinement:

- **Listing**: fetches habits by `user_id`, ordered by creation time.
- **Creation**: validates required `title` and `user_id`, normalizes numeric targets, and persists metadata such as category and daily goals.
- **AI Suggestions**: invokes LangChain-backed services to generate habit plans and rewrite habit ideas for clarity.
- **Update/Delete**: supports partial updates, cascaded removal of progress and schedules, and consistent 404 responses when records are absent.

5.2 Task Controller

Task endpoints focus on flexible creation and status management:

- **Listing**: returns tasks for a user ordered by creation date.
- **Creation**: enforces `name` and `user_id`, validates status against allowed values, normalizes durations, and stores optional scheduling bounds like `schedule_after` and `due_date`.
- **Status Updates**: ensures valid identifiers and status transitions before persisting state changes.

5.3 User and Authentication Controllers

User routes manage registration, login, profile updates, and email verification. Registration flows create users, issue verification tokens, and optionally seed starter habits. Login validates credentials, signs JWTs, and returns profile metadata to hydrate the frontend quickly.

5.4 Messaging and Community Controllers

Messaging endpoints store chat messages, associate them with senders and recipients, and fetch conversation histories in chronological order. Group challenge routes allow creating challenges, joining or leaving groups, posting contextual messages, and tracking leaderboard standings.

5.5 AI Controller

The generic AI endpoint proxies requests to configured large language models. It accepts prompts and system instructions, returning generated text. This decouples front-end experimentation from backend provider changes and centralizes API key management.

5.6 API Surface Map

Table **??** summarizes prominent endpoints and intents.

Table 2: Representative API endpoints

Path	**Method(s)**	**Purpose**
/users/register	POST	Create user, trigger verification email
/users/login	POST	Authenticate and issue JWT
/habits	GET/POST	List or create habits for a user
/habits/:id	PATCH/DELETE	Update or remove a habit
/habits/ai/plan	POST	Generate structured habit plan
/tasks	GET/POST	List or create tasks
/tasks/:id/status	PATCH	Transition a task status
/notifications	GET	Retrieve unread alerts
/messages	GET/POST	Fetch or send direct messages
/group-challenges	GET/POST	Explore or create challenges
/calendar	GET/POST	Sync external calendar data

5.7 Route Surface

The Express server mounts routers for users, habits, progress, schedules, notifications, group challenges, achievements, friends, analytics, tasks, avatars, daily challenges, smart scheduler, library, AI chat, assistant, calendar, messaging, and the generic AI endpoint. Each router encapsulates its own validation and business logic while sharing authentication and error handling patterns established in `server.js`.

6 AI and Coaching

AI-powered features rely on LangChain integrations targeting Anthropic models. The habit controller exposes endpoints to generate structured habit plans and rewrite freeform habit ideas. These services enable:

- **Behavioral scoping**: breaking ambitious habits into clear steps.
- **Tone and clarity**: rewriting user input into motivating, actionable statements.
- **Feedback loops**: embedding AI prompts within the UI (*HabitCoach*) to guide users before they commit to new routines.

6.1 Prompt Design

Prompts emphasize brevity and behavioral nudges. The system primes the model with the user's stated goal, constraints (time, tools, energy), and preferred tone. Outputs are formatted as checklists or bullet points to ease comprehension.

6.2 Safety and Guardrails

- **Rate Limits**: configurable caps on requests per minute to prevent abuse and control costs.
- **Content Filtering**: client-side checks can reject inappropriate input before forwarding to the model.
- **Auditability**: logs of prompts and responses (with PII minimization) support debugging and user support escalations.

6.3 Future AI Extensions

Potential improvements include retrieval-augmented generation using user history as context, sentiment-aware encouragement, and automatic schedule insertion based on predicted effort for each habit step.

7 Frontend Architecture

7.1 Framework and Layout

The front end uses React with Vite for tooling and CoreUI for layout primitives. Navigation is defined in `src/_nav.js`, grouping routes into *Plan*
track, *Connect*, and *You* sections. The base layout provides a sidebar, header, and content area consistent with admin dashboards.

7.2 Routing and Lazy Loading

Routes in `src/routes.js` lazy-load feature bundles for authentication, dashboards, habits, planner, notifications, tasks, profile, community, and support pages. This keeps the initial bundle lean while supporting a wide surface of productivity tools.

7.3 Key Screens

- **Dashboard**: centralizes user metrics, reminders, and navigation shortcuts.
- **Planner and Schedules**: offer calendar-style overviews and smart scheduling options.
- **Tasks and Habits**: dedicated pages for CRUD operations, progress visualization, and AI habit coaching.

- **Community**: friends, messaging, group challenges, and leaderboards to foster accountability.
- **Profile**: personal settings, avatars, and notification preferences.
- **Support**: contact forms and help center resources.

7.4 State Management

Local component state handles transient UI interactions, while API responses are cached through simple hooks. For larger-scale collaboration features, adopting a state library (Redux Toolkit or Zustand) would standardize loading states and optimistic updates.

7.5 Design System Integration

CoreUI provides data tables, cards, charts, and navigation primitives. Custom theme tokens (colors for habit categories, accent colors for achievements) keep branding consistent. Icons and typography are selected for readability on dense dashboards.

7.6 Accessibility Considerations

Contrast ratios, focus outlines, semantic HTML landmarks, and ARIA labels are prioritized to make the planner usable with keyboards and screen readers. Motion is minimized, and user preferences for reduced animations can be honored through CSS media queries.

7.7 Responsive Layouts

Grid-based layouts adapt cards and tables for tablets and phones. Collapsible sidebars preserve navigation without sacrificing screen space for calendar views. Touch-friendly hit targets (44px minimum) are recommended for mobile interactions.

8 Security and Validation

8.1 Authentication and Authorization

JWT-based authentication (via `jsonwebtoken`) secures protected endpoints. Password storage leverages `bcryptjs`. Middleware layers would typically verify tokens before reaching controller logic (implementation details reside in route modules beyond this report's excerpt).

8.2 Input Validation and Error Semantics

Controllers validate required fields (e.g., `user_id`, `name`) and acceptable status values. Responses consistently use 400 for malformed requests and 404 for missing resources, while the global error handler standardizes 500 responses for unexpected failures.

8.3 Authorization Model

Resources are always scoped by `user_id`. Multi-tenant boundaries become critical if organizations are introduced; a future `organization_id` would partition data and permissions. Role-based access control could differentiate members, admins, and coaches.

8.4 Secrets Management

API keys for AI providers and JWT secrets reside in environment variables. In production, secrets should be injected via a vault (e.g., AWS Secrets Manager or HashiCorp Vault) with rotation policies and audit trails.

8.5 Data Privacy

Personally identifiable information (PII) such as names and emails are limited to essential use cases. Features that log AI prompts should redact PII or hash identifiers to comply with privacy regulations.

8.6 Data Integrity

Startup routines proactively remove orphaned records before syncing schemas, reducing referential drift. Associations in `models/index.js` enforce foreign-key relationships, and cleanup helpers delete dependent records when parents are removed (e.g., habit deletion cascades to progress and schedules).

9 Scheduling and Productivity Workflows

9.1 Habits vs. Tasks

Habits capture recurring behaviors with optional daily goals and categories, whereas tasks represent discrete units of work with durations and due dates. Both types surface on planners and dashboards, but habits emphasize streaks and progress logs, while tasks emphasize completion status and time allocation.

9.2 Smart Scheduling and Busy Blocks

Busy schedules protect time on the calendar, preventing overbooking. Planner routes connect to smart scheduling logic (backend service not shown in this excerpt) to suggest optimal time slots based on availability and task metadata.

9.3 Progress and Achievements

Progress entries timestamp habit completions. Achievements and group challenges translate those completions into motivational milestones, with many-to-many links enabling users to share progress and compete collaboratively.

9.4 Calendar Integration Patterns

External calendars sync through OAuth (future enhancement) or token-based feeds. Events are normalized into `CalendarEvent` rows, with fields for start, end, status, and source. Conflicts are detected by overlapping intervals against existing busy blocks.

9.5 Notification Cadence

Reminders respect user timezones. A typical cadence includes morning summaries, midday nudges for pending habits, and evening wrap-ups. Push mechanisms could include email, SMS, or in-app toasts depending on user preferences.

10 Community and Communication

10.1 Friendships and Group Challenges

Friend relationships are modeled through a self-referential join table, allowing bidirectional invites and confirmations. Group challenges link users to collective goals, with messages scoped to challenges for contextual discussion.

10.2 Notifications and Messaging

Notifications provide user-specific alerts (e.g., reminders, challenge updates), while direct messages enable one-to-one communication. The chat model tracks senders and recipients, preserving conversational history.

10.3 Moderation

Community features require guardrails: profanity filters, abuse reporting, and rate limiting on message posting. Admin tools could allow muting or removing users who violate community guidelines.

10.4 Leaderboards

Leaderboards aggregate progress and challenge completions. To avoid discouraging new users, percentile-based tiers or personal best tracking can be surfaced alongside absolute rankings.

11 Deployment Considerations

11.1 Environment Management

The backend expects environment variables for database connectivity, JWT secrets, and third-party integrations. The frontend can be configured via Vite's environment handling for API base URLs and feature toggles.

11.2 Build and Runtime

- **Backend**: run `npm install` then `npm run start` (or `npm run dev` with nodemon) in `Backend/`.
- **Frontend**: run `npm install` then `npm run dev` in `Frontend/`, accessing the app via the configured host and port.

11.3 Static Assets and Uploads

The Express server serves uploaded files from `/uploads`, enabling avatar and attachment features without requiring a separate CDN in development.

11.4 Containerization

Dockerfiles can encapsulate both backend and frontend. Multi-stage builds shrink image sizes by separating dependency installation from runtime layers. Compose files would orchestrate the web app, database, and reverse proxy (e.g., Nginx).

11.5 Continuous Integration/Delivery

CI pipelines should lint, test, build, and publish artifacts. CD can deploy to a staging environment for manual QA before promoting to production. Feature flags allow safe rollouts of experimental AI prompts.

11.6 Scaling Considerations

Horizontal scaling of the backend relies on statelessness; session data should be JWT-based or stored in Redis. The database can scale vertically or via read replicas. CDN-backed static hosting for the frontend reduces load on origin servers.

12 Testing and Quality

12.1 Manual and Automated Checks

While automated test suites are not included in the current repository snapshot, quality assurance can follow these pillars:

- Unit tests for controllers to validate HTTP semantics and error handling.
- Integration tests exercising Sequelize models and associations.
- End-to-end UI tests covering critical flows: registration, login, habit creation, task completion, and AI coaching interactions.

12.2 Static Analysis

Linters (ESLint) and formatters (Prettier) can be integrated in both frontend and backend pipelines. Type checking with TypeScript or JSDoc annotations would further reduce runtime defects.

12.3 Test Data Management

Factories or fixtures can provision repeatable datasets for integration tests, including users with habits, tasks, and friendships. A seeded demo mode would accelerate manual QA and stakeholder demos.

12.4 Performance Testing

Load tests targeting task creation, habit plan generation, and message posting ensure endpoints remain responsive under concurrent usage. Synthetic monitoring could periodically exercise critical paths.

12.5 UX Research

Qualitative feedback loops—surveys, usability tests, and feature flags for A/B tests—help validate whether AI prompts and scheduling recommendations actually improve habit adherence.

13 Future Work

- **Advanced Analytics**: richer habit adherence trends, burn-down charts for tasks, and predictive scheduling.
- **Collaboration**: shared planners for teams or families, with role-based permissions.
- **Mobile Experience**: React Native client reusing API contracts to reach more platforms.
- **Offline Support**: local-first sync for tasks and habits, with conflict resolution strategies.
- **Extensible AI**: personalized coaching tuned to user progress and contextual data.
- **Data Warehouse**: event pipelines and BI dashboards to study cohort retention and feature adoption.
- **Compliance**: GDPR/CCPA tooling for export, deletion, and consent tracking.
- **Plugin Ecosystem**: allow third-party extensions (focus timers, mindfulness modules) via scoped tokens and manifest declarations.
- **Open API**: document the REST surface with OpenAPI/Swagger for partner integrations.

- **Gamification Depth**: seasonal events, badges with rarity tiers, and streak insurance options to keep engagement steady.

14 Project Management and Delivery

14.1 Milestones

- **M1 — Foundations**: establish models, authentication, and CRUD for habits/tasks; ship a minimal planner view.
- **M2 — Engagement**: add achievements, notifications, and group challenges with leaderboards.
- **M3 — AI Coaching**: release habit plan generation, idea rewriting, and assistant memories for contextual guidance.
- **M4 — Integrations**: calendar sync, avatar uploads, and library content for education.
- **M5 — Hardening**: observability, migration tooling, and performance optimizations for scale.

14.2 Risk Management

Key risks include AI cost overruns, data breaches, and low user retention. Mitigate with budget alerts, layered security reviews, and continuous UX research. A risk register should assign owners and track mitigations.

14.3 Contribution Guidelines

Contributors should open issues with reproduction steps, follow coding standards consistent with ESLint/Prettier defaults, and include unit tests where possible. Pull requests must describe API changes and database impacts to avoid regressions.

15 Operations Runbook

15.1 Incident Response

1. Triage incoming alerts (error spikes, elevated latency) and identify blast radius.
2. Roll back recent deployments if correlated with incidents.
3. Examine logs for failing routes; reproduce with synthetic requests.
4. Communicate status updates to stakeholders and users when appropriate.
5. Post-incident, produce a timeline and remediation tasks to prevent recurrence.

15.2 Backup and Recovery

Nightly database backups should capture user-generated data and uploads. Recovery drills ensure restoration procedures are well understood. For AI prompts, a privacy-safe log store enables replay without leaking sensitive content.

15.3 Change Management

Feature flags and canary deployments reduce blast radius. Schema changes should be backward compatible, staged with additive columns before removing legacy fields.

16 Ethical Considerations

StepHabit influences behavior; ethical design is crucial. AI advice should avoid medical or mental health diagnoses, provide disclaimers, and encourage users to consult professionals when appropriate. Data minimization and clear consent flows respect user autonomy. Community guidelines should promote positivity and inclusivity.

17 Conclusion

StepHabit blends goal tracking, scheduling, social accountability, and AI assistance into a cohesive platform. Its architecture balances modularity (via Express routers and Sequelize models) with rapid UI development (through CoreUI components and lazy-loaded routes). The cleanup routines and association mappings provide a strong foundation for data integrity, while the AI endpoints introduce differentiated value for habit formation. This report captures the current blueprint and offers guidance for continued iteration toward a more capable and supportive productivity companion.