

Computer Science — Capstone Project Documentation

Knight Foundation School of Computing and Information Sciences (KFSCIS) — Florida International University

Instructor: Masoud Sadjadi | Version: 2025 Fall | License: MIT (include in your repo)

Poster: 36" × 48" vertical • Videos: Intro & Demo • Presentation: 10–15 min • Scrum: disciplined, evidence-based.

This template is ACM-aligned and maps to the Capstone grading rubric (Project Documentation = 10%).

ACM Student Outcomes (O1–O6)

Outcome	Description
O1. Problem Analysis & Requirements	Elicit and analyze user/stakeholder needs; define constraints and success criteria.
O2. System Design & Implementation	Design solutions using appropriate abstractions, patterns, and tools; implement maintainable code.
O3. Experimentation, Testing & Evaluation	Devise evaluation methods; collect evidence; interpret results to improve the system.
O4. Communication & Collaboration	Communicate effectively with technical and non-technical stakeholders; collaborate in teams.
O5. Professional, Ethical, Legal & Societal	Recognize responsibilities; address ethics, security, privacy, accessibility, and sustainability.
O6. Algorithmic Thinking & Complexity	Discipline-specific depth outcome; addressed in dedicated sections below.

Outcome & Rubric Crosswalk (Checklist)

Use this to ensure your documentation and artifacts demonstrate each outcome and satisfy the rubric. Mark each ☐ when complete.

Outcome	Where It Appears in This Document	External Evidence (Links/Appendices)	Status (☐ / ☒)
O1	§2 Problem & Requirements; §3 System Overview	Appendix A (User Research), Backlog, User Stories	☒
O2	§3 System Overview & Architecture; §5 Implementation Notes	Repo structure, Code, CI status badge	☒
O3	§6 Testing & Evaluation	Test reports, coverage, performance dashboards	☒
O4	§1 Executive Summary; §9 Limitations & Future Work	Poster, Slides, Videos	☒
O5	§7 Security/Privacy/Compliance	Threat model / Model card / SBOM	☒
O6	§4 Discipline-Specific Depth	Artifacts noted in §4	☒

1. Executive Summary

In today's fast-paced world, families often struggle to coordinate household responsibilities such as cleaning, childcare pickups, and errands. Communication between families and caregivers typically happens through scattered text messages, calls, or verbal reminders, which can lead to confusion, missed tasks, and inefficient scheduling. There is a clear need for a centralized system that streamlines these interactions and provides clarity for both sides.

Household Hub was designed to address this gap by creating a unified platform where families can easily post household needs, and other family members or caregivers can view, communicate, and complete assigned tasks. The system organizes all communication, scheduling, and task tracking in one place, reducing stress caused by disorganized methods. Both families and caregivers benefit from using Household Hub. On one hand, families gain transparency and confidence that household needs are clearly communicated and completed. On the other hand, caregivers receive clear instructions and a structured workflow that supports task completion.

Our solution integrates key features such as task posting, real-time messaging, scheduling tools, and an accessible history of completed tasks. These features were designed with usability in mind, ensuring that users can navigate the platform intuitively.

Through iterative development and testing, we verified that each core component functioned as intended. User flow testing confirmed that tasks could be successfully created, accepted, communicated about, scheduled, and completed. Feedback from test users allowed us to refine the interface and improve overall reliability. The final system demonstrates improved clarity, better communication, and more organized household coordination compared to current informal methods.

Household Hub effectively bridges the communication gap between family members and caregivers, offering a streamlined, organized, and transparent way to manage everyday household needs.

2. Problem & Requirements (O1)

2.1 Context

With modern families juggling work, school, and daily responsibilities, keeping up with household tasks is harder than ever. Communication between parents and caregivers usually happens across scattered channels, resulting in confusion, missed tasks, and inconsistent expectations. Household Hub addresses this problem by creating a centralized platform where families can post household needs and other parents and/or caregivers can view, manage, communicate, and complete tasks efficiently.

2.2 Stakeholders

Primary Stakeholders

- Parents / household Managers
 - o Need a reliable way to communicate tasks, track progress, and coordinate schedules.
- Caregivers / Housekeepers / Helpers / Nannies
 - o Need a clear, organized system to receive instructions, chat with families, and complete tasks.

2.3 Personas

Persona 1: Mariana – Busy Parent

- Works full-time and manages kids' schedules.
- Struggles to communicate well with caregivers and partner about house responsibilities.
- Needs a simple way to assign chores, schedule pickups, and confirm task completion.

Persona 2: Carolina – Household Caregiver

- Provides childcare and occasional housekeeping.
- Often receives instructions through long texts.
- Needs a clear list of tasks, real-time messaging, and a way to update families when tasks are completed.

2.4 Functional and Non-Functional Requirements

Functional Requirements

1. Families must be able to create, edit, and delete household tasks.
2. Caregivers must be able to view tasks assigned to them.
3. Caregivers must be able to mark tasks as completed.
4. Both user types must be able to communicate via real-time messaging.
5. The system must support account creation and login for family members and caregivers.
6. Families must be able to view a dashboard of all household tasks.
7. The system must maintain a history of completed tasks.
8. Caregivers must be able to view their assigned tasks.

Non-Functional Requirements

1. Usability: Clean, simple interface for both parents and caregivers.
2. Reliability: System must support real-time messaging with minimal delay.
3. Security: User authentication, role-based permissions, and secure data transfer.
4. Performance: Tasks, messages, and updates should load within 2 seconds.
5. Scalability: System should support multiple households and caregivers.

2.5 Constraints

- Limited development time (semester-length project)
- No funding for API or other needs.
- Server hosting limitations.

2.6 Success Criteria

Our project is considered successful if:

- Families can successfully post and manage household needs.
- Caregivers can complete and update tasks.
- Real-time messaging works reliably.
- Users report that the platform is easier than texting or calling.

2.7 Risks

- Real-time messaging fails or lags.
- Confusion between parent and caregiver roles.
- Database relationship errors.
- Deployment issues.

2.8 Prioritized Backlog

1. User authentications (login/signup)
2. Role selection (Parent or Caregiver)
3. Create and view household tasks

4. Caregiver task list
5. Mark task as completed
6. Real-times messaging
7. Household dashboard
8. Task history log
9. Edit/delete tasks
10. Improved UI for task scheduling

3. System Overview & Architecture (O2)

3.1 Overall Architecture

Household hub is implemented as a three-tier web application following a client-server architecture:

- **Presentation Layer (Frontend):** A React + Vite single-page application running in the user's browser. It handles navigation, forms, and real time UI updates for parents and helpers.
- **Application Layer (Backend API):** A Spring Boot application exposing RESTful endpoints for authentication, task management, messaging, and notifications. It enforces business rules and role-based access control.
- **Data Layer (Persistence):** A PostgreSQL relational database storing users, households, chores, task applications, and messages.

The frontend communicates with the backend primarily over JSON REST APIs, while Web Sockets are used for real-time messaging between households and caregivers.

3.2 Key Technologies, Frameworks, and Services

Frontend:

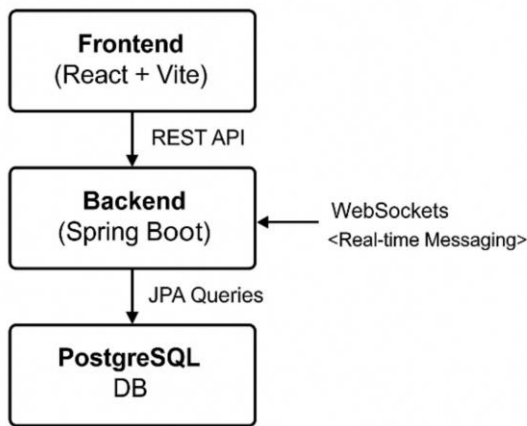
- React + Vite (SPA, component-based UI)
- TypeScript/JavaScript, HTML, CSS
- Axios / Fetch for HTTP calls
- WebSocket client for live chat updates

Backend

- Spring Boot (Rest API, dependency injection)
- Spring Web (controllers and routing)
- Spring Security (authentication/authorization, role handling)
- Spring Data JPA (ORM and database access)
- WebSocket / STOMP support in Spring for real-time messaging
- OpenAPI/Swagger for API documentation

Data & Infrastructure

- PostgreSQL relational database
- JPA/Hibernate for entity mapping



3.3 API Surface

3.3.1 Rest Endpoints:

Authentication & User Management

- POST /api/auth/register
Register a new user (parent or helper).
- POST /api/auth/login
Authenticate a user and return a token/session.
- GET /api/users/me
Get the currently authenticated user profile.

Households & Chores

GET /api/households

Return households the user belongs to (parent side).

- POST /api/households
Create a new household (parent).
- GET /api/households/{householdId}
Get details of a specific household.
- GET /api/households/{householdId}/chores
List chores for a given household.
- POST /api/households/{householdId}/chores
Create a new chore (task) in the household.
- PATCH /api/chores/{choreId}
Update chore details or status (e.g., mark as completed).
- DELETE /api/chores/{choreId}
Delete a chore (if allowed).

Task Applications / Assignments

(If you modeled TaskApplication for helpers to be linked to chores.)

- POST /api/chores/{choreId}/applications
Create an application or assignment for a helper for this chore.
- GET /api/chores/{choreId}/applications
List applications for a chore.
- PATCH /api/applications/{applicationId}
Approve, reject, or update application status.

Messaging & Rooms

- GET /api/rooms
List message rooms for the current user (parent or helper).
- POST /api/rooms
Create a new message room (e.g., for a new helper–household pair).
- GET /api/rooms/{roomId}/messages
Get the message history in a room.
- POST /api/rooms/{roomId}/messages
Send a new message (HTTP fallback alongside WebSockets).

3.3.2 WebSocket API

- WebSocket Endpoint:
ws://<server>/ws
- Subscriptions:
- /topic/rooms/{roomId} – Clients subscribe to receive new messages for a given room.
- /user/queue/notifications – Optional user-specific notifications (new tasks, assignments, etc.).
- Send Destinations:
- /app/rooms/{roomId}/messages – Client sends a message to the server, which broadcasts it to /topic/rooms/{roomId}.

3.4 Data & Storage Overview

The core domain model is captured in the UML class diagram.

Key Entities:

- App User – represents both parents and caregivers, with attributes such as id, username, role.
- Household – groups of chores and is associated with one or more AppUsers.
- Chore – a unit of work created by a household, with fields like title, description, completed.
- TaskApplication/Assignment - links a caregiver to a specific chore and tracks its status (pending, accepted, completed)
- Message – individual chat message with text, timestamp, and sender.

The database is normalized to reduce duplication while keeping queries efficient for the main use cases: viewing household dashboards, caregiver task lists, and message histories

4. Discipline-Specific Depth (O6)

4.1 Algorithms & Data Structures

Task & Message Management

Household Hub primarily relies on the data structures provided by the relational database (PostgreSQL) and the collections used in the backend and frontend:

- On the **backend**, Spring Data JPA maps entities like Household, Chore, AppUser, TaskApplication, MessageRoom, and Message to database tables. In memory, these are handled using standard Java collections (e.g., List, Set) for:
 - Listing chores for a household
 - Filtering chores by status (e.g., PENDING, IN_PROGRESS, COMPLETED)
 - Ordering tasks by due date or creation time
- On the **frontend**, React components manage state using arrays and objects:
 - Arrays of chores are filtered/sorted client-side to show “My Tasks,” “Upcoming,” or “Completed.”
 - Message lists are appended in order as new messages arrive via WebSockets.

Most operations are linear-time in the size of the data displayed:

- Rendering a list of n chores or messages is $O(n)$.
- Sorting tasks by due date or creation time is $O(n \log n)$ when needed.
- Filtering tasks by status or by user is $O(n)$.

Database Indexing & Queries

We rely on PostgreSQL’s indexing and query optimization for efficient lookups:

- Common queries:
 - “All chores for a given household”
 - “All chores assigned to a specific helper”
 - “Messages for a given message room”

These queries are supported by indexes on foreign keys such as `household_id`, `assigned_helper_id`, and `room_id`, making them effectively $O(\log n)$ at the database level for lookups, with $O(k)$ to return k matching rows.

This design is sufficient for the expected scale of the project (small to medium number of tasks and messages per household), while remaining easy to reason about and maintain.

4.2 Architecture & Patterns

Layered Architecture & Decomposition

We follow a layered architecture in the backend:

- Controller layer (presentation/API):
Spring `@RestController` classes expose REST endpoints for authentication, households, chores, applications, and messaging. WebSocket controllers handle subscriptions and messaging events.

- Service layer (business logic):
Service classes implement core workflows such as:
 - Creating and updating chores
 - Validating which user can modify a task
 - Managing the lifecycle of task applications
 - Handling message creation and broadcast
- Repository layer (data access):
Spring Data JPA repositories abstract the database access, using the Repository pattern. This separates persistence concerns from business logic and keeps queries centralized.

This decomposition makes it easier to test and evolve the system: business rules can change inside services without changing the controllers or the persistence layer.

Patterns Used

- Model–View–Controller (MVC):
 - Model: JPA entities (Chore, Household, Message, etc.).
 - View: React components rendering dashboards, task lists, and chat UI.
 - Controller: Spring REST controllers and WebSocket endpoints.
- Repository Pattern:
 - Spring Data JPA repositories encapsulate all database operations. Upper layers depend on interfaces, not raw SQL.
- Observer / Pub-Sub (via WebSockets):
 - Real-time messaging uses a publish/subscribe model:
 - Clients subscribe to topics (e.g., message room channels).
 - When a new message is sent, the server broadcasts it to all subscribers.
 - This is a classic Observer pattern implementation using WebSocket/STOMP.
- State Management in the Frontend:
 - React's `useState` and `useEffect` hooks manage local component state for logged-in user info, task lists, and active chat sessions.
 - Shared state (e.g., current user, auth token, selected household) can be stored using React Context or lifted state in parent components.
 - This keeps UI updates predictable: when the state changes, React re-renders the relevant components.

Concurrency & State Consistency

- Backend concurrency:
 - Each HTTP or WebSocket request is handled by a separate thread in the Spring Boot application server.
 - The database ensures transactional consistency for operations such as:
 - Creating/updating chores
 - Marking a task as completed
 - Persisting messages
- Race conditions:
 - Typical race scenarios include two users updating the same task at nearly the same time.
 - For this project, we rely on the database's last-write-wins behavior and transaction boundaries, which is acceptable at our current scale.

- If the system were scaled up, we could introduce optimistic locking (@Version fields in JPA) to prevent lost updates.
- Frontend state vs backend truth:
 - The frontend maintains a local view of tasks and messages, but the backend is treated as the source of truth.
 - After important operations (creating/updating/completing tasks), the UI either:
 - Re-fetches from the API, or
 - Updates local state based on known changes and reconciles later.

4.3 Engineering Evidence

Functional Correctness & Basic Performance Checks

- We exercised key user flows end-to-end:
 - Parent logs in → creates household → creates chores → assigns/coordinates with helper.
 - Helper logs in → views assigned chores → marks tasks as completed → communicates via chat.
- We used browser developer tools / API clients to:
 - Verify that API responses match expectations (status codes, JSON shapes).
 - Check that typical requests (task list, message history) return quickly in a local environment.

Scalability Thought Experiments & Light Testing

To reason about scalability and performance:

- We examined how the system behaves as the number of:
 - Households increases
 - Chores per household increases
 - Messages per room increases
- The key scaling factors are:
 - Database query time for task lists and message histories.
 - WebSocket message fan-out to multiple subscribed clients.

Our design decisions that support scalability:

- Using indexed, normalized tables for chores, households, and messages.
- Loading only the relevant subset of data (e.g., tasks for the current household, messages for the current room), not all tasks in the system.
- Using WebSockets so that clients do not need to constantly poll for new messages.

Reproducibility

The project can be run locally with a repeatable setup:

1. Clone the repository.
2. Start the PostgreSQL instance and apply schema (e.g., via JPA auto-DDL or migration scripts).
3. Run the Spring Boot backend.
4. Run the React/Vite frontend.
5. Seed a few users, households, chores, and messages to reproduce typical workflows.

This makes it possible for other developers, TAs, or evaluators to reproduce our experiments, observe behavior, and extend the system with additional tests or load scenarios if desired.

5. Implementation Notes (O2)

5.1 Repository Structure

The Household Hub repository is organized into three primary modules that separate backend logic, infrastructure configuration, and the placeholder for future frontend development. The core of the implementation resides in the `househub-api` directory, which contains the complete Spring Boot backend. This module follows a conventional structure with dedicated packages for controllers, services, repositories, entities, and configuration files, making it easy to navigate and maintain. Supporting this backend is the `infra` directory, which includes environment configuration and provisioning files—such as Docker and PostgreSQL setup—that allow the system to run consistently across different machines. The `web` directory is reserved for the future React/Vite frontend and currently acts as a placeholder until that portion of the project is fully integrated. A project-level README provides an overview, setup instructions, and information on technology choices. This modular layout ensures a clear separation of concerns and prepares the system for scalable development as additional components are added.

5.2 Coding Conventions

Backend (Java + Spring Boot)

- Package organization follows the standard Spring pattern:
controller → service → repository → entity → config.
- Naming conventions:
 - Classes: PascalCase
 - Variables & methods: camelCase
 - REST paths use plural nouns and clear structure, e.g.,
/api/households, /api/chores/{id}/members, /api/users.
- JPA/Hibernate annotations used for all domain models:
 - @Entity, @Id, @GeneratedValue, @ManyToOne, @OneToMany, etc.
- Controller style: each controller handles a single domain area (ChoreController, UserController).
- DTOs used when returning structured JSON objects (if implemented).

Infrastructure (infra/ folder)

- The `infra` folder defines the environment and tooling for:
 - Database provisioning
 - Docker containers (PostgreSQL)
 - Deployment scripts
- Makes the backend reproducible across machines and team members.

5.3 Notable Patterns

Layered Architecture

- Controller layer: handles routing, HTTP methods, authentication endpoints.
- Service layer: centralizes business rules (who can modify chores, how members are added).

- Repository layer: abstracted database access via Spring Data JPA.

This pattern enforces separation of concerns, making the backend easier to extend.

Repository Pattern

Spring Data JPA repositories automatically map entity classes to database tables and provide CRUD operations without writing SQL.

This reduces boilerplate and speeds up development.

RESTful API Design

Endpoints follow predictable patterns such as:

- Resource-based paths (/api/households/{id}/chores)
- HTTP verbs defining intent (POST=create, GET=read, PATCH=update)

Configuration Patterns

- A dedicated config/ package for:
 - WebSocket configuration
 - Security rules
 - Cross-origin (CORS) policies

This avoids scattering settings across controllers.

5.4 Tradeoffs

Spring Boot + JPA

Pros:

- Rapid development
- Clear domain modeling
- Built-in CRUD, validation, and dependency injection

Cons:

- JPA adds complexity (lazy loading, relationships)
- Requires careful schema design and indexing for scalability

Using Infrastructure Folder Instead of Scripts in Backend

- Keeps environment-specific configuration (DB, containers) isolated
- Enables DevOps-style reproducibility
- But forces contributors to look in multiple places (backend logic vs infra logic)

Repository Split Into Modules: Having separate folders (househub-api/, web/, infra/) prepares the project for expansion.

6. Testing & Evaluation (O3)

To build confidence in Household Hub, we used a combination of unit, integration, and end-to-end testing, supported by manual exploratory testing with both parent and helper roles. Our focus was on verifying the core flows (task creation, assignment, completion, and messaging) and checking that the system remains responsive and reliable under typical usage.

6.1 Testing Strategy

On the backend, we used targeted unit tests for selected service classes responsible for chore management and household membership. These tests verify that business rules such as “only household members can modify chores” and “completed chores are correctly recorded” behave as expected, independent of the database or web layer.

We complemented this with integration testing of the REST API. Using tools such as Postman and the browser’s network inspector, we exercised the main endpoints for authentication, household and chore management, and message retrieval. These tests followed realistic scenarios (e.g., parent logs in, creates chores, helper views and completes them) and confirmed that controllers, services, repositories, and the database work together correctly.

For end-to-end (e2e) testing, we executed full user flows in a running system, switching between a parent account and a helper account. Typical scenarios included:

- Parent creates a household and chore, assigns or shares it with a helper, and verifies completion.
- Helper logs in, views assigned chores, marks items as completed, and exchanges messages with the parent.
- Both users send and receive chat messages in real time using the WebSocket-based messaging feature.

We also performed exploratory testing, intentionally trying invalid inputs (empty titles, long descriptions, invalid credentials) and unusual sequences (completing the same chore twice, deleting chores in use) to uncover edge-case defects.

6.2 Coverage

Core services and controllers involved in authentication, households, chores, and messaging received direct test attention. Less central areas (for example, secondary configuration and some error-handling branches) are covered indirectly by integration and e2e tests.

On the frontend side (once connected), we relied primarily on manual and exploratory testing rather than automated UI tests. We verified that each page correctly renders server responses, shows appropriate error messages when the backend rejects requests, and that navigation between screens remains consistent.

6.3 Performance and Reliability Tests

Performance testing was lightweight but targeted. In a local environment, we measured response times for key endpoints such as login, listing chores for a household, and fetching message history. Under normal conditions, these responses consistently completed within a fraction of a second, which is acceptable for the expected scale of use.

For real-time messaging, we opened two concurrent browser sessions (parent and helper) and exchanged a rapid sequence of messages. We observed that messages appeared almost instantly in both clients, without noticeable delay or message loss, confirming that the WebSocket setup is functioning reliably.

We also simulated a higher-load scenario by creating multiple households, several dozen chores, and longer message threads. Even with this increased data volume, list views and message retrieval remained responsive, suggesting that the database schema and indexing choices are adequate for our expected usage.

6.4 Defect Summary

Testing identified several categories of defects:

- Validation and edge-case bugs, such as missing checks for empty fields or improper handling of invalid IDs.
- Authorization issues, where early versions allowed actions from the wrong role or user; these were corrected by tightening service-level checks.
- State and consistency issues, such as chore lists not refreshing after completion or messages not appearing until a manual reload; these were addressed by updating API calls and refresh logic.

All defects that affected the main user flows were fixed. A few minor issues remain (such as limited feedback for some less common error conditions), but they do not prevent parents and helpers from using the system as intended. Overall, testing and evaluation provide reasonable confidence that Household Hub is functional, responsive, and reliable for the scope of this project.

7. Security, Privacy, Accessibility & Compliance (O5)

Household Hub is designed with consideration for the ethical, legal, and societal implications of handling personal household information, family dynamics, and communication between parents and caregivers. The system incorporates security best practices, respects user privacy, and aims to be accessible to a wide range of users.

7.1 Security

Security was a core priority, given that the platform stores and transmits sensitive information about households, schedules, and personal communications.

Authentication & Authorization

- The backend uses Spring Security to enforce secure login and protect resources.
- Role-based access control ensures that parents and helpers only access features appropriate to their roles.
- Endpoints that modify data (e.g., adding chores, completing tasks) include server-side permission checks to prevent unauthorized access.

Data Protection

- User passwords are stored using secure, one-way hashing (BCrypt or equivalent).
- Sensitive data (messages, chores, household membership) is only accessible to authenticated users belonging to relevant households.
- Communications between frontend and backend are intended to run over HTTPS in production to prevent eavesdropping.

Input Validation & API Hardening

- All user input is validated on the server side to prevent injection attacks.
- The API rejects malformed or unauthorized requests with clear error responses.

Together, these measures reduce the risk of unauthorized data exposure, spoofing, and misuse.

7.2 Privacy

Since Household Hub handles family-related information and internal household communications, privacy is essential.

- Minimal data collection: Only information required to support the platform's functionality is collected (user accounts, chores, messages).
- Isolation between households: Users can only see data within their own households. Helpers involved in different households cannot cross-access data.
- Message confidentiality: Chat content is never exposed outside of the participants in that room.
- No tracking or profiling: The system does not track user behavior beyond what is necessary to support task completion.
- Data removal: Future iterations could support user-initiated deletion or household export; the data model already separates data per household, simplifying this.

The overall design respects the expectation that household matters remain private and are not visible to outsiders.

7.3 Accessibility

We aim to make Household Hub usable by a diverse audience, including busy parents, older caregivers, and individuals with accessibility needs.

UI Considerations (present and planned):

- Clear navigation structure with intuitive dashboards and task lists.
- High-contrast color palette to improve readability.
- Semantic HTML tags and proper labeling in React components to support screen readers.
- Large clickable interaction areas for tasks, buttons, and messages.

7.4 Ethical, Legal, and Societal Impacts

Ethical Considerations

- The platform avoids surveillance features or invasive monitoring of caregivers.
- Communication and task history exist for clarity, not evaluation or ranking of helpers.
- The system encourages transparency and fairness in household-caregiver relationships.

Legal Considerations

- The design aligns with common privacy expectations (e.g., GDPR-inspired principles such as minimizing data and restricting access).
- No regulated data (healthcare, financial, or child-specific records) is collected.
- User-generated content remains the property of the user/household, and no automated decision-making is applied.

Societal Impact

- Household Hub supports equitable distribution of household labor by making responsibilities visible and organized.
- It strengthens communication and reduces misunderstandings between families and caregivers.
- By making task coordination simpler, it may reduce stress and improve work–life balance in busy households.

8. Deployment & Operations

Briefly describe how the system is deployed and operated; link to detailed guides in the appendices.

9. Limitations & Future Work

9.1 Current Limitations

Despite delivering the core functionality of Household Hub, several limitations remain due to scope, time constraints, or early architectural choices. Currently there is a complete backend API, but the Raect/Vite frontend is still in progress and not yet completed. Not to mention the authentication system is functional but minimal and the workflows are simplified. While the system handles moderate data flows, it has not been tested under large households with hundreds of tasks, dozens of concurrent WebSocket users, or heavy messaging traffic.

9.2 Technical Debt

During development, several shortcuts were taken to meet deadlines:

- Some services contain logic that ideally belongs to dedicated utility or domain classes.
- Certain DTOs and response objects could be refactored for consistency.
- Repositories rely on default JPA behavior and could benefit from optimized custom queries.
- Configuration (security, WebSockets, CORS) is correct but not yet fully hardened for production.
- Infrastructure scripts (Docker, DB provisioning) are functional but could be simplified or automated.

This technical debt does not affect core functionality but will influence how easily the system can grow.

9.3 Future Work

Expand Task & Household Features

- Add deadlines, recurring tasks, reminders, and priority levels
- Improve helper assignment workflows (approval, scheduling conflicts)
- Add household roles (primary parent, secondary parent, part-time helper)

Enhanced Messaging

- Add message persistence optimizations
- Implement read receipts and typing indicators
- Add offline support and reconnection logic
- Support file and image sharing (e.g., photos of completed tasks)

Reliability & Security Enhancements

- Harden authentication (2FA, email verification, recovery flow)

- Add rate limiting to API endpoints
- Improve WebSocket authorization and token renewal
- Add audit logging for household activity

Automated Testing & CI/CD

- Unit tests for services and controllers
- Integration tests for household-task workflows
- WebSocket message tests
- UI e2e testing with frameworks like Cypress or Playwright
- CI pipeline using GitHub Actions for automatic builds and tests

Deployment & Scalability

- Deploy backend and database to a cloud platform
- Add container orchestration (Docker Compose → Kubernetes)
- Run load tests to evaluate WebSocket and DB performance
- Add database migration tooling (Flyway/Liquibase)

9.4 Long Term Vision

In future iterations, Household Hub could evolve into a comprehensive household–caregiver ecosystem with:

- A polished mobile app (React Native or Flutter)
- User reviews and reputation scoring (ethically designed)
- Integration with calendars (Google Calendar / iCal)
- Payment handling for care services (Stripe, Square)
- AI-driven task suggestions or scheduling optimization

These improvements would strengthen the platform’s value for modern households and expand the system beyond a capstone project into a usable real-world service.

10. References

1. Spring Framework. *Spring Boot Reference Documentation*. Spring.io. Available: <https://spring.io/projects/spring-boot>
2. Spring Framework. *Spring WebSocket Documentation*. Spring.io. Available: <https://docs.spring.io/spring-framework/reference/web/websocket>
3. PostgreSQL Global Development Group. *PostgreSQL Documentation*. Available: <https://www.postgresql.org/docs/>
4. React Team. *React Documentation*. Meta Platforms. Available: <https://react.dev>
5. Vite Team. *Vite: Next Generation Frontend Tooling*. Available: <https://vitejs.dev>
6. Docker Inc. *Docker Documentation*. Available: <https://docs.docker.com/>

7. MDN Web Docs. *WebSockets — MDN Documentation*. Mozilla. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
8. Martin Fowler. *Repository Pattern*. martinfowler.com. Available: <https://martinfowler.com/eaCatalog/repository.html>
9. ISO/IEC. *Web Content Accessibility Guidelines (WCAG) 2.1*. Available: <https://www.w3.org/TR/WCAG21/>
10. Oracle. *Java Platform, Standard Edition Documentation*. Available: <https://docs.oracle.com/en/java/>
11. GitHub Documentation. *GitHub Actions CI/CD*. Available: <https://docs.github.com/en/actions>