# CareFlow Nexus

## Bed Turnover Automation Prototype

Technical Specification & Build Manual

Team: Team Temp

Version: 1.0

# Contents

# 1 Overview

CareFlow Nexus is a prototype demonstrating an autonomous bed turnover workflow in a hospital using agentic AI. The system simulates how minimal human input can trigger a full hospital operational sequence — from patient admission to bed availability — using intelligent automated agents.

The prototype focuses solely on operational automation rather than medical decision-making. Once a doctor initiates an admission request, agents assign resources, schedule recurring maintenance tasks, and coordinate cleaning and nursing workloads.

Primary goal:

- Showcase a functioning AI-coordinated workflow with observable end-to-end automation of the bed turnover process.

# 2 MVP Scope

The MVP implements:

- One simulated hospital unit.
- 5–10 predefined beds, nurses, and cleaners.
- A full workflow: admission $\rightarrow$ occupancy $\rightarrow$ recurring tasks $\rightarrow$ discharge $\rightarrow$ reset (bed available again).
- Automated assignments for:
  - Bed allocation.
  - Pre-admission cleaning.
  - Nurse assignment.
  - Recurring nurse check-ups.
  - Recurring daily cleaning.
  - Post-discharge cleaning.

Out of scope:

- Authentication and user identity management.
- Diagnosis or treatment decision-making.
- Billing, EMR systems, or cross-department workflows.

# 3 Architecture Summary

## 3.1 Technology Stack

- **Frontend:** Next.js deployed on Vercel.

- **Backend:** FastAPI deployed on Railway.
- **Database:** Supabase (PostgreSQL).
- **Agents:** Hugging Face Spaces (Python worker process).
- **LLM:** Groq API (Llama model).

## 3.2 Key Design Principles

- The backend is the only component that writes or modifies database state.
- The frontend reads system state directly from Supabase (via realtime or polling).
- Agents never access the database directly; all communication is via backend APIs.
- WebSockets are used for push notifications to agents, backed by periodic polling for reliability.
- Agents run remotely on Hugging Face Spaces and communicate with the backend through secure HTTPS polling and optional WebSocket subscription.

## 3.3 Workflow Transition Logic (Backend Responsibility)

**Workflow Entry Rule:** When `POST /patients/{id}/admit` is called, the backend **must** create an initial `bed_assignment` task (status: `pending`) assigned to the `MASTER` agent. This task begins the automated workflow and ensures the system does not stall waiting for an initial trigger.

Agents do not create new tasks directly. Instead, the backend enforces workflow progression through automatic follow-up task generation when certain actions are completed.

Example workflow rules:

- **On patient admission:**
- When `POST/patients/{id}/admit` is called, the backend must immediately create a `bed_assignment` task (status: `pending`) assigned to the MASTER agent.
- No other workflow actions occur until this task is processed.

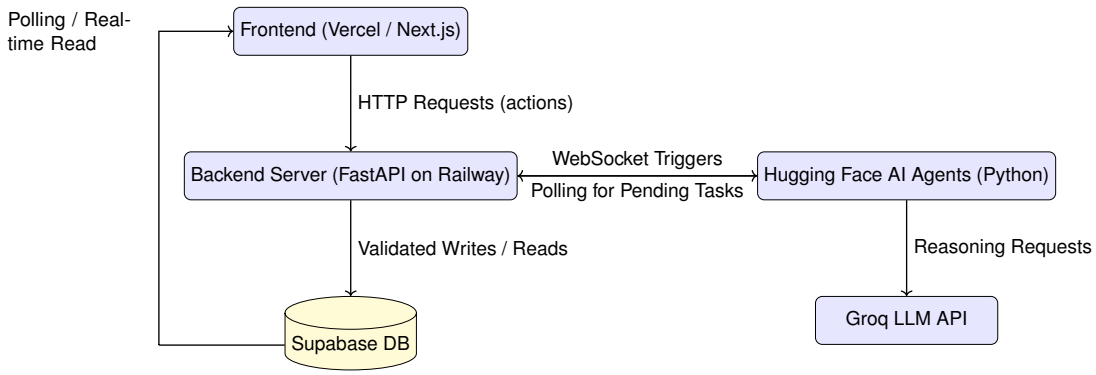
- **After a MASTER agent assigns a bed:**
  - Update admission record.
  - Update patient and bed states.
  - Automatically create a pending `cleaning` task for the Cleaner Agent.
- **After a Cleaner marks pre-admission cleaning complete:**
  - Bed becomes `occupied`.

- Create a pending `nurse_assignment` task for the Nurse Agent.

- **After nurse assignment:**
  - Backend schedules recurring nurse checkup and cleaning tasks.
- **After discharge:**
  - Cancel or mark as `failed` any currently `pending`, `assigned`, or `locked` tasks for this patient to prevent leftover recurring or invalid actions.
  - Create a pending `post_discharge_cleaning` task for the Cleaner Agent.
  - Update patient status to `discharge_requested`.
  - Mark the bed as `available` only after final cleaning is completed.

This ensures that agents operate statelessly and workflow progression is reliable and deterministic.

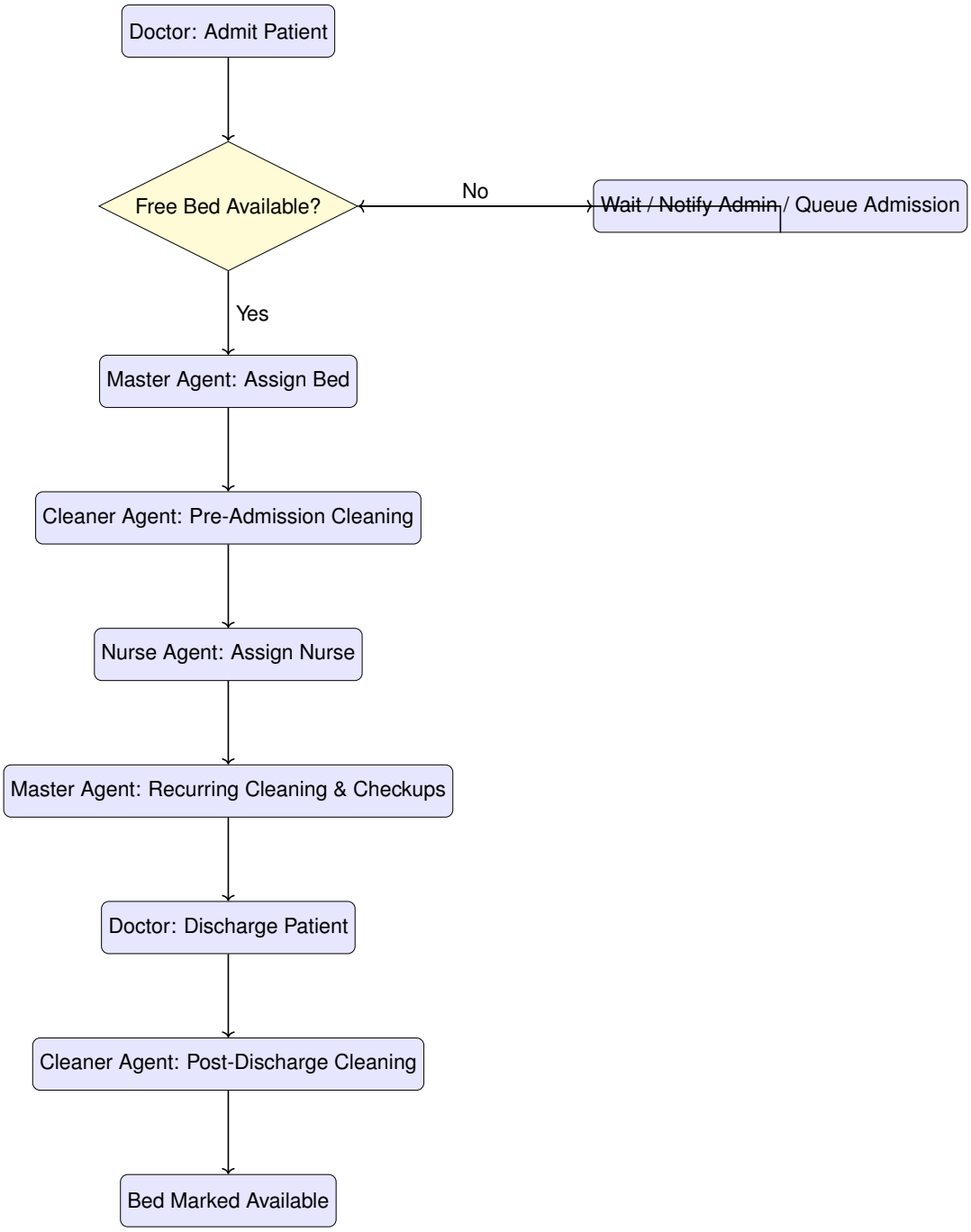
# 4 Interaction Architecture Diagram




*Note: The AI Agents block represents a Hugging Face-hosted worker service rather than a local runtime.*

# 5 Workflow Flowchart

## 5.1 High-Level Bed Turnover Flow




The same cleaning workflow can be triggered mid-stay by a nurse or doctor through a generic *request cleaning* action on a specific bed.

# 6 Roles and System Responsibilities

## 6.1 User Roles

- **Doctor**
  - Initiates the patient workflow by admitting a patient.
  - Triggers discharge once treatment is considered complete.
  - Does not manually assign beds, nurses, or cleaners.
- **Nurse**
  - Receives and accepts nurse assignment tasks.
  - Marks nursing tasks as completed.
  - Can trigger additional cleaning for a bed if needed.
- **Cleaner**
  - Receives cleaning tasks (pre-admission, recurring, and post-discharge).
  - Marks cleaning tasks as completed.
- **Admin**
  - Observes overall system state, including bed usage and task queues.
  - Has no direct workflow mutation actions in the MVP.

## 6.2 Agents

- **Master Agent**
  - Reacts only to admission events. Discharge handling bypasses the Master Agent and directly creates a `post_discharge_cleaning` task for the Cleaner Agent.
  - Orchestrates workflow steps (bed assignment, recurring tasks).
- **Nurse Agent**
  - Assigns an available nurse to an admitted patient.
  - Balances workload across nurses.
- **Cleaner Agent**
  - Assigns cleaners to cleaning tasks.
  - Handles pre-admission, recurring, and post-discharge cleaning.
  - Automatically receives post-discharge cleaning tasks when a doctor triggers discharge.

# 7 Database Schema

The database follows an event-driven workflow model where state is derived rather than duplicated. To avoid circular dependencies and ensure clean workflow updates, a dedicated **admissions** table is used as the single source of truth linking patients and beds.

## 7.1 Tables

**patients**

- id (PK)
- name (text)
- status (enum: `pending_bed`, `awaiting_cleaning`, `assigned`, `under_care`, `discharge_requested`, `discharged`)
- created_at (timestamp)

**beds**

- id (PK)
- status (enum: `available`, `pending_cleaning`, `cleaning_in_progress`, `occupied`)
- type (optional: ICU, ward, general) – future scalability
- created_at (timestamp)

**admissions**

- id (PK)
- patient_id (FK)
- bed_id (FK)
- admitted_at (timestamp)
- discharged_at (timestamp, nullable)
- status (enum: active, pending, completed)

**staff**

- id (PK)
- name (text)
- role (enum: doctor, nurse, cleaner)

*Note:* The staff workload value is **not stored** in the database. It is computed dynamically by counting non-completed tasks assigned to each staff member at runtime:

$$workload = COUNT(tasks\ WHERE\ status \neq' completed')$$

Agents use this value to perform fair workload balancing.

**tasks**

- id (PK)
- type (enum: `cleaning`, `nurse_checkup`, `bed_assignment`, `nurse_assignment`, `post_discharge_cleaning`)
- target_role (enum: AGENT, HUMAN)
- agent_role (enum: MASTER, CLEANER, NURSE, nullable)
- assigned_to (FK to staff, nullable for agent tasks)
- patient_id (FK nullable)
- bed_id (FK)
- status (enum: pending, locked, assigned, accepted, in_progress, completed, failed)
- scheduled_at (timestamp)
- created_at (timestamp)

**events**

- id (PK)
- type (enum: `admission_requested`, `cleaning_required`, `nurse_assignment_requir` `discharge_requested`, `scheduled_event`)
- payload (JSON)
- processed (boolean)
- created_at (timestamp)

# 8 Naming and State Conventions

A unified naming system is used across the backend, frontend, database, and AI agent decisions.

## 8.1 Core Entities

- **patient**
- **bed**
- **task**
- **event**
- **staff**

All identifiers follow `snake_case` for JSON and database values.

## 8.2 Bed States

- `available`
- `pending_cleaning`
- `cleaning_in_progress`
- `occupied`

## 8.3 Patient States

- pending_bed
- awaiting_cleaning
- assigned
- under_care
- discharge_requested
- discharged

## 8.4 Task Types

- cleaning
- nurse_checkup
- bed_assignment
- nurse_assignment
- post_discharge_cleaning

## 8.5 Task States

- pending
- locked
- assigned
- accepted
- in_progress
- completed
- failed

## 8.6 Event Types

- admission_requested
- bed_assignment_required
- cleaning_required
- nurse_assignment_required
- scheduled_cleaning_trigger
- scheduled_nurse_checkup_trigger
- discharge_requested
- post_discharge_cleaning

## 8.7 Agent Roles

- MASTER
- CLEANER
- NURSE

### 8.8 Standard Action Keywords (LLM Output)

- `assign_bed`
- `assign_cleaner`
- `assign_nurse`
- `schedule_cleaning`
- `schedule_checkup`

Example LLM decision JSON:

```
{
  "action": "assign_cleaner",
  "bed_id": 2
}
```

*Note: Agents never select specific staff members. They only determine the action and target resource (bed or patient). The backend assigns a human staff member based on current workload and role availability.*

## 9 API Contract

All endpoints are assumed to be prefixed with `/api` in deployment (e.g., `/api/patients/1/admit`).

### 9.1 Frontend-Initiated Actions

| Method | Endpoint | Used By | Description |
|---|---|---|---|
| POST | `/patients` | Doctor | Create a new patient record. |
| POST | `/patients/{id}/admit` | Doctor | Start admission workflow for patient {id}. |
| POST | `/admissions/{id}/cancel` | Admin/Doctor | Abort a stuck or invalid admission and reset bed/task state. |
| POST | `/beds/{id}/clean` | Doctor/Nurse | Request cleaning for bed {id} (generic trigger). |
| POST | `/tasks/{id}/accept` | Nurse/Cleaner | Mark task {id} as accepted by staff. |
| POST | `/tasks/{id}/complete` | Nurse/Cleaner | Mark task {id} as completed. |

Example: Admit patient

**POST /api/patients**

```
{
  "name": "John Doe"
}
```

**POST /api/patients/1/admit**

```
(no JSON body required for MVP)
```

## 9.2 Agent-Facing Endpoints

| Method | Endpoint | Used By | Description |
|---|---|---|---|
| GET | `/tasks/pending?role={ROLE}&include=context` | Agents | Poll pending tasks & receive decision context (MASTER/NURSE/CLEANER). |
| POST | `/agent/tasks/{id}/decision` | Agents | Submit structured decision for task {id}. |

Example: Agent polling

**GET /api/tasks/pending?role=MASTER&include=context**

```
[
  {
    "task_id": 12,
    "type": "bed_assignment",
    "patient_id": 1,
    "bed_id": null,
    "status": "pending",
    "options": {
      "available_beds": [1, 3, 4]
    }
  }
]
```

Example: Agent decision

**POST /api/agent/tasks/12/decision**

```
{
  "action": "assign_bed",
  "bed_id": 3
}
```

The backend immediately marks tasks as `locked` when they are sent to an agent, to prevent two agents from handling the same task. If an agent fails or times out, the task can be reset back to `pending`.

### 9.3 No-Option Handling Rule

If the backend returns a valid task but the associated context has no available resources (e.g., no available beds or nurses), the agent must return a defer response:

```
{
  "action": "defer",
  "reason": "no_resources_available"
}
```

The backend must then:

- Unlock the task (status returns to `pending`)
- Add a retry timestamp (minimum retry delay recommended: 5–10 minutes)

## 10 Screens and Expected Behavior

Each role has a dedicated dashboard. All dashboards share:

- A header showing the current role.
- A view of relevant beds/tasks.
- Buttons that map directly to the API actions defined above.

### 10.1 Admin Screen

- Bed grid with color-coded status (available, occupied, cleaning).
- Task queue overview (all open tasks).
- Activity log showing recent events (admissions, assignments, cleaning, discharges).
- Staff list with role and availability.
- A button labeled `Trigger Scheduled Events` that calls `POST/debug/trigger-schedule`. This endpoint generates scheduled cleaning and nurse checkup tasks for all currently occupied beds. This mechanism replaces automatic cron-based scheduling for the MVP and ensures recurring task flow can be demonstrated manually.

### 10.2 Doctor Screen

- Button: `Admit Patient` (opens simple name input form).
- Table of admitted patients including:
  - Patient Name

- Bed Number
- Assigned Nurse
- Current Status
- `Discharge` button

### 10.3 Nurse Screen

- List of tasks assigned to the logged-in nurse.
- For each task:
  - Task Type (e.g., nurse_checkup).
  - Patient and Bed information.
  - Buttons: `Accept`, `Mark Completed`.
  - For certain tasks, a `Request Cleaning` button that triggers `POST/beds/{id}/clean`.

### 10.4 Cleaner Screen

- List of cleaning tasks (pre-admission, recurring, and post-discharge).
- For each task:
  - Bed Number.
  - Trigger Type (e.g., pre-admission, scheduled, post-discharge).
  - Button: `Mark Cleaned` (maps to `/tasks/{id}/complete`).

## 11 Team Allocation

- **Backend & Agents (2 members)**
  - Implement FastAPI endpoints.
  - Implement event $\rightarrow$ task generation logic.
  - Implement Python agent loops and Groq integration.
- **Frontend (1 member)**
  - Build Next.js UI for all role dashboards.
  - Implement polling/subscriptions for state updates.
  - Wire buttons to the defined API endpoints.
- **Database & Testing (1 member)**
  - Configure Supabase schema.
  - Seed with test data (beds, staff, sample patients).
  - Run end-to-end workflow tests and fix inconsistent states.

# 12 Build & Run Instructions

## 12.1 Suggested Bring-Up Order

1. Create tables in Supabase matching Section 6.
2. Start the backend (FastAPI) and verify all endpoints using a tool like curl/Postman.
3. Deploy and run agents on Hugging Face Spaces (Python background runner) and confirm they can:
    - Poll `/tasks/pending?role={ROLE}&include=context`.
    - Call Groq API.
    - Send decisions to `/agent/tasks/{id}/decision`.
4. Deploy frontend to Vercel and point it at the backend and Supabase.

## 12.2 Testing Checklist

- Admit a patient and verify a bed is assigned by the Master Agent.
- Verify pre-admission cleaning and nurse assignment tasks appear and can be completed.
- Confirm recurring cleaning and nurse checkup tasks appear over time (or via manual triggers for the demo).
- Discharge the patient and verify post-discharge cleaning and bed availability.

# 13 Development Testing Aids (Drivers & Stubs)

Because the system is distributed across three independently developed components (Agents, Backend, and Frontend), temporary testing scaffolding is required to allow development in parallel. This section defines the use of drivers and stubs to simulate missing components during the build phase.

## 13.1 Stubs

A **stub** simulates a dependent component that is not yet implemented or not currently available.

Expected usage:

- When the Backend is not yet generating tasks, the Agents may call a stub endpoint returning predefined task JSON payloads.
- When Agents are offline, the Backend may stub a valid response to simulate a completed agent decision.

- The Frontend may request data from stubbed API routes before real database state is connected.

Example stub output for development:

```
{
  "task_id": 99,
  "type": "bed_assignment",
  "patient_id": 1,
  "options": { "available_beds": [1,2,3] }
}
```

## 13.2 Drivers

A **driver** is a test harness that forces interactions forward without the real user or agent.

Drivers allow developers to trigger workflow actions without UI or Agent logic.

Examples:

- A backend CLI script (‘driver_admit.py‘) that calls:

  ```
  POST /patients/1/admit
  ```

- A driver to simulate agent behavior:

  ```
  POST /agent/tasks/99/decision
  {
    "action": "assign_bed",
    "bed_id": 2
  }
  ```

- A driver button in the Admin UI: `Run Agent Logic Once`

## 13.3 Stub/Driver Toggle Flags

For consistency across the team, all stubs and drivers must be controlled using:

- `USE_STUBS = true/false`
- `USE_DRIVERS = true/false`

When disabled, all components must run using real logic.

## 13.4 Testing Responsibility Matrix

| Role | Uses Stubs? | Uses Drivers? | Purpose |
|---|---|---|---|
| Agent Dev | Yes (task polling stub) | Yes | Validate reasoning logic |
| Backend Dev | No after DB ready | Yes | Force workflow transitions |
| Frontend Dev | Yes (mock API) | Optional | UI development without backend/live agents |

### 13.5 Removal Rule

Before final deployment or demo, all stub and driver code must be:

- Disabled in configuration,
- Marked with TODO tags,
- Verified not called during live workflow.

This ensures the functioning system demonstrates actual automation rather than scripted behavior.

# A Appendix A: JSON Formats and Logs

Example event payload (stored in `events.payload`):

```
{
  "patient_id": 1,
  "requested_by": "doctor",
  "timestamp": "2025-11-26T10:30:00Z"
}
```

Example task object returned to agents:

```
{
  "task_id": 21,
  "type": "cleaning",
  "bed_id": 5,
  "patient_id": 1,
  "status": "pending",
  "options": {
    "available_beds": [5],
    "priority": "high"
  }
}
```

Example log line (for internal debugging):

```
[2025-11-26T10:32:10Z] MASTER assigned bed 5 to patient 1
```

# B Appendix B: Example State Transition

- Initial:

  - patient.status = `pending_bed`
  - bed.status = `available`

- After bed assignment:
  - patient.status = `awaiting_cleaning`
  - bed.status = `pending_cleaning`

- After pre-cleaning:
  - patient.status = `under_care`
  - bed.status = `occupied`

- After discharge and final cleaning:
  - patient.status = `discharged`
  - bed.status = `available`

# C Hugging Face Agent Runtime Notes

**Important Hosting Note (Hugging Face Spaces)**

Free-tier Hugging Face Spaces may sleep after periods of inactivity, even if heartbeat logs or background loops are running. For a hackathon demo, it is recommended to:

- Manually open the Space in a browser 5–10 minutes before the demo.
- Verify that agents are successfully polling tasks and sending decisions.
- Be prepared to restart the Space if it has gone idle.

This runtime model is acceptable for a prototype and live demonstration, but a production system would require a more persistent hosting approach.

*This document defines the implementation blueprint for the CareFlow Nexus prototype.*

# D Appendix C: Developer Workflow Checklist

## Backend

- Validate all incoming agent decisions.
- Update database state according to workflow rules.
- Generate follow-up tasks automatically.
- Enforce defer handling and retry timers.

## Agent Runtime

- Poll tasks by role.
- Parse backend context (beds, staff, workload).
- Produce minimal JSON actions.
- If no valid options exist → return `defer`.

## Frontend

- Poll state from Supabase.
- Update UI reactively.
- Trigger workflows only through defined endpoints.