ChatGPT

Product Upload Workflow Management System – Architecture & Implementation Plan

Introduction

This document outlines a comprehensive architecture and feature breakdown for a web application that manages and tracks product upload workflows for a Shopify store. The system will streamline how a small team (Super Admin, Warehouse Manager, Editors, Auditor) collaborates to take new products from arrival to live on the Shopify store. It will be deployed on Replit, using a Node.js/Express backend, a React frontend, and a PostgreSQL database for persistence. Key features include task creation from incoming product shipments, automatic breakdown into per-product tasks, role-based task assignment, a structured workflow with enforced state transitions, detailed time tracking (lead time, cycle time, etc.), quality checklists, dashboards for each role, SLA monitoring with alerts, and visual reports (e.g. control charts and cumulative flow diagrams) to identify bottlenecks and aging work-in-progress.

System Roles and Permissions

- **Super Admin (Owner):** Full access to all features and data. The Super Admin can configure the system, create and assign tasks, and view all dashboards. They have a birds-eye view of team productivity, workflow bottlenecks, and aging WIP (work in progress). The Super Admin can perform any state transition on tasks as needed.
- **Warehouse Manager (WM):** Responsible for intake of new products (e.g. based on Purchase Orders or shipments). The WM (and Super Admin) can create new "to-do" entries for each received batch of products, capturing vendor name, received date, and order number. The WM oversees task triage and assignment to Editors, monitors active product uploads, and manages the review queue. They have a dashboard focusing on active to-dos/POs, tasks awaiting review, and service-level agreement (SLA) timers (e.g. how long tasks have waited to be assigned or reviewed). The WM can change task states up through review and publication (but cannot modify content details like an Editor can).
- **Editor:** Editors are content creators who fulfill the product upload tasks. They can view unassigned tasks, self-assign tasks or be assigned by the WM, and then move tasks through **ASSIGNED** → **IN_PROGRESS** → **READY_FOR_REVIEW** states by entering product details and ensuring checklist items are completed. Editors only see tasks relevant to them (or available to claim) and have an Editor dashboard listing their assigned tasks, tasks in progress, and any deadlines or SLA alerts. They have permission to edit product information fields and checklist items on their tasks, but cannot mark tasks as published or done without review.
- **Auditor:** A read-only role that can view all tasks, dashboards, and an audit log of all actions and state changes. The Auditor's goal is to supervise accountability – they **cannot** modify tasks or change statuses, but they have access to a comprehensive history of who changed each task's state and when. This ensures transparency and helps enforce process compliance. The Auditor might, for example, review if SLAs are being met and if any tasks are stuck, but any intervention must be done by other roles. The Auditor's view may include an **Activity Log** dashboard showing each state transition, timestamp, and the user who performed it.

Workflow States and Task Lifecycle

Each product task progresses through a defined sequence of workflow states. The application uses a **finite state machine** approach to enforce valid transitions and maintain data consistency [1]. The workflow states (with allowed transitions) are:

- **NEW:** Initial state when a to-do is created and its child product tasks are generated. This indicates a new product awaiting triage. Only the WM or Super Admin can create tasks in the NEW state. Next step: moved to TRIAGE by WM/Admin after initial review.
- **TRIAGE:** Indicates the WM (or admin) has acknowledged the new task and possibly added details (e.g. priority or product grouping). In TRIAGE, the task is ready to be assigned to an Editor. Next step: **ASSIGNED** (WM/Admin assigns to an Editor, or an Editor self-assigns, moving it to ASSIGNED).
- **ASSIGNED:** The task is now owned by a specific Editor but work has not started yet. In this state, the Editor or WM can transition it to **IN_PROGRESS** when actual content entry begins. (If an Editor self-assigns directly from NEW, the system may skip TRIAGE or mark it as triaged and assigned in one step.)
- **IN_PROGRESS:** The Editor is actively working on this product (entering Shopify fields, uploading images, etc.). The task remains in progress until the Editor believes it meets the Definition of Done (checklist complete). Next step: **READY_FOR_REVIEW**. The Editor moves it to READY_FOR_REVIEW when the product listing is fully prepared.
- **READY_FOR_REVIEW:** The task is awaiting review by the WM or Super Admin. At this stage, the Editor can no longer edit the content (unless it's sent back). The reviewer checks the product details and quality. Two outcomes are possible from here:
- **PUBLISHED:** If the review is successful (all criteria met), the reviewer marks the task as Published. This implies the product is now live on Shopify (the actual publication to Shopify might be a manual step done by the reviewer, but the state indicates it's published). Transition to PUBLISHED likely triggers the system to record the publish timestamp. Next step: **QA_APPROVED** (a QA verification step).
- **CHANGES_REQUESTED:** If the reviewer finds issues, they transition the task to Changes Requested. This state triggers a feedback loop (often denoted by a ↺ loop in the workflow) – the task goes back to the Editor for corrections. The Editor will address the feedback and then move the task back to READY_FOR_REVIEW. (The system may treat CHANGES_REQUESTED as a sub-state of review; for simplicity, we include it as a state that loops back to in-progress work.)
- **CHANGES_REQUESTED:** A reviewer has requested modifications. The task is effectively "on hold" for the Editor to address issues. The Editor gains edit access again and will move the task back to **IN_PROGRESS** (or directly to **IN_PROGRESS**/**ASSIGNED** state) to make changes. After making fixes, the Editor again marks **READY_FOR_REVIEW**. (This loop can repeat until the reviewer is satisfied. The system will log each cycle of review and rework for audit purposes.)
- **PUBLISHED:** Indicates the product has been published to the Shopify store. At this point, the content is live. However, the workflow isn't completely done – a final quality assurance check can occur. Next step: **QA_APPROVED**. (In some teams, publishing might automatically imply QA approval, but here we have an explicit QA state for oversight.)
- **QA_APPROVED:** The product listing has been verified post-publication (e.g. the Auditor or another QA responsible person has reviewed the live product to ensure everything is correct – no typos, all images looking good, SEO metadata properly showing, etc.). The Auditor (or WM/Admin performing QA) marks the task as QA_APPROVED. Next step: **DONE**.

- **DONE:** Final state, indicating the product upload task is fully completed, with all checks passed. No further action is needed, and the task is closed. (No transitions out of DONE – it's a terminal state.)

State Transition Rules and Enforcement: The backend will strictly enforce these transitions as a state machine to prevent invalid state jumps. For example, a task cannot go from NEW directly to IN_PROGRESS without being assigned, and it cannot skip the review to directly Done. Each transition will be allowed only if the task's current state and the user's role meet the prerequisites (this is implemented via transition "guards" and permissions [2]). For instance, only a WM or Admin can move a task from NEW to TRIAGE or ASSIGNED; only the assigned Editor can move a task from ASSIGNED to IN_PROGRESS and then to READY_FOR_REVIEW; only a WM/Admin reviewer can move from READY_FOR_REVIEW to PUBLISHED or CHANGES_REQUESTED, etc. These guard conditions (role checks and state prerequisites) ensure **data integrity by preventing invalid state changes** [1]. All states and transitions will be represented in code (e.g. using an enumeration for states and a mapping of allowed transitions). This explicit workflow definition makes the process clear and testable, and by enforcing valid transitions we maintain consistency [1].

Audit Trail: Each state change triggers an entry in a TaskStatusHistory log with a timestamp and the user who made the change. This provides a detailed audit trail of the task's lifecycle. State machine enforcement combined with this log makes the workflow **auditable** – we can always review who moved a task to what state and when [3]. The Auditor role will have access to this history for accountability.

Time Tracking: As part of each status change, the system captures timestamps to compute key performance metrics. We will record the creation time, the time of first start (when it enters IN_PROGRESS), and the completion time (when it reaches DONE). Additionally, each intermediate transition time is logged (e.g. assigned time, review start, publish time, etc.). These timestamps allow calculation of:

- **Lead Time:** The total time from task creation (NEW) to final completion (DONE). This measures the end-to-end throughput of the workflow for each product [4]. Lead time includes all waiting and processing time.
- **Cycle Time:** The active working time from when work actually started to when it completed [5]. In our context, cycle time can be defined from the moment an Editor begins work (e.g. enters IN_PROGRESS) to the moment the task is done. This excludes the initial waiting time before an editor picked it up. (By tracking both lead and cycle time, we can distinguish delays in queue vs. actual work time [6] [7].)
- **Time Since Creation / Aging:** For open tasks, we will compute how long it's been since creation (current time minus creation timestamp). This is used to identify aging work items that might be at risk of falling behind.
- **Time Since Assignment:** For tasks that have been assigned but not completed, we track how much time has passed since assignment. This helps monitor if an Editor has been holding a task for too long without completion. Similarly, we might track time in each status (e.g. how long a task has been in Ready for Review, how long in In Progress) to pinpoint bottlenecks.

These time metrics will feed into SLA monitoring and dashboard visuals.

Data Model (PostgreSQL Schema)

The system will use a normalized relational schema in PostgreSQL to store users, tasks, and related workflow data. The key tables and their schemas include:

- **Users:** Stores user accounts and role information. Fields: `user_id` (PK), `name`, `email` (for login), `password_hash`, `role` (enum or FK to Roles table). Roles can be enumerated as "SuperAdmin", "WarehouseManager", "Editor", "Auditor". (We may also use a separate Roles reference table to normalize role definitions and permissions.) Each user's role determines their access and what actions they can perform in the application.
- **ToDos (ProductIntake):** Represents a high-level to-do entry, typically corresponding to a Purchase Order or shipment containing multiple products. Fields: `todo_id` (PK), `vendor_name`, `order_number`, `received_date`, `created_by` (who logged it, FK to Users), `created_at`. The ToDo provides a grouping for child product tasks. For example, if 10 new products arrived from vendor X under order #123, the WM creates one ToDo with those details, and the system will generate 10 child tasks (one per product/SKU). This table helps track product batches and allows the WM to see which POs are still in progress.
- **Tasks (ProductTasks):** Represents the unit of work for each product or SKU to be listed. Fields: `task_id` (PK), `todo_id` (FK to ToDos, allowing grouping), `product_name` (or placeholder name if not known initially), `sku` (if available), `assigned_to` (FK to Users, nullable if unassigned), `current_status` (enum of the workflow state), and timestamp fields for key events (e.g. `created_at`, `assigned_at`, `started_at`, `ready_for_review_at`, `published_at`, `done_at`). The task also includes fields for core product data that Editors will fill in: e.g. `title`, `description`, `price`, `tags`, `barcode`, etc., as well as maybe a JSON field or related table for variants if needed. Each task is one product listing to be created on Shopify. Storing the required product fields in the task table itself (or a related ProductDetails table) ensures Editors can input all necessary info. We keep `current_status` for quick filtering, but rely on the TaskStatusHistory for the full transition log.
- **TaskStatusHistory:** A log of every state change for tasks. Fields: `log_id` (PK), `task_id` (FK to Tasks), `from_status`, `to_status`, `changed_by` (FK to Users), `changed_at` (timestamp). Every time a task moves to a new workflow stage, a record is inserted. This provides the detailed timeline for auditing and for calculating how long the task spent in each status. For example, by querying this table we can find when a task entered and exited "IN_PROGRESS" to compute its in-progress duration, or how long it waited in "READY_FOR_REVIEW". This table enables building the control charts and other visualizations (since we have timestamps for each stage transition) and is essential for compliance (tracking who did what).
- **ChecklistItems:** Represents the checklist of required subtasks/fields for each product task. We will implement a product upload checklist to define the Definition of Done. This could be modeled in two ways:
- A **Checklist Template** table defining all possible checklist items (e.g. "Title entered", "Description written", "At least one image uploaded", "Images meet quality criteria", "SEO meta title set", etc.), and a **TaskChecklist** table linking tasks to these items with a completion status. For instance, `ChecklistItemTemplates(id, description, mandatory)` and `TaskChecklist(task_id, item_id, is_completed, completed_by)`. When a new task is created, the system populates TaskChecklist with all required items (marking them incomplete).
- Alternatively, include checklist booleans as columns in the Tasks table (e.g. `title_done`, `desc_done`, `images_done`, etc.), but that is less flexible. The normalized approach with a

TaskChecklist table is better for extensibility and tracking exactly which items are done. Each checklist item can store additional info like `validated_at` or details (e.g. an image resolution value). The **Definition of Done enforcement** will query these items. For example, all mandatory items must be `is_completed = true` to allow a task to transition to Ready for Review or Published. This enforcement can be done via a database constraint or (more flexibly) in application logic. (Inspiration is taken from how Jira can enforce checklist completion via workflow validators [8] – if a user tries to transition an issue without completing all mandatory checklist items, the transition is blocked [8].)

- **Comments/Feedback (optional):** To facilitate communication on changes requested, we may have a `TaskComments` table: `comment_id, task_id, author_id, comment_text, timestamp`. This allows reviewers to leave notes when requesting changes, and Editors to clarify updates. This is not explicitly requested but would enhance the workflow loop for CHANGES_REQUESTED.
- **SLA Settings (optional):** If we want to store SLA thresholds in the DB (instead of hardcoding), a table like `SLA_Settings` could define time limits for each metric (e.g. `max_assign_time`, `max_review_time`, etc.). However, given a small team and likely fixed SLAs, this can be hardcoded or configured by an Admin in a settings page.

All tables will use foreign keys to maintain referential integrity (e.g. if a User is deleted or deactivated, their tasks and history remain but maybe marked as such). The schema is fully normalized to avoid data duplication. For example, vendor name and order number are stored at the ToDo level and not repeated in each task (they can be joined when needed). The use of separate tables for status history and checklist items adheres to normalization and makes the system flexible and consistent.

We will add necessary indexes (e.g. index on Tasks by `current_status` for quick filtering active vs done tasks, index on TaskChecklist by task_id, etc.). PostgreSQL provides robust support for these relational constraints and indexing, ensuring good performance even as the number of tasks grows.

Application Architecture and Technology Stack

The application follows a classic **three-tier architecture** (presentation, application logic, data) with clear separation of concerns:

- **Frontend:** A single-page application (SPA) built with **React**. This will provide a dynamic, responsive user interface for all roles. The React app will handle routing for different views (dashboards, task detail pages, etc.) and manage state locally (with Context or a state management library like Redux) for things like the current user's session, their assigned tasks list, etc. The UI will communicate with the backend via RESTful API calls (using `fetch` or Axios). Being on Replit, the static files can be served by the Express server, or we can use Replit's configuration to run the React dev server separately during development. For production on Replit, we will likely build the React app and let Express serve the static build files. Role-based access will be enforced on the client-side by routing (e.g. certain routes/components only render for certain roles) **and** on the server-side by API permission checks. This double layer ensures both usability and security.

- **Backend:** A **Node.js + Express** application serving a REST API. The backend encapsulates all business logic: creating tasks, validating state transitions, enforcing permissions, and performing database operations (using PostgreSQL). We will implement structured endpoints such as:

- `POST /todos` – Create a new to-do (requires WM or Admin). This handler will create the ToDo record and the associated Tasks for each product. The request might include a list of product identifiers or a number of products. For example, if the WM scans a PO and knows it has N products, they might send N and some basic info to generate placeholder tasks. Alternatively, they might upload a CSV of SKUs – but that's out of scope unless needed. The simplest approach is after creating the to-do, the UI directs the WM to add individual product tasks under it.
- `POST /tasks` (or included in the above) – Add a new product task (if tasks can be added one by one to a to-do). This requires specifying which ToDo it belongs to and initial details like SKU or name.
- `GET /tasks` – Retrieve tasks list, with query parameters to filter (e.g. `?status=NEW` or `?assignee=me`). Editors might fetch `GET /tasks?assignee=me&status!=DONE` for their current tasks. The WM might fetch `?status=READY_FOR_REVIEW` for the review queue. Pagination may not be critical for a small team, but we can implement basic paging if needed.
- `GET /tasks/:id` – Get detailed info on a specific task (including product fields and checklist status). This is used when an Editor opens the task to work on it, or a reviewer opens it to review.
- `PUT /tasks/:id` or specific sub-routes for status changes:
    - `PUT /tasks/:id/assign` – Assign a task to an editor (body contains `assignee_id`). This sets `assigned_to` and transitions status from NEW/TRIAGE to ASSIGNED (with `assigned_at` timestamp). Only WM/Admin or an Editor self-assigning to themselves is allowed. On assignment, the backend will check that the task is currently unassigned and in the right state.
    - `PUT /tasks/:id/start` – Mark task as IN_PROGRESS (this could also be combined with assign if self-assign implies immediate start). When an Editor clicks "Start Work", this endpoint sets status to IN_PROGRESS and logs `started_at`. It will verify that `assigned_to` is the current user (or that the user is allowed to start it).
    - `PUT /tasks/:id/complete` – Editor marking task ready for review. This will actually set status to READY_FOR_REVIEW. But before doing so, the server will **validate the Definition of Done**: it checks all required fields are filled and all mandatory checklist items for the task are completed. If anything is missing, the API returns an error and does **not** change the status. This ensures that a task cannot leave IN_PROGRESS until it meets the quality bar (similar to a Jira workflow validator that all DoD checklist items must be done [8]). If validation passes, it sets `ready_for_review_at` timestamp and status.
    - `PUT /tasks/:id/review` – For a reviewer to mark as PUBLISHED or request changes. The request body or an endpoint parameter might indicate the action. For example, `PUT /tasks/:id/review?action=publish` vs `action=request_changes`. If publish: backend checks that current status is READY_FOR_REVIEW and the user is WM/Admin, then sets status to PUBLISHED (`published_at` timestamp). If changes requested: sets status to CHANGES_REQUESTED and perhaps records a reason/comment if provided. On changes requested, the `assigned_to` might remain the same editor and that editor will be notified. The Editor can then transition it back to IN_PROGRESS (or we could allow directly going from CHANGES_REQUESTED to IN_PROGRESS on the Editor's action). The loop is enforced by allowing that transition only if current status is CHANGES_REQUESTED and current user is the assigned editor.
    - `PUT /tasks/:id/qa_approve` – Mark as QA_APPROVED (likely by Admin or whomever does QA). And then `PUT /tasks/:id/done` – mark final done. We might combine these (maybe QA_APPROVED *is* the done state, but since the workflow lists both, we treat

QA_APPROVED as penultimate and DONE as final closure). Possibly marking QA_APPROVED could automatically set DONE if we consider them essentially the same step, but to match the requested workflow, we keep them distinct. The backend will ensure only an authorized role can do these (perhaps Admin or the Auditor if we give Auditor permission to mark QA after review). If the Auditor truly has no edit rights, then likely the QA approval would be done by the WM or Admin after a separate QA check, unless the Auditor communicates issues externally.

- `PATCH /tasks/:id` – Update fields of a task (e.g. Editor saving product info). Editors will use this to save title, description, etc., and to check/uncheck checklist items as they complete them. Each field update can be immediately saved or we can allow the Editor to work and then save all at once. The checklist items might also have their own endpoint, e.g. `PUT /tasks/:id/checklist/item_id` to mark complete. But updating them via the main task payload is simpler. We will also implement validation on these endpoints (e.g. ensuring price is a positive number, images meet criteria if possible to check in backend, etc.).

- `GET /todos` – List all to-dos (with summary of how many tasks each contains and their statuses). WM dashboard will use this to see active POs. Could include counts like X tasks done out of Y.

- `GET /dashboard` – We might create specialized endpoints to retrieve aggregated data for dashboards (e.g. for Admin: compute productivity metrics, counts of tasks in each state, maybe precompute the control chart data points and CFD series). These could also be computed client-side by pulling all tasks or history, but that might be heavy. Instead, we can leverage SQL to do aggregation:
  - E.g. a query for average lead time of tasks completed in the last 30 days, or count of tasks per editor. We could create a view or just have the API compute and return JSON for charts.
  - `GET /reports/cfd` – returns data series of dates vs counts per status for CFD chart.
  - `GET /reports/control-chart` – returns list of completed tasks with their cycle times and completion dates for plotting.
- `GET /audit-log` – For the Auditor, return recent TaskStatusHistory entries (join with user and task info) to display who changed what when.

The Express backend will use middleware for authentication and role-based authorization. Likely JWT-based auth or session cookies (Replit deployment can use simple cookie sessions if we prefer). For simplicity, we can have a login endpoint issuing a token that the React app stores. Each API call requires a valid token, and we decode it to get `user_id` and role. Authorization middleware then checks if the user's role is allowed for that endpoint or action. For example, an Editor trying to call the publish endpoint would get a 403 Forbidden.

We will use an ORM (such as **Sequelize** or **Prisma**) to interact with PostgreSQL, which will speed up development and ensure compatibility on Replit. The ORM models will mirror the tables described in the schema. Using an ORM also allows us to define the associations (one-to-many from ToDo to Tasks, etc.) and possibly use migrations to set up the schema on first run. If using Prisma, for instance, we can define the data model and have it migrate the PostgreSQL database accordingly.

Deployment on Replit: Replit can run a Node.js server and we can add PostgreSQL as part of the Replit environment (Replit offers a persistent filesystem where we might run a lightweight Postgres instance, or

connect to an external database service). We will bundle the application as a single Replit project that runs the Express server (listening on the default Replit port). The React app will be built and served from the Express `public` folder or similar. Environment-specific configurations (like database connection string, JWT secret) will be stored in Replit's secret configuration to avoid exposing them.

The Node/Express server will listen for requests from the React frontend (which, if on the same domain in production, avoids CORS issues). The architecture is **monolithic** for simplicity (front-end and back-end in one deployment, rather than microservices). This is suitable for a small team app and simplifies deployment to Replit.

Task Creation & Breakdown (To-Do Intake)

Feature 1: Creating To-Dos for Received Products. When new products arrive (e.g. a shipment from a vendor corresponding to a purchase order), the Warehouse Manager or Super Admin logs it in the system as a "To-Do". In the UI, this is a form where they enter: Vendor Name, Order/PO Number, Received Date, and any general notes. Submitting this form triggers `POST /todos` on the backend, creating a new ToDo record [9]. The system will respond by opening the detail view for that to-do, where the WM can specify the individual products included.

Feature 2: Automatic Breakdown into Product Tasks. Upon creating a to-do, the system can either: (a) prompt the WM to enter a list of SKUs or product names, or (b) if the WM knows the count, automatically generate that many empty tasks to be filled in. For example, if 5 new products came in, the WM could enter "5" and the system will create 5 Task entries under that to-do, each initially in NEW status and perhaps labeled "Product 1, Product 2, …" as placeholders. The WM can then edit each task to add specifics (or assign them to editors for them to fill out). Alternatively, the WM might add tasks one by one by clicking "Add Product" on the to-do page for each item.

Assigning Vendor/PO info: Each child task inherits some info from the to-do (like vendor and PO number), so editors know context. We avoid duplicate data by linking tasks to the to-do. The UI will likely show the vendor and PO on each task page (fetched via join or additional API call).

Initial TRIAGE: All newly created tasks start in **NEW**. The WM can open the to-do and see all its tasks listed as NEW. The WM might update some meta info or prioritize them. When ready, the WM (or Admin) moves tasks to **TRIAGE** – possibly this is as simple as "mark as triaged" action on the UI for each or in bulk. (Optionally, we could combine NEW and TRIAGE, but the workflow explicitly had both. We'll treat NEW as "not yet acknowledged", and TRIAGE as "acknowledged and to be assigned".)

Bulk Operations: If a to-do has many tasks, the system will support bulk assigning or bulk state changes. For instance, the WM could select all tasks in that to-do and assign them to an Editor in one action if desired. This saves time in large POs.

The creation and breakdown feature ensures that for every incoming batch of products, there is a corresponding set of tasks in the system, each representing a product to be created on Shopify. This lays the foundation for tracking each product through to completion.

Task Assignment & Self-Assignment

Feature 3: Editor Assignment of Tasks. Once tasks are triaged, the next step is to get them worked on by Editors. The system supports two modes of assignment: - **Manager Assignment:** The Warehouse Manager or Admin can assign tasks to a specific Editor. In the UI, the WM might have an "Assign" dropdown on each task (or select multiple tasks and assign to an Editor from a list of users). When a task is assigned via the `PUT /tasks/:id/assign` endpoint, the backend sets the task's `assigned_to` field and transitions its status from TRIAGE (or NEW) to **ASSIGNED**. The `assigned_at` timestamp is recorded. The assigned Editor will see this task appear in their "Assigned to Me" list. If using notifications, the system could also highlight to the Editor that a new task was assigned (e.g. via a toast message or simply it shows up on their dashboard).
- **Self-Assignment (Pull system):** Editors can also take initiative and pick up unassigned tasks. The Editors' dashboard will include an **Unassigned Tasks** or **Available Tasks** section (filtered to tasks in TRIAGE or NEW that are not yet assigned). An Editor can click "Claim" or "Start" on one of these, which triggers the same assign endpoint, assigning the task to that editor. This immediately moves the task to ASSIGNED (and possibly directly to IN_PROGRESS if they choose "Start Now"). We enforce that only tasks in the appropriate state can be self-assigned, and once taken, it won't appear for others. Self-assignment empowers editors to pick work when they are free, implementing a pull-based workflow common in Kanban.

The assignment mechanism will check for race conditions (if two editors try to claim the same task at once, only one assignment will succeed – the other will get an error and the UI will refresh that it's no longer available).

After assignment, the task's status is ASSIGNED. In this state, it is awaiting the Editor to actually begin work. The **time to assign** SLA clock stops at this point (we measure from creation to assignment for SLA) and the **time to start** clock begins.

Automatic Status Change on Start: We may choose to have the Editor explicitly click "Start Work" (transitioning from ASSIGNED to IN_PROGRESS), or we could assume that as soon as it's assigned to an Editor, they will start, and treat assignment as the start. However, having an explicit start allows measuring if there was a delay between assignment and actual work commencement. We will implement a "Start" button for editors on tasks that are ASSIGNED to them. This sets status to IN_PROGRESS and logs `started_at`. This two-step assignment/start process is useful if, for example, managers assign tasks in the morning but editors only begin in afternoon – we can then see that gap.

The UI will clearly indicate which tasks are assigned to whom and their status. The WM can see at a glance tasks that are unassigned (to make sure none fall through the cracks), and tasks that are assigned but not started (if an Editor is taking too long to begin, WM can intervene or reassign). We will allow reassigning tasks if needed: a WM/Admin can change the `assigned_to` on a task if it's in ASSIGNED or IN_PROGRESS (likely only if in ASSIGNED and not yet started, to avoid disruption). Reassigning will also be captured in history (with possibly a note in the audit log).

Permission Checks: Only WM or Admin can assign tasks to others. Editors can only assign (claim) tasks to themselves (not to other editors). This is enforced by backend: if an Editor attempts to assign a task to someone else, the API will reject it. The self-assignment endpoint will implicitly use the logged-in user's ID as assignee regardless of what is sent.

By supporting both push (manager assigns) and pull (editor claims) models, the system remains flexible for how the team prefers to operate. A small team might often use self-assignment (Editors picking what to do next from triaged tasks), but the WM can override or pre-assign if certain tasks are high priority or suited to specific editors.

Task Execution & In-Progress Work

Once an Editor has a task and is ready to execute, they will transition it to **IN_PROGRESS** (if not done automatically). At this point, the Editor is responsible for gathering all required information for the product and entering it into the system.

Product Information Entry: Each task has a detailed view where the Editor can input all Shopify-required fields and additional data. The product checklist will be prominently displayed here (likely as a list of sections or checkboxes). Key fields to cover (as per Shopify requirements and best practices) include:

- **Title:** The product name to display. *Checklist:* Title entered (cannot be blank) [9] .
- **Description:** A rich text or markdown field for the product description. *Checklist:* Description written with required details (cannot be blank) [9] .
- **Variants:** If the product has variants (size/color, etc.), the Editor will have a UI to add variants. Each variant may have fields like option values, SKU, price, barcode, inventory quantity. At minimum for a simple product, there is one default variant. *Checklist:* All variants and SKUs added as needed (and SKUs are unique).
- **Price:** The selling price (and optionally compare-at price). *Checklist:* Price set (non-zero) [9] .
- **Inventory Quantity:** Ensure an initial stock or inventory policy is set (this might be optional if not tracking inventory). *Checklist:* Inventory quantity entered (could be part of variant section) [9] .
- **Barcode:** If provided by vendor or required for scanning, ensure a barcode/ISBN/UPC is recorded for each variant [10] . *Checklist:* Barcode field filled (if applicable; can be optional but in our context they specifically mentioned it, so likely required if available).
- **Tags:** A list of tags for organization. The Editor should add relevant tags (categories, attributes) to help organize products on Shopify. *Checklist:* At least one tag added (and any specific tag conventions followed). Tags improve search and filtering on the store [9] .
- **Product Images:** The Editor will upload product images. We'll integrate an image upload component (since on Replit we may just store images on the file system or use a third-party like Cloudinary – but given no external integration specified, we might store base64 or file links in the DB). The system should enforce image quality requirements:
- **Resolution:** Images should meet a minimum resolution. (Shopify recommends 2048x2048 for product photos for zoom capability [11] , with minimum 800x800 for zoom to work [12] .) We'll set a rule like "each image must be at least 800x800; recommended 2048x2048". If an Editor uploads something smaller, the UI can warn them.
- **Aspect Ratio:** Ideally consistent (Shopify product images are often square). We'll encourage a standard aspect ratio (1:1) for a professional uniform look [13] [11] . The system could flag if one image is drastically different shape from others.
- **Watermarks:** The team policy is no watermarks on product images (to maintain a clean, professional appearance and avoid distracting overlays). If vendors provided images with watermarks or logos, the Editors should remove them or use alternate images. We won't automatically detect watermarks (that's complex), but it's a checklist item to verify "Images have no watermarks or unwanted text/logos".

- *Checklist:* All required images uploaded (e.g. at least one image per product) and they meet quality guidelines (resolution >= X, correct aspect ratio, no watermark). This might be a multi-part check where the app can automatically verify resolution and maybe aspect ratio, while the Editor manually confirms absence of watermarks.
- **SEO Fields:** Shopify allows editing the search engine listing preview (SEO title and meta description) and of course alt text on images:
- **SEO Title & Meta Description:** The Editor should customize the meta title and description for the product page to be SEO-friendly. By default, Shopify might use the product title and first part of description, but it's best to craft them. *Checklist:* SEO title set (usually can be the same as product title or a variant if needed for length), and meta description written (concise summary with keywords).
- **Alt Text for Images:** Each product image should have descriptive alt text for accessibility and SEO. *Checklist:* Alt text added for all images.

- **Structured Data/JSON-LD:** If the store uses structured data for products (e.g. via theme or app), ensure all required info for rich snippets is present. The Editor might not directly edit JSON-LD, but the checklist reminds them to include things like brand, SKU, price, etc., because the structured data (often auto-generated by theme) will pull from those fields. *Checklist:* All data needed for product schema is provided (price, availability, SKU, etc.). This overlaps with other fields but reinforces the importance of completeness.

- **Product Organization:** (Not explicitly listed, but typically important) – Vendor, Product Type, Collection assignment. We have vendor from the to-do, which we could automatically link as the product's Vendor in Shopify. Product Type could be filled if known (e.g. "Electronics > Phone"). Collections or categories might be assigned via tags or explicit collections. *Checklist:* Vendor and Product Type set if used; product added to relevant collections (this might be via tags or manual if needed).

As the Editor fills out these fields, the system can show real-time validation (e.g. cannot save if mandatory fields are empty). We'll use form validation in React plus server validation on save. The **Definition of Done checklist is integrated in the form** – each item can be a checkbox that either auto-checks when the corresponding field is filled or requires manual checking if it's qualitative. For instance, when the Editor types a description, we can automatically mark "Description provided" as done. For "Image quality", an Editor might check it after reviewing the images.

Saving Progress: Editors may not complete everything in one go, so they can save a draft while IN_PROGRESS. The task stays IN_PROGRESS until they explicitly mark it ready for review. We'll autosave or provide a Save button that calls `PATCH /tasks/:id` with partial updates (the Express app using ORM will update the fields and return the updated task).

All changes by the Editor are tracked. We could log content changes in an edit log if needed (not required, but at least the last updated timestamp on the task).

When the Editor believes the product is fully ready, they click **"Mark Ready for Review"**. This triggers the status transition to READY_FOR_REVIEW. The backend, as mentioned, will enforce that **all required fields are present and all mandatory checklist items are completed before allowing this transition** [8]. If something is missing, the API responds with a specific error (which the UI can display, e.g. "Cannot mark as Ready – Alt text is missing for one or more images" or "Barcode is missing" depending on the rules). This is

our **Definition of Done automation**: we only consider a task done (or ready to handoff) when it meets the agreed checklist criteria, ensuring consistent quality [8]. By embedding this in the workflow, we **prevent tasks from moving forward with incomplete work**, which maintains high quality standards. This is a best practice seen in Agile teams where a Definition of Done is a **measurable checklist** and transitions are blocked until it's fulfilled [8].

Once the task is in READY_FOR_REVIEW, the Editor can no longer edit the fields (the UI will lock them to prevent further changes while it's under review). If edits are needed, the task must go to CHANGES_REQUESTED and back to IN_PROGRESS.

To summarize this phase: Editors utilize a robust form and checklist interface to input product data, with validations and an enforced checklist to ensure completeness. The state machine and DoD checks guarantee that only quality-complete tasks move to the review stage, thereby automating quality control in the process.

Review & Publication Workflow

After an Editor marks a task as READY_FOR_REVIEW, it appears in the **Reviewer's queue**. The Warehouse Manager (or designated reviewer, could also be the Super Admin or another role in a larger team) will have a dashboard section showing all tasks in READY_FOR_REVIEW state, possibly sorted by how long they've been waiting (to honor FIFO or SLA for review turnaround).

Reviewing a Task: The reviewer opens the task's detail page in a read-only or review mode. They can see all the fields filled by the Editor and the checklist results (e.g. all checkmarks completed). They verify if the product data meets requirements and is accurate: - Check for typos or formatting issues in title/description. - Ensure images are good (no unexpected issues). - Verify pricing, variants, etc., align with any source info (perhaps the packing list). - Ensure SEO meta looks good (no truncation, relevant keywords). - Possibly use a preview mode (if the app could show a preview of how the listing might look on the storefront, though that might be out of scope; otherwise they might cross-check on a staging site or later on the live site).

If everything looks good, the reviewer proceeds to **publish** the product. Since our system isn't directly integrated with Shopify in this spec, "publishing" can be interpreted as the reviewer taking the data and creating the product in Shopify manually, then marking the task as PUBLISHED in our system. However, if we wanted to automate, we could integrate Shopify's API to actually create the product when this action is taken. The question did not explicitly request integration, so we will assume manual publish with a tracking state.

Mark as Published: The reviewer clicks "Publish" (or "Approve and Publish"), invoking the `PUT /tasks/:id/review?action=publish` endpoint. The backend will: - Check that current status is READY_FOR_REVIEW. - Possibly double-check that all checklist items are still complete (in case something was marked incomplete accidentally). - Set the status to **PUBLISHED**. - Record `published_at = now()` and `published_by = user_id` (we might add a field for who published in TaskStatusHistory anyway). - If we had Shopify API integration, here we'd push the product to Shopify and potentially store the Shopify product ID for reference, but in this standalone context we skip actual API calls.

After marking PUBLISHED, the task moves out of the review queue. The Editor could be notified that their product was approved (not mandatory, but maybe a small notification on their dashboard that product X was published successfully).

Changes Requested Loop: If the reviewer finds issues, they will click “Request Changes”. This might bring up a modal to enter comments on what needs fixing (or they could add a comment in the task). When confirmed, it calls the API to set status to **CHANGES_REQUESTED**. We log the time and who set it, and attach the comment if provided (in `TaskComments`). The task now appears back on the Editor’s radar, likely highlighted as needing rework. The Editor assigned will see it in their task list again (perhaps under a filter “Needs Revisions”). The Editor opens it, sees the reviewer’s feedback, and the fields are now unlocked for editing again (the system should allow editing in CHANGES_REQUESTED, which is effectively similar to IN_PROGRESS but flagged differently). The Editor makes the necessary corrections. Once done, they again click “Ready for Review”, and the cycle repeats. We keep track of how many times changes were requested (maybe count the loops, or simply it's in the history log each time).

This loop continues until the reviewer is satisfied. Each time, timestamps are updated. For SLA and metrics, if a task had changes requested, its **cycle time** extends (since we count from first start to done) and we could also measure “rework time”. But since it's uncommon to measure separately, we include it in overall cycle time.

QA Approval: Once a task is published, one more step is the **QA_APPROVED** state. This step is an additional safety net where someone (could be the Auditor or another senior person) verifies the live product on the website. This might include checking the live page for formatting, checking that it appears in the right collections, etc. In our roles, the Auditor is view-only, so perhaps the QA approval is done by the WM or Super Admin after the product is live. Alternatively, if the Auditor role were allowed to mark QA (we might extend their permission just for this action if needed).

The QA person views the product on the actual Shopify store and in the system marks the task as QA_APPROVED if everything looks correct. If something is wrong even after publish, they might move it back to CHANGES_REQUESTED (which would imply unpublishing or editing the live product – a bit complex, but ideally QA issues are minor and can be fixed on the fly). More likely, QA approval is just a quick check.

So the WM or Admin uses a `PUT /tasks/:id/qa_approve` to set status QA_APPROVED. Then possibly immediately calls `PUT /tasks/:id/done` to close it out as DONE. We can combine QA_APPROVED and DONE in one action or treat them as two sequential steps (maybe the moment you mark QA Approved, the system auto-moves to Done, since nothing else is needed – QA_APPROVED could be considered a substate that immediately transitions to DONE). But we will adhere to listed workflow and keep them separate states for clarity.

Done State: Marking a task DONE finalizes it. The backend sets `done_at` timestamp. At this point lead time is calculated (created_at to done_at). The task no longer appears in any active lists except in archive or reports. It’s effectively closed. If any further edits are needed after done, the process would have to reopen a new task or something – but normally done is done.

Notifications & Tracking: We will implement subtle notification cues in the UI for these transitions: - Editors see when tasks are returned for changes or when tasks are approved. - Reviewers/WM see when an

Editor marks something ready (perhaps a badge icon or an email if we extended to that, but UI alert should suffice). - Admin sees when tasks reach done or if any task is stuck in a loop too long.

Permissions Recap in Review Phase: Only WM/Admin can transition from READY_FOR_REVIEW to PUBLISHED or CHANGES_REQUESTED. Only WM/Admin (or assigned QA role) can transition PUBLISHED to QA_APPROVED and to DONE. Editors cannot directly alter states once in review except to work on changes if sent back. This ensures proper separation of duties for quality control.

By implementing a formal review and publish stage, the system ensures that no product goes live without oversight, and any corrections are tracked. The Definition of Done automation already prevented incomplete work; the review adds subjective quality control (catching errors or enforcing style guidelines). All of this is logged, contributing to accountability.

Dashboards and Role-Specific Views

Feature 7: Role-Based Dashboards – Each role gets a tailored dashboard highlighting the information and metrics most relevant to their responsibilities. We outline each:

Super Admin Dashboard

The Super Admin's dashboard provides an overview of the entire team's productivity and the health of the workflow. Key components:
- **Team Productivity Metrics:** Summary stats such as number of tasks completed this week/month, broken down by Editor. For example, a table or chart of tasks published per editor in the last 30 days, and their average cycle times. This helps identify high performers or who might be overloaded.
- **Throughput & Lead Time:** Display overall throughput (tasks completed per week) and average lead time. A chart could show tasks completed per week (throughput) and a rolling average lead time [14] [15]. This indicates if the team is improving or slowing down.
- **Bottleneck Identification:** A widget highlighting if any stage is becoming a bottleneck. For instance, "Tasks waiting for review: 5 (longest waiting: 3 days)" or "Tasks in progress with Editors: 10 (2 tasks older than 5 days)". The Super Admin can see if tasks pile up at a certain status, which might signal a resource issue. The cumulative flow diagram (discussed below) is a prime visualization for bottlenecks, so the Admin dashboard will include a **Cumulative Flow Diagram (CFD)**. The CFD will show at a glance if any category of work is growing (e.g. a widening band for "In Progress" could mean a bottleneck in completing tasks) [16] [17].
- **Aging Work in Progress:** A list or chart of the oldest active tasks (how long since creation). For example, "Oldest open tasks: Task #123 (10 days, In Progress by Editor A), Task #130 (8 days, Awaiting Review)". This draws attention to tasks that risk falling through cracks. A **Control Chart** can complement this by showing the distribution of cycle times for completed tasks and highlighting outliers [18] [19]. The Admin's dashboard could have a control chart scatterplot – each completed task as a point with completion date on X-axis and cycle time on Y-axis. Any point far above the average line indicates an unusually long task [19]. This helps the Admin investigate why those tasks took so long (perhaps indicating an obstacle or complexity). The control chart also shows process variability and stability [18], which the Admin monitors for process improvement.
- **Work In Progress (WIP) Limits / Counts:** If the team sets a soft WIP limit (e.g. each editor should not exceed 3 tasks in progress), the dashboard can show current WIP per person. E.g. "Editor A: 2 tasks in progress; Editor B: 4 tasks in progress (⚠ over WIP limit)". This helps in balancing workload.

- **Overall Counts:** A snapshot of how many tasks are in each state right now (like a mini pipeline summary: X New, Y In Progress, Z In Review, etc., and how many Done this week). This is essentially the current column counts on a Kanban board, also depicted by the rightmost slice of the CFD. It gives a sense of current WIP and backlog.
- **Aging WIP Chart:** Optionally, an aging WIP chart could be included which specifically focuses on tasks that are currently in progress and plots their age against some percentiles [20]. This chart would show each active task (unfinished) and how long it's been open compared to typical cycle time. It's a bit advanced, but it can quickly signal which active tasks are aging beyond normal. For example, tasks nearing the 85th percentile line for cycle time are at risk [21]. If an aging WIP chart is too much, a simpler "days in current status" column on task lists can achieve a similar purpose.

The Super Admin essentially sees everything and can drill down as needed (e.g. clicking a chart point could open that task's details). Their dashboard is about monitoring the **system performance** and ensuring nothing is stuck.

Warehouse Manager (WM) Dashboard

The WM's dashboard is tuned to operational oversight of ongoing work: - **Active To-Dos/Purchase Orders:** A section listing each open to-do (PO) with summary status. For example: "PO #123 (Vendor ABC, received 2025-09-10): 10 tasks, 6 Done, 2 In Progress, 2 Ready for Review; Oldest task age: 5 days". This allows the WM to track each batch of products and see which ones are lagging. If a PO is mostly done except one task, they can chase that specifically. Sorting by received date helps WM ensure older POs are completed before newer ones (to respect FIFO intake processing, if that's a policy).
- **Task Pipeline View:** The WM needs to see how tasks are flowing. A Kanban-style board could be presented (columns for each status with task cards) filtered to active tasks. The WM can use this interactive board to move tasks if needed (e.g. assign/unassign or expedite something). However, implementing a full drag-and-drop board is optional. At minimum, the WM can see counts and lists by status similar to Admin.
- **Review Queue:** A prominent section for tasks in READY_FOR_REVIEW (awaiting WM's review). This should list each task with how long it's been waiting (e.g. Task "Blue T-Shirt" – ready for review 2 hours ago). The WM should prioritize these to keep flow. Possibly color-code if something has been waiting beyond SLA (e.g. highlight after 24 hours waiting for review). The WM can click each to review and either publish or request changes.
- **SLA Countdown Timers:** The WM dashboard will include alerts for SLA breaches or near-breaches: - *Time to assign:* Perhaps a list of any tasks that have been NEW for more than X hours (say 24h). E.g. "3 tasks unassigned for >24h (PO #125)". This reminds WM to assign those.
- *Time to start:* Tasks assigned to editors but still not in progress after Y time. E.g. "Task #140 assigned to Editor B 2 days ago and not started (SLA 1 day exceeded) – consider follow-up."
- *Time to complete:* Tasks in progress for a long time. E.g. "Task 'Green Mug' in progress for 5 days (exceeds target of 3 days)."
- *Time to review:* Tasks ready for review for too long. E.g. "Task 'Red Dress' waiting for review for 2 days (SLA 1 day)."
These could be presented as a list of warnings or an "Alerts" panel. The UI can use icons (like a little clock or exclamation) next to tasks or in the list to denote breach. The SLA thresholds can be defined (perhaps 1-2 days for each stage in a small e-commerce context, or even hours if trying to be very fast). Since no external notification (email/SMS) is included, these UI alerts are critical to catch the WM's attention. We might use red text or badges for overdue tasks.
- **Bottleneck WIP:** The WM might also have a simplified CFD focusing on the last 1-2 weeks to see if any

stage is growing. But this might be too high-level for WM; they likely prefer concrete lists of tasks to act on.
- **Editor Workload:** The WM could have a quick view of each Editor's workload: e.g. Editor A – 3 tasks (1 in progress, 2 assigned not started); Editor B – 5 tasks (4 in progress, 1 in review changes). This helps WM balance assignment. If one editor has too much WIP and another is free, the WM might reassign or adjust.

In summary, the WM's dashboard is about **actionable lists** – what needs assignment, what needs review, where are we close to missing SLAs – and an overview of each incoming product batch.

Editor Dashboard

The Editors' dashboard focuses on their personal work and guidance on priorities: - **My Tasks List:** A list of tasks assigned to the logged-in Editor. This can be grouped by status: - *To Start (Assigned):* tasks they have but haven't started yet. Show since when assigned (e.g. "assigned 3 days ago" to prod a bit if it's aging). - *In Progress:* tasks currently being worked on. Possibly sort by deadlines or the order they were assigned. The Editor can click "Resume" to go into the edit interface for that task. - *In Review:* tasks that the Editor completed and are now in review (READ_ONLY for them). They can see which are waiting and for how long (though they can't do anything until feedback). It's useful for them to know, as they might nudge the WM if something is stuck or prepare for possible changes. - *Changes Requested:* tasks that came back for rework. These should be high priority for the Editor. The dashboard should flag them (maybe a separate section or an icon on them). The Editor clicks to see reviewer comments and fix immediately.
- **Checklist Completion Status:** For tasks in progress, the Editor dashboard could show a progress bar or count of checklist items completed vs total. This gives a quick sense of how far along each task is. For example, "Red Dress: 5/7 checklist items done". This can help the Editor manage their time and also pick which task to finish first (maybe the one that's 90% done).
- **Deadlines/SLA Warnings:** If the team sets expected turnaround times, the Editor view should inform them if any of their tasks are overdue or close to SLA breach. E.g. highlight a task in red if it's been in progress longer than the typical allowed time. This aligns the Editor with the SLAs so they can prioritize accordingly.
- **Self-Assign New Task:** If editors are expected to pull new work, the dashboard will have a section like "Available Tasks" showing unassigned triaged tasks. An Editor can pick one from here to add to their list. The UI should maybe display basic info: product name or placeholder, PO/vendor (to gauge complexity perhaps), and how long it's been waiting (so they might choose older ones first). They click "Claim" and it moves to My Tasks.
- **Guidelines & Shortcuts:** Optionally, the Editor dashboard can include quick links to reference material (like a quick checklist guide or image guidelines document) to remind them of quality standards. This isn't required, but a nice touch for usability.

Essentially, the Editor's dashboard is their personal Kanban of tasks and ensures they stay aware of priorities and what's next. It encourages self-management by showing them if something is aging or if they have capacity to take new tasks.

Auditor View

The Auditor has a special view primarily for **monitoring and auditing**: - **Audit Log:** A chronological feed of all significant actions in the system. Each entry could look like: "[2025-09-16 10:20] Editor Alice moved Task 104 (Blue Shirt) from IN_PROGRESS to READY_FOR_REVIEW" or "[2025-09-17 09:15] WM Bob assigned Task 110 (Green Shoes) to Editor Charlie". This log can be filterable by date range, by user, or by task. It provides

full traceability. The Auditor can use search (e.g. find all actions by a certain user, or all tasks that had Changes Requested multiple times). This helps investigate any anomalies or ensure the process is followed (for example, if an Editor consistently has multiple change requests, that might be noted).
- **Task Browser (Read-Only):** The Auditor can have a view similar to the Admin's overall task list or board, but read-only. They can filter tasks by status, assignee, etc., to inspect details. For instance, they may look at all tasks currently in PUBLISHED state to spot-check them, or tasks that took over X days to complete historically, etc.
- **Metrics:** While the Auditor is not managing workflow day-to-day, they might still be interested in some metrics for compliance or performance auditing. They could have access to the same charts as Admin (lead/cycle times, etc.) but it's not to intervene, just to report. Or they might just export data.
- **No Edit Controls:** All UI elements for changing state or editing tasks will be disabled for Auditors. If the Auditor notices an issue (like a task stuck or a quality problem), their role would be to raise it with the team externally rather than fix it themselves in the system.

The Auditor's role ensures transparency. Knowing that an Auditor can see every action might also encourage team members to follow the process closely (accountability). The audit log data can also be used in retrospectives or for continuous improvement.

SLA Monitoring & Alerting

Feature 8: SLA Monitoring (Service-Level Agreements) – The system will track four key elapsed times against target thresholds to ensure the workflow moves swiftly:

- **Time to Assign:** The time from when a task is created (NEW) to when it is assigned to an Editor (ASSIGNED). If this exceeds a certain threshold (e.g. 1 business day), the system should flag it. This measures how quickly new work is being taken up. A long assign time might indicate the WM is busy or there is confusion on who should take it. The system will highlight tasks that have been in NEW or TRIAGE too long without assignment. For example, an alert "Task #120 (received 3 days ago) has not been assigned yet – SLA 1 day breached." The WM dashboard will have this alert, and possibly the Admin too. This essentially monitors the **reaction time** to new work (which is often considered separate from cycle time [22]).

- **Time to Start Work:** The time from assignment to actual start (IN_PROGRESS). We measure this to ensure Editors begin work promptly after taking a task. If, say, an Editor is assigned a task and 2 days pass without starting, an alert will notify the Editor ("Task X was assigned to you 2 days ago and hasn't been started") and the WM (so they can possibly reassign or follow up). This could be part of "aging WIP" since an assigned-but-not-started task is essentially idle WIP. The threshold might be short (maybe same day or 0.5 days, depending on how quickly they're expected to begin).

- **Time to Complete (Cycle Time SLA):** The time from when an Editor starts work (IN_PROGRESS) to when the task is completed (DONE). There might be intermediate milestones, but essentially this covers from start through review and changes to done. If a task takes too long to complete, it might breach SLA. We might set a guideline like "all product uploads should be done within 7 days of receipt" or something. But since we break it down, maybe "once editing starts, task should be done within 2 days". This is tricky as changes requested can extend it. We can measure the overall lead time and have a threshold too. Possibly multiple SLA tiers:

- *Cycle Time SLA:* If the team expects that once active, a task is usually done in, say, 2 days, we flag those that exceed that.

- The control chart on Admin's side will show if many tasks exceed a certain range, but SLA alert is a more immediate thing in UI for an ongoing task. So for any in-progress task, we can calculate `now - started_at`; if > threshold, mark it red or show "Overdue". This draws attention to tasks dragging on. The WM's view of tasks in progress can use this to ask Editors if help is needed.

- **Time to Review:** The time from Ready for Review to Published (or to first review action). This monitors the reviewer's responsiveness. If an Editor finishes a task and it sits waiting for review for, say, more than X hours (maybe 24 hours max, ideally less), then alert the WM/Admin. This ensures that the reviewer doesn't bottleneck the system. The WM dashboard will definitely highlight tasks waiting for review > SLA. Perhaps if multiple people (like Admin vs WM) can review, the Admin seeing a delay might step in to review if WM is unavailable. It's about ensuring quick QA turnaround.

Implementing Alerts: Since we are not doing emails or external notifications initially, our approach is: - **Visual cues on Dashboards:** as described, color coding or alert lists for tasks violating or nearing SLAs. - **Badge counts:** For example, if there's an "Alerts" menu or icon, show a number of tasks overdue. However, likely just showing them in context (like a red "Late" label next to the task) is enough. - **Sorting/Filtering:** Perhaps automatically sort lists such that overdue items rise to the top. Or provide a filter "Show overdue only".

We will define the SLA durations as constants in the config (e.g. assign within 24h, start within 24h of assign, review within 24h of ready, complete within 3 days of start – these can be adjusted based on business needs). Because the team is small, these might be informally agreed. We might allow the Super Admin to configure these in the future via a settings UI that updates the SLA_Settings table, but initially hardcoded or environment config is fine.

During each page load or via periodic refresh (we could have the frontend poll every few minutes or the user refresh manually), the application will recalc times. It might be more efficient to have the backend compute an "SLA status" for each task on fetch. For example, when returning tasks, the API could include fields like `time_since_creation`, `time_since_assignment`, etc., and a boolean or status if it's overdue. However, these are easily computed on frontend too. Given the small scale, doing it on frontend is acceptable. But to maintain single source of truth and use DB server time, I lean to computing in backend: - A SQL query can compute age differences using `NOW()` on the server. For instance, selecting tasks in each state and comparing timestamps to threshold intervals. - Or the logic can be in Node: e.g. for each task returned in a list, attach an `alerts: { assign_overdue: true/false, review_overdue: ..., ... }`.

The UI will then interpret those flags to display alerts.

No matter the implementation, the outcome is that managers are promptly informed within the app of any SLA breaches so they can take corrective action (reassigning, expediting, etc.), thus maintaining a smooth flow.

Reporting and Visualization (Control Charts, CFD, Task Aging)

To ensure the team can analyze and continuously improve their process, the application will include rich visualization of workflow data:

Control Chart (Cycle Time Scatterplot): This chart plots each completed task's cycle time (y-axis) against the date it was completed (x-axis). Each point is one task. We will also overlay a rolling average or percentile lines if possible [23] [21]. The control chart helps spot trends and outliers in how long tasks take [18]. For example, if most points cluster around 2 days but a few are at 10 days, those outliers (circled or highlighted) indicate issues to investigate [19]. A stable process would show relatively tight clustering, whereas large variability means unpredictability [18]. This is valuable for the Super Admin and WM to identify what causes some tasks to lag (was it waiting on vendor info? was it a huge product with many variants?). It can also show improvement or degradation over time – if the trend of points is downward, cycle times are improving (faster completions) [24]. We may implement this using a chart library (e.g. Chart.js or Recharts) in the Admin dashboard. The data comes from TaskStatusHistory or directly from tasks (we can compute cycle time per task as `done_at - started_at`). We might precompute average and std deviation to draw control limits, or just visually eyeball it with the scatter. The control chart is mainly for internal process improvement discussions.

Cumulative Flow Diagram (CFD): The CFD is an area chart showing how many tasks were in each state over time [25] [26]. The X-axis is time (e.g. days or weeks), Y-axis is cumulative number of tasks. Each state (or group of states) is represented by a colored layer band. For our workflow, we might include states like: To Do (NEW+TRIAGE), In Progress (ASSIGNED+IN_PROGRESS possibly combined), In Review (READY_FOR_REVIEW+CHANGES_REQUESTED), Published (PUBLISHED+QA_APPROVED perhaps), and Done. Or we include each major state separately. Every day, the number of tasks in each state is calculated and plotted. Over time, as tasks move to Done, the Done band grows monotonically (the top of the Done band is the throughput line showing total completed) [27] [28]. The WIP is the area between Done and backlog. The CFD allows the team to see if work is flowing steadily or if WIP is building up in a particular stage: - Parallel bands (steady width) indicate a stable flow where new tasks = completed tasks in balance [28]. - If the band for "In Progress" or "Review" starts widening over time, that means tasks accumulate there (bottleneck forming) [29] [28]. For example, if the review band widens, it means tasks are getting stuck waiting for review. - A narrowing band could indicate that stage is being over-resourced or starved of work (less WIP there than usual) [28]. - The slope of the Done band (throughput line) shows how quickly tasks are being finished [28]. An upward slope that is steady is good; if it flattens, completions have stalled.

We will generate the CFD by taking snapshots of task counts. This can be done by querying the status history: for each day, count how many tasks had status X on that day. We might simplify by assuming tasks only move forward (which they mostly do except the loop for changes). Even with loop, the counts can be derived. A simpler approach is to sample daily at midnight: count tasks in each status (this requires a cron or we can generate it on-demand from history logs). On-demand approach: For each day in the range, determine which tasks were in each state. Since that's heavy, we might maintain a daily summary table. But given manageable data, we could approximate it.

The CFD will be displayed on the Admin (and maybe WM) dashboard. It could cover the last N weeks or be interactive with a date range. This is a powerful visual for the WM and Admin to discuss process changes. For example, if the WIP (in progress) band is constantly growing, they either need to throttle new tasks or add more editor capacity or find why tasks aren't closing.

Aging Work in Progress Chart: Mentioned earlier, this chart is typically a scatterplot (or bubble chart) showing current open tasks' age and maybe their status. X-axis might be days since creation or start, Y-axis some identifier or category. Often, percentile lines of past cycle time are drawn as a benchmark [21]. For instance, a line for 50th percentile cycle time, 70th, 85th – tasks above those lines are older than that proportion of past tasks. This helps answer "how does the age of current items compare to how long items typically take to finish?" [30]. If an active task is already older than 85% of past completed tasks' cycle times, it's a likely problem (flow debt) [31]. The Aging WIP chart gives a quick overview of all current WIP ages in one view, complementing the CFD and control chart which focus on completed items. We can provide a simplified version: e.g. a bar chart of each active task age vs average. However, since we're already highlighting aging in lists, a full chart may be optional. If time permits, a small chart with tasks as bars colored by status could be provided.

Reports for SLAs: We can include a small report perhaps monthly of how many tasks breached SLAs. But that might be overkill; the real-time alerting is more important.

Technology for Visualization: We will likely use a JavaScript charting library on the frontend. Options include Chart.js (simple and Replit-friendly), Recharts (if using React), or even Google Charts. We'll feed data via the API endpoints (/reports/...). The charts will be interactive (hover to see values, etc.).

Data accuracy: The TaskStatusHistory log ensures we have data for these charts. For example, cycle time for each task is simply (done_at - start_at) which we have. Lead time is (done_at - created_at). Throughput per week is count of tasks with done_at in that week. WIP on a given day is count of tasks not done by that day minus not started yet by that day, etc. These calculations will be tested to ensure charts are correct.

Control Chart Example: If on October 1, 10 tasks were completed, each with a certain cycle time in hours or days, each will appear as a dot above October 1 at its cycle time. We might draw a horizontal line for the average cycle time of, say, the last 30 tasks. If we see a wide spread, that signals variability to address [18]. If a bunch of points are far above the line (like clustering at high values), that signals a persistent bottleneck issue [19]. The team could then analyze what happened on those tasks.

CFD Example: Over September, the "In Progress" band might steadily increase from 5 to 15 tasks while "Done" band's growth slowed – this clearly shows a bottleneck where tasks enter faster than exit (perhaps due to a slow review process). The team can react by limiting how many tasks are being started or improving review capacity. CFDs help monitor **workflow stability** and ensure continuous flow [17].

All these visual tools support the **Kaizen** (continuous improvement) approach: identify where time is spent, where work piles up, and then tweak the process or resource allocation accordingly.

Definition of Done Automation & Quality Control

Ensuring each product task is truly "done" to a high standard before moving forward is vital. We have implemented this through a **Definition of Done (DoD) checklist** and automated validation:

- **Explicit Checklist:** As described, each task carries a checklist of required items (Shopify fields, image and SEO checks). This checklist is essentially the Definition of Done for adding a product. It is "short, measurable, and testable" – ideal characteristics of a DoD [32]. By making it explicit and visible on the

task, we communicate the expectations clearly to Editors and have a shared agreement of what "done" means.

- **Enforced via Workflow Validator:** Much like Jira's workflow validators that enforce checklists [8], our system prevents transitions at key points unless the checklist is satisfied. Specifically, the transition from IN_PROGRESS → READY_FOR_REVIEW will fail if any mandatory checklist item is not completed [8]. Similarly, we could also enforce that a task cannot be marked PUBLISHED (or DONE) unless the checklist is 100%. However, if it made it to ready for review, presumably it was 100% at that time. If somehow a reviewer adds a new requirement and wants to enforce before done, they could hold it in review until updated.
- **Automatic vs Manual Checks:** Some checklist items will be auto-checked by the system:
- Fields like title, description, price, etc., we can auto-mark as done once text is entered (and validate non-emptiness).
- Image resolution can be checked by the system reading the image metadata or HTML canvas to get dimensions – if below threshold, we do not mark that item complete (or flag it).
- Alt text: we can ensure every image uploaded has an alt text field filled – only then mark the "Alt text provided" item.
- These reduce manual effort and error. Items that are subjective (like "Description follows style guide" or "Images have no watermark") might need the Editor to tick a box confirming they've done it. We could require a reviewer to verify those as well.

- **Blocking Publish if Needed:** We could add another validator: when moving from READY_FOR_REVIEW to PUBLISHED, ensure nothing regressed. Ideally, the Editor cannot uncheck or remove something once in review. But if, say, a reviewer themselves edits something (if that was allowed), we'd want to re-check. Generally, if it passed the first gate, it should be fine. But double enforcement on publish is fine as a safety net.

- **Completion Automation:** When all items are checked, the system could automatically highlight that "All Done – ready to publish" on the Editor side. But we still require the human step to mark ready, giving them a chance to double-check everything. The automation is mainly about *prevention of mistakes*. This process ensures that when a task reaches a reviewer, it's already in a pretty good state (no missing data), allowing the reviewer to focus on content quality, not completeness.

- **Quality Metrics:** We might track how often tasks come back with Changes Requested as a measure of initial quality. High rework might indicate the checklist needs to be improved or editors need training on certain points. This is something the Auditor/Admin could watch (e.g. tasks with multiple review cycles).

- **Continuous Improvement of DoD:** The Super Admin can refine the checklist over time. For example, if a new requirement comes (like "must add a product to at least one manual collection"), they can add a checklist item and perhaps corresponding field. We should allow editing the checklist template (maybe not via UI initially, but at least via DB or config changes).

Using a Definition of Done checklist and automating its enforcement is a best practice to ensure quality and consistency [8] [33]. It "makes it easy for the team to know what's expected" and ensures the product manager's quality vision is delivered every time [33]. By embedding it directly into the workflow, we *bake quality into the process*, rather than relying on catching errors at the end. In agile terms, we are moving

quality checks as far upstream as possible (shift-left principle) – the Editor addresses them before requesting review.

State Machine Best Practices & Data Integrity

Throughout the design, we leverage state machine principles to manage the complex workflow: - We explicitly define all possible states and allowed transitions (as detailed in the Workflow section). By doing so, the application can enforce that tasks follow the intended path, preserving process integrity [1]. This avoids erroneous states (like skipping review). - **Transition Guards:** Each transition has guard conditions – for example, a transition may require certain checklist items complete, or a certain role to execute it [2]. We implement these guards in the backend logic for each state-change endpoint. If a condition fails, the transition is rejected with a clear message to the user. This prevents bad data (e.g. a task marked Done with missing info) and keeps the workflow consistent. - **Permissions tied to States:** We link roles to what they can do in each state [34]. E.g., Editors can edit tasks only when in ASSIGNED/IN_PROGRESS or CHANGES_REQUESTED (essentially when the task is "with them"), but not when it's under review or done. Reviewers (WM/Admin) can only push forward from review states but not edit content (unless they reassign or use admin override). This mapping of permissions to state ensures that, for instance, an Editor can't accidentally publish their own task without review. - **State Actions:** We can define actions that happen on entering or leaving states [35]. For example, on entering READY_FOR_REVIEW, automatically assign the task to the WM's review queue (maybe send a notification or just add to list). On leaving READY_FOR_REVIEW to CHANGES_REQUESTED, automatically assign it back to the original Editor and unlock the form. These can be coded in our transition handlers. It makes the user experience smoother (less manual steps to reassign, etc.). - **Data Consistency:** Using a state machine prevents contradictory data. For instance, if a task is marked DONE, our rules will ensure all necessary timestamps are filled, `assigned_to` is set, etc. If someone tried to mark DONE without a published_at or without an assignee, the state machine approach would have prevented reaching that state in the first place. This consistency is important for reliable reporting and avoiding edge cases.

- **Error Handling:** If somehow a transition fails (due to bug or manual DB tampering), the system should handle it gracefully. But since users won't have direct DB access, and we validate at API, it's safe.

In essence, the state machine approach provides a **structured and error-proof workflow engine** within our app. It reduces complexity by breaking the process into clear steps [36], and any attempt to do an invalid operation is caught early (with an appropriate message). This aligns with best practices for workflow design – making states and transitions first-class concepts leads to clarity and easier maintenance.

Conclusion

This technical plan described a full-stack solution for managing Shopify product upload workflows with fine-grained control and oversight. We combined proven concepts from agile workflow management (Kanban boards, lead/cycle time analysis, Definition of Done checklists, state machine enforcement) to create a robust system tailored to the team's needs. The architecture leverages Node.js/Express for a secure API backend, PostgreSQL for reliable data storage (fully normalized for consistency), and a React frontend for a rich interactive user experience. Each user role gets a customized interface to maximize their efficiency: Warehouse Manager and Admin to coordinate and monitor, Editors to execute with clarity on what's needed, and Auditors to verify and hold the team accountable.

By implementing strict workflow states and automating quality gates, the system ensures that products go live correctly and promptly. Visual dashboards and charts (Control Chart, CFD, aging WIP) provide insight into the process, helping identify bottlenecks and continuously improve throughput [17] [26]. SLA tracking within the app makes sure that team members are immediately aware of any delays relative to targets, so nothing slips through unnoticed.

In summary, this system will bring organization, transparency, and accountability to the product upload process. It will reduce errors (through checklist enforcement), reduce delays (through SLA alerts and clear responsibilities), and provide management with the data needed to optimize team performance. The design is scalable to more team members or increased product volume, and flexible enough to adapt (e.g. adjust checklists or add new states if the process evolves). With this in place on Replit, the small team can collaborate efficiently to keep the Shopify store up-to-date with new products in a controlled, measurable way.

Sources: The design principles and features were informed by industry best practices in agile workflow management and e-commerce content management, including the use of lead/cycle time metrics [4] [5], control charts for process stability [18], cumulative flow diagrams for WIP analysis [26], and Definition of Done checklists enforced in tooling [8]. These practices ensure a stable and predictable workflow with high quality outputs.

---

[1] [3] [35] [36] State machines | Model your business structure | commercetools Composable Commerce
https://docs.commercetools.com/learning-model-your-business-structure/state-machines/state-machines-page

[2] [34] Proper workflow: States, Transitions, etc - Ideas & Features - Fibery.io Community
https://community.fibery.io/t/proper-workflow-states-transitions-etc/1626

[4] [5] [14] [16] [18] [19] [26] 4 Kanban Metrics You Should Be Using in 2024 | Atlassian
https://www.atlassian.com/agile/project-management/kanban-metrics

[6] [7] [22] [23] [24] Timing metrics explained: Lead time vs Cycle time - Screenful Guide
https://screenful.com/guide/timings-metrics-explained-lead-time-vs-cycle-time

[8] [33] Definition of Done in Jira : How to create a DoD Checklist
https://www.herocoders.com/blog/definition-of-done

[9] Shopify Product Data Entry | FireBear
https://firebearstudio.com/blog/shopify-product-data-entry.html

[10] Adding and updating products - Shopify Help Center
https://help.shopify.com/en/manual/products/add-update-products

[11] [12] [13] Website Image Size Guidelines for 2025 - Shopify
https://www.shopify.com/blog/image-sizes

[15] [17] [25] [27] [28] [29] Cumulative flow diagrams — how to create and read them
https://business.adobe.com/blog/basics/cumulative-flow

[20] [21] Kanban Charts & How to Track Progress in Kanban
https://www.knowledgehut.com/blog/agile/kanban-charts

30 Kanban Metrics Cheat Sheet - Project Manager Template
https://www.projectmanagertemplate.com/ru/cheat-crib-sheet/kanban-metrics-cheat-sheet

31 Want to see how long your tasks are actually taking? The Aging WIP ...
https://www.facebook.com/NaveHQ/posts/want-to-see-how-long-your-tasks-are-actually-taking-the-aging-wip-chart-is-your-/975315707929303/

32 Getting started with a Definition of Done (DoD) - Scrum.org
https://www.scrum.org/resources/blog/getting-started-definition-done-dod