ChatGPT

Creative Operations System Technical Blueprint

Development Context and Assumptions

Team & Build Approach: We assume a small internal NOCO development team (1–2 engineers) will build this system. This means favoring simplicity and direct implementation over complex, heavy frameworks. The initial architecture can be a monolithic web application (to reduce overhead) with clear modular separation for rules and UI.

Initial Deployment Scope: The system will first be used internally by NOCO (single-agency single-tenant). We will design core logic for one organization's needs, avoiding multi-tenant complexity at MVP stage. However, the codebase will be structured to add multi-agency support later (e.g. by scoping data with an Agency/Client ID and externalizing certain configurations). In other words, **build for one, plan for many** – do not prematurely add tenancy overhead, but keep the door open for scaling up to a SaaS offering in future phases.

Platform Choices: There are no hard constraints given, so we choose proven, developer-friendly tech: - **Backend:** A TypeScript/Node.js server (or Python/Django – any stack with strong support for state machines and business logic). Node/TS is suitable for a small team and allows using libraries for workflows. A relational database (PostgreSQL) will store structured data (clients, projects, payments, etc.). The backend will embed the rule engine and state management layer. - **Frontend:** A modern reactive web UI (e.g. React with TypeScript) to implement the dynamic interface that reflects state and permissions in real-time. React's ecosystem can handle complex stateful UI, and libraries like XState (state machines) can manage client-side workflow visualization. - **Hosting:** Initially, a cloud deployment (e.g. a small VM or container on AWS/Azure or a platform like Vercel for the frontend) will suffice for NOCO's use. We'll favor a containerized deployment for portability. No specialized infrastructure is required at MVP; we avoid over-engineering and keep hosting simple.

Rationale: An internal small team with Node/React can move fast and iterate. Keeping the first version single-tenant and monolithic avoids premature scaling issues. We **avoid overengineering** — focusing on core functionality that's immediately valuable [1] . This context influences our design: rules may be initially hard-coded (faster to implement for one client) and later made configurable for multi-tenant SaaS, and the UI will be tailored to one organization's workflows at first.

Phase 1: System Core & Technical Paradigm (State-Machine Driven)

Why State Machines: The system must *actively enforce* business rules and sequential workflows, so a state-machine driven architecture is essential. In a creative operations context, many entities (projects, deliverables, invoices, etc.) go through defined states (e.g. *Draft → In Review → Completed* for a deliverable) and certain actions are only allowed in certain states. A finite state machine paradigm guarantees that at any time, an entity is in one well-defined state, and transitions to other states occur only by explicit allowed events. This prevents ambiguous statuses or out-of-order steps. By using state machines, *we eliminate undefined or contradictory states* – the system is always in a valid status and you

cannot have conflicting conditions simultaneously [2]. Importantly, *undefined transitions are outright prohibited*; if a transition from state A to state B isn't defined in the workflow, the system will not allow it [3]. In short, the system doesn't rely on user convention or after-the-fact warnings – it **prevents invalid actions at the source**.

Entities with State: Key domain entities will have enforced state machines. For example: - A **Project** might be in states like *Pending Scoping*, *Active*, *On Hold*, *Completed*. - A **Deliverable** (piece of work to deliver) might be *In Progress*, *In Review (with client)*, *Approved*, *Delivered*. - A **Revision Cycle** might be *Open*, *Pending Client Feedback*, *Closed*. - **Payments/Invoices** could be *Unpaid*, *Paid*, *Past Due*. Each such entity's state machine encodes allowed moves (transitions) and blocks moves that violate rules (e.g. you cannot move a deliverable to "Delivered" if its invoice is unpaid). The state acts as a gatekeeper.

State Transitions & Blocking: Transitions are triggered by **events** in the system (usually user actions or system timers). An event attempts to take an entity from one state to another – the state machine logic will check if this event is permitted given the current state and defined transitions. If yes, the transition happens and may fire associated actions; if not, the system *blocks it outright*. For instance, an event "Deliver Files" on a project will only succeed if the project's *Payment Status* state is "Paid" (no payment, no delivery rule). If the payment state is "Unpaid", the system will refuse to execute the "Deliver" event – *the user cannot bypass this*. This is fundamentally different from a typical app that might just show a warning; here the forbidden action is non-selectable or results in an immediate error, with no workaround. By encoding business rules as state transition **constraints**, we ensure the application enforces policy by design, not by after-the-fact checks.

Preventing Actions vs Warning Users: In this system, rules are preventative. Instead of popping up warnings ("Are you sure?" or "This might violate policy"), the system simply will not present or process forbidden actions. For example, if a deliverable has reached its maximum allowed revision rounds, the "Request Revision" button is disabled or removed – the user is not even given the chance to exceed the limit. This design follows the philosophy that the best way to prevent rule violations is to make them impossible to do, rather than relying on user judgment with warnings. (For comparison, YouTrack's workflow rules work similarly: any undefined or guard-failed transition is ignored, and you must meet the guard condition to proceed [4].) The UI will reflect this by greying out or hiding controls that the current state disallows, reinforcing that "no means no."

Core Primitives Definitions: To clarify the logic building blocks, we define: - **State:** A discrete status of an entity representing a stage in its lifecycle. States are usually named (e.g. "In Progress", "Awaiting Payment"). An entity can only be in one state at a time (per state machine). States often carry implicit permissions and rules (e.g. "Approved" deliverable state means client can download files). - **Transition:** A permitted change from one state to another, triggered by an event. Transitions have a *source state*, a *target state*, and are labeled by the event or action name (e.g. *Submit for Approval*, *Complete Project*). Transitions can have conditions (guards) that must be true (like "invoice paid") and can execute side-effects (like sending notifications) when they occur. Any transition not explicitly defined is disallowed [5] – this ensures the workflow can't skip steps or go backwards unless rules allow it. - **Constraint (Guard):** A boolean rule attached to a transition that must evaluate true for the transition to be allowed. Constraints enforce business rules at the moment of transition. For example, *guard:* `if (invoice.status == Paid)` on the *Deliver* transition; if the invoice isn't paid, the guard fails and the transition is blocked. Guards centralize rule checks so that forbidden moves simply won't execute. They also provide a single point to attach messages/logging explaining why an action is blocked (e.g. "Cannot deliver: payment is outstanding"). - **Event:** An occurrence that triggers state evaluation. Events can be user-initiated (e.g. a user clicks "Approve Design") or system-generated (e.g. a monthly cron event triggers "Expire Retainer Hours"). Each event is handled by the state machine, which looks up the current state and checks if that event has a valid transition out of that state. If so, it moves to the new

state and fires any associated actions. Events are the only way to change state – which makes all state changes explicit and traceable (no silent or accidental state changes). This explicit event-driven transition is crucial for predictability and debugging [2]. - **Audit Log:** Every significant event and state transition is recorded in an **audit log** for accountability. An audit log is essentially *a chronological record of all actions and events in the system, documenting who did what and when* [6]. In our system, every time a state transition occurs (or is attempted and blocked), we log an entry: user X attempted event Y on entity Z at time T, and whether it succeeded or was rejected (with reason). The audit log ensures **operational safety** – we can always trace back to see if a deliverable was sent without payment (should never happen) or who overrode a constraint if that capability exists. This provides transparency and is important if disputes arise ("why wasn't I given the files?" – check the log: it shows payment was late, so system auto-blocked it). Audit logs also cover configuration changes and rule overrides. They are critical for compliance and trust, giving a clear history of state changes and rule enforcement in case we need to prove the process was followed.

In summary, a state-machine driven core guarantees **clarity and authority in enforcement**. Each entity's allowed behavior is formalized in code, leaving no room for improvisation. The system acts as a gatekeeper at the data layer: if a rule doesn't allow an action, the system won't perform it – period. This lays the foundation for the rule-enforced approach of the entire SaaS.

Phase 2: Domain Model & Data Structure (Entities and Relationships)

We adopt a domain-driven design, identifying core **entities** that mirror the real-world objects in a creative agency's operation. Below are the essential entities, their relationships, and where the business rules attach:

- **Client:** Represents an agency's client (the brand or company receiving creative services). Clients have contracts and projects. *Relationship:* A Client can have multiple Contracts/Projects. *Importance:* Clients may have specific settings (e.g. their payment terms, or whether they have rights to RAW assets). Generally, **rules are enforced *on behalf of* clients** – e.g. the system protects clients from receiving work without paying (and protects the agency from doing unpaid work). The Client entity itself might not carry many rules, but is linked to many rule-bearing entities (contract terms, payment status, etc.).

- **Contract/Scope:** Formal agreement with a Client defining the terms: project scope, payment schedule, number of revision rounds included, retainer hours, asset rights, etc. A Contract can span multiple Projects or deliverables, or there might be one contract per project. *Relationship:* Contract belongs to a Client. *Rules Ownership:* The **Contract** is a primary container of business rules – it encodes the policies that the system must enforce. For example, the contract specifies **payment terms** (e.g. 50% upfront, or Net 30 on invoice), **revision limits** (e.g. "max 3 revision rounds included" [7]), **retainer terms** (e.g. hours per month, expiry), and **asset rights** (e.g. "RAW files not provided unless separately purchased"). These rule values are stored in the contract and the rule engine references them. **State Triggers:** Signing a contract might transition a Project from Pending to Active. The contract's existence with certain terms triggers enforcement (e.g. if revision count in a deliverable hits the contract's limit, state changes to "Out of Scope").

- **Project:** A container for a set of creative deliverables or a campaign. Projects go through states (perhaps *Planned* → *Active* → *In Review* → *Completed* → *Archived*). *Relationship:* A Project is under a Contract/Client. A Project has many Deliverables and may have associated Retainer Allocations

and Invoices. *Rules:* A Project might enforce sequential phase rules – e.g., cannot start (Active) until contract is signed and initial payment received; cannot be marked Completed until all deliverables are delivered and final invoice paid. Projects also might aggregate status (if any deliverable is blocked by a rule, the project as a whole might show a blocked status). *Triggers:* Project state changes often coordinate sub-entities: e.g., moving a project to "Completed" might auto-complete all open tasks or require all Deliverables to be Approved first (enforce sequential handoff).

- **Deliverable:** A specific output (design, video, etc.) to be delivered to the client. Deliverables are central to daily operations and have the strictest enforcement:

- **States:** e.g. *In Progress (internal work) → In Review (sent to client for feedback) → Approved (client approved) → Delivered (final files delivered)*.
- *Relationship:* Deliverables belong to a Project (and thus to a Client via project). They have associated Revision Cycles and Assets.

- *Rules:* **No Payment → No Delivery** is primarily checked at the deliverable level. Before moving to "Delivered" state (or before releasing final files), the system checks the related Invoice/Payment. If payment is incomplete, the **Deliverable cannot transition to Delivered**. This means the client cannot download final high-res files until payment is received (the contract likely states the agency owns the work until paid [8]). The system enforces that by not providing the download link or by locking the file.

    - **Limited Revision Rounds:** Each Deliverable is usually allowed a limited number of revision cycles (as per contract). The Deliverable or its Revision Cycle entity will count revisions. Once the limit (say 2 rounds) is reached, no further "Request Revision" action is allowed without an approved scope change. The deliverable could enter a state like "Revision Limit Met" which requires either client approval of extra charges or moves to finalization. This implements the contract clause *"max two rounds of revisions then additional fees apply"* [7] in the system.
    - **Approval gating:** A deliverable might require client approval (sign-off) before it can be delivered. The system will ensure the deliverable goes through "Client Approved" state explicitly (with a client-side approval action) before "Delivered" can happen.
    - **Asset protection:** By default, deliverables only include final outputs, not source files. If a client tries to access *RAW assets* for a deliverable (e.g. Photoshop source files), the system checks if the contract permits it. Usually, **RAW assets are protected** – not provided unless a separate agreement or fee is in place. Many agencies keep source files and only deliver PDFs/images; source files are extra [9]. Our Asset entity (below) will encode this, but Deliverable state might reflect if raw assets are "unlocked" or not.
    - **Triggers:** When a Deliverable is marked Delivered, it could trigger an automated email to client with a download link *only if* payment is completed, etc. If a client attempts to download from the portal, the system checks deliverable state and payment status to either allow or block.

- **Revision Cycle:** Represents one round of revisions on a Deliverable. Each cycle may go through states: *Open (awaiting client feedback) → In Progress (designers revising) → Closed*.

- *Relationship:* Revision Cycles belong to a Deliverable. Possibly one active revision at a time.
- *Rules:* The number of Revision Cycles per deliverable is capped. When a deliverable is first delivered for review, a Revision Cycle opens. If the client requests changes, that cycle closes and a new one may open (count increment). The **contract's revision limit** is checked each time a

new cycle would open. If the limit is exceeded, the system will block opening another cycle. The deliverable might then require a Change Order (new Contract or contract amendment) to proceed.

- *Triggers:* On closing a revision cycle, the deliverable might move back to "In Progress" state for the team. On exhausting revisions, the deliverable could auto-transition to a "Final Approval Required" state where either the client accepts as-is or negotiates a new contract.

- **Retainer Allocation:** If the client is on a monthly retainer, this entity tracks the hours or credits allocated and used in a given period (month).

- *Relationship:* A Retainer Allocation is linked to a Contract (the retainer agreement) and perhaps to specific Projects or tasks consuming it. Usually one per month per contract.
- *Rules:* **Hours expire monthly** – if the client doesn't use all hours in a month, they expire (use-it-or-lose-it) and do not roll over [10]. The system enforces this by resetting available hours each period and not carrying over the balance. When a new month begins, the previous allocation's remaining hours become 0 (or archived). This prevents clients from accumulating unused hours beyond what the contract allows. The Retainer Allocation entity likely has a validity date range for each chunk of hours. Once past the end date, any remaining hours can't be applied to new tasks.
    - Also, tasks logged against retainer hours must decrement from the current month's allocation. If hours run out, the system might block logging more time or flag that additional work will be billed separately.

- *Triggers:* End-of-month cron jobs trigger an event to mark retainer hours expired and perhaps generate a report of usage. The state of a retainer allocation might be *Active*, *Depleted*, *Expired*. If a user tries to book work under an expired retainer, the system will stop them unless a new allocation is in place.

- **Payment / Invoice:** Represents an invoice billed to the client, often tied to a project or time period. It has amounts, due date, status, etc.

- *Relationship:* Invoices link to Contract/Project (and thus Client). A project could have multiple invoices (e.g. schedule or milestones).
- *Rules:* **No Payment, No Delivery** is enforced here. An Invoice has a **status** (e.g. *Pending Payment*, *Paid*, *Overdue*). The critical rule: deliverables under that invoice cannot reach final delivery while the invoice is Pending/Overdue. The system might lock the Deliverable state at "Approved/ Pending Delivery" and not allow transition to Delivered until invoice.status == Paid. As soon as the invoice is marked Paid (which could be through an integrated payment gateway or manual admin update), an event triggers allowing those deliverables to move to Delivered. This aligns with standard practice: *agency retains ownership until full payment* [8].
    - If a payment is overdue, the system could also restrict other actions (e.g. halt new project kick-offs for that client or prevent booking studio time).
    - Payments can also trigger state transitions: e.g. receiving an upfront payment might transition a Project from "Pending" to "Active" automatically.

- *Triggers:* Payment status updates may come from external systems (accounting software or payment processor webhook). Those will fire events in our system to update invoice state and then cascade to unlock or lock certain things (e.g., if an invoice becomes overdue, perhaps deliverables revert to a "On Hold" state).

- **Asset:** Any file or creative asset produced. We distinguish **Final Assets** vs **RAW Assets**:

- *Final Asset:* The deliverable output meant for client use (e.g. exported PNG, final video). These are delivered when a deliverable is marked Delivered. Final assets are usually accessible to the client once delivered (given payment is done).
- *RAW Asset:* The source files (Adobe project files, raw footage, design source) that are typically *not* delivered unless specified. The system should mark these as **protected by default**. Each asset might have a flag for "Protected" versus "Shareable". RAW assets default to Protected, meaning only internal team (or specific roles) can download them. If a client attempts to request RAW files, a rule checks if they have entitlement. Often, contracts say raw files require an extra fee or are not provided; our system enforces that by either not listing raw files on the client side, or by requiring a special approval workflow (maybe an event "Client Requested RAW" triggers an ops review). Most commonly, if no explicit entitlement, the answer is simply "No, you cannot have it". As an example from industry: many do *not hand over any files other than the finished work* by default [9].
- *Relationship:* Assets belong to a Deliverable (or a Project). They have metadata like file type, upload date, etc.
- *Rules:* Download permissions for assets are state and role controlled (see Permissions section). Final assets are accessible to the Client only when deliverable is in Delivered state *and* requisite payments are complete. RAW assets are never accessible to Client unless an override state is set (perhaps a Deliverable state "RAW Access Granted" if they paid for it). Internally, assets might also be staged: e.g. intermediate files only visible to the internal team until final.

- *Triggers:* When a deliverable moves to Delivered, a process might package final assets for client download (e.g. generate a ZIP). If a deliverable is reverted (say payment bounced and we mark invoice unpaid again), the system could disable the download link (perhaps moving deliverable back to an earlier state and marking assets as temporarily inaccessible).

- **Studio Booking:** Represents scheduling the internal studio/resources for things like photoshoots or recordings.

- *Relationship:* A Booking is associated with either a Project or a specific date/time and resource (like reserving the photography studio for Client X on date Y).
- *Rules:* Because using the studio involves risk and cost, **booking requires a signed waiver and acceptance of liability** from the client or involved parties. The system will enforce that for any studio booking, a digital waiver must be signed (perhaps the Client user has to check a box or upload a signed form) before the booking state can be confirmed. So the Booking entity might have states: *Tentative* → *Confirmed* → *Completed*. The transition Tentative→Confirmed is only allowed if the waiver flag is true (i.e., they signed). If not, the booking stays Tentative and the date/time cannot be fully reserved. This ensures legal protection is in place.
    - Additionally, the system could require that the client has paid any deposit or fee related to the booking upfront (again no free booking without payment).
    - If a booking involves external contractors or equipment, their agreements might also be attached and required.

- *Triggers:* When a user tries to confirm a booking, the system checks waiver received (maybe an uploaded PDF or a digital signature recorded). If missing, it blocks confirmation and prompts them that a waiver is required. Once confirmed, the system might send a confirmation email including the liability terms the client agreed to.

- **User Role:** We have various user roles in the system: **Owner** (agency owner/executive), **Ops** (operations/project manager), **Studio** (photographers, producers in studio), **Digital** (designers, post-production team), **Client** (the client-side user).

- Roles define baseline permissions (what sections of the app they can see, which actions they can initiate) but **role alone is not enough to grant an action** – the state of the specific object also matters (state-based permissions will be detailed in Phase 4).
- *Relationship:* Users can belong to one or more roles, but in our context roles are mostly distinct groups. A user is tied to an Agency in a multi-tenant future (for now, one agency's users).
- *Rules & Triggers:* Role defines who can trigger certain state transitions. For example, only a **Client** user can move a deliverable from "In Review" to "Approved" (sign-off), whereas only an **Ops** or **Owner** can move a project to "Active" or adjust a contract. **Studio** role users might only interact with the Studio Booking module (e.g. mark a booking as Completed, but cannot confirm it without Ops approval perhaps). The **Digital** (internal creative team) might mark deliverables as ready for client review, but cannot directly deliver to client – Ops must verify payment first. These distinctions are configured by mapping transitions to roles (e.g. transition "Submit to Client for Review" – allowed to Digital role when deliverable state = In Progress; transition "Approve Design" – allowed to Client role when deliverable state = In Review; transition "Deliver Final Files" – allowed to Ops/Owner when state = Approved *and* payment done).

Entity Rule Ownership vs Consumption: In this model, **Contracts** are the primary owners of configuration rules (limits, terms). **Projects/Deliverables** consume those rules – they implement the logic to enforce them. For instance, Contract says "2 revisions max" and the Deliverable's Revision Cycle logic uses that value to allow or block new cycles. Similarly, Contract might say "Retainer hours expire monthly, no rollover" and the Retainer Allocation entity enforces that by resetting hours. **Payments/ Invoices** trigger state changes (they don't own rules themselves, but their status is consumed by rules on Deliverables and Projects). **User Roles** own some rules about who can do what, but **state machines** further constrain those actions based on object state (we will combine these in the permissions model).

Triggers of State Transitions: - **Users (human actions)**: e.g. an Ops user clicking "Mark Project Complete" triggers a transition on Project; a Client clicking "Request Revision" triggers a transition on Revision Cycle; a Designer clicking "Upload Final Files" triggers deliverable -> In Review state. - **System Events:** e.g. a scheduled job on the last day of the month triggers a "ExpireRetainer" event on Retainer Allocation; a payment webhook triggers an "InvoicePaid" event in the system which transitions Invoice state and then possibly Project/Deliverables. - **Chained Transitions:** sometimes one entity's state change triggers another's. For example, if a Deliverable transitions to Delivered, the Project might check if all deliverables are delivered and then auto-transition to Completed. These are implemented either as listeners in the domain model or within the state machine definitions (entry/exit actions on states).

All these ensure the domain model isn't just passive data – it actively participates in enforcing the defined business rules and workflow sequence. The relationships help propagate rules: e.g., a late payment at invoice level bubbles up to block deliverable and project completion; an expired retainer on contract level prevents new tasks from being created under projects for that client, etc. By modeling real-world rules directly in the relationships and states, we create a system where **the design reflects operational reality, not just static info**.

(No SQL schema provided here, but each entity would correspond to a table or collection. For example, Deliverable table with status field and foreign keys to Project and Invoice, etc. The emphasis is on conceptual relations and rules.)

Phase 3: Rule Engine Design

The **Rule Engine** is the heart of how policies are evaluated and enforced in the system. It determines *at runtime* whether an attempted action is allowed given the current data (states, values) and predefined rules. Key design decisions for our rule engine:

- **Location of Rules:** Initially, many rules will be **hard-coded in the application logic** for simplicity (MVP stage). For example, the check "if invoice not paid, block delivery" might be an `if` statement or a state machine guard in code. This is acceptable for one internal deployment where rules are well-known and stable. As we evolve to a SaaS for multiple agencies, we plan to make many of these rules **configurable** per tenant (via an admin UI or config files). That means eventually moving rules out of code and into a rules repository or database where they can differ per agency or be updated without code changes. A likely approach is to introduce a rules configuration table (or JSON policy files) storing things like max revisions, payment required flags, etc., associated with a contract or agency profile. Initially though, to get up and running, we **bake in the core universal rules** since these are not optional (e.g. the system will *always* enforce no delivery without payment, we don't need that to be toggled at first).

- **Evaluation Mechanism:** The engine will evaluate rules primarily during state transition attempts. This can be implemented as:

- Guard clauses on state machine transitions (if using a library like XState or a custom state pattern). For example, a transition "deliver()" has `guard: invoice.isPaid == true`. If false, the transition is not taken.
- Alternatively, a central **rules service** that, given an action and context, returns allowed or not. For instance, before a controller method executes an action, it calls `Rules.canDeliver(projectId)` which checks the necessary conditions.
- We prefer embedding the rules in the state machine definitions, as it keeps logic close to workflow. But we'll structure the code such that it's easy to refactor. Possibly using a simple rule evaluation library or even a syntax like JSON-based rules (not fully needed at start).

- **Example:** Pseudocode for deliverable delivery rule:

```
stateMachine.deliverable.transition('Deliver', from='Approved',
to='Delivered', guard=(ctx) => {
   if (!ctx.deliverable.project.invoice.isPaid) {
       return false;
   }
   return true;
});
```

If returns false, the engine prevents transition and could raise an error event.

- **Rule Priorities & Conflict Resolution:** In a strict system like this, most rules are straightforward (do X only if Y). But conflicts could arise if multiple rules overlap. For example, consider a scenario: *Revision limit reached* vs *Client has not approved deliverable*. The system might encounter a situation where two constraints both apply. We need a strategy:

- Generally, **most restrictive rule wins**. If any active rule says “no”, the action is a no-go. The engine will evaluate all relevant conditions, and if one fails, it blocks the action. We’re not negotiating between rules – the design is to **err on the side of blocking** unless explicitly allowed. So it’s a logical AND of all required conditions for an action.
- If an action has multiple guard conditions (invoice paid AND within revision limit AND user has correct role), all must be true to proceed. This approach inherently handles conflicts by requiring all business conditions be satisfied.

- There may be cases where rules could conflict in outcomes. For instance, a client might have a special override on revision count (maybe they purchased an extra round) – this is handled as a rule override rather than a conflict (discussed below). Another example: If a deliverable’s contract says “No RAW files” but an Ops user wants to manually send one, the system should ideally force an override action (so someone with authority explicitly breaks the rule, which is logged). The engine won’t “auto-resolve” it; it will just see a violation unless an override flag is present.

- **Rule Overrides:** Despite our goal of strict enforcement, real operations sometimes require exceptions. The system allows overrides but in a controlled manner:

- Only specific roles (likely **Owner** or authorized Ops) can perform an override. An override is an action that forces something to happen despite a rule. For example, an Owner might override the payment rule to deliver files to a long-time client who is trusted to pay later.
- Implementing this: an override might be a special admin-only event, such as `ForceDeliver`. This transition would bypass the usual guard but still record that an override was used.
- **Audit Logging of Overrides:** Every override must be clearly logged (who did it, when, and which rule was bypassed). The audit entry would note something like “Override: Delivered deliverable D123 to Client ACME despite Unpaid invoice, by User X (Owner) on [date]”. This ensures accountability – overrides are possible but never silent. They’re effectively “break glass in case of emergency” operations.
- The engine design can incorporate overrides by layering rules: e.g., normally `canDeliver = invoicePaid`, but if user.role == Owner and they provide an override confirmation (maybe a UI prompt “Override rules?”), we let it go through and tag it as override in log.
- Overrides should be rare. We might even require a reason/comment field when overriding so it’s documented.

- **No silent overrides:** the system itself won’t override rules on its own. (I.e., we won’t auto ignore a rule in edge cases – either the rule is configured differently for that context or a human explicitly overrides.)

- **Examples of Rule Evaluations:**

- *Revision Limit Reached:* When a client tries to request another revision, the rule engine checks `if (deliverable.revisionCount < contract.maxRevisions)`. If false (count >= max), the engine blocks the “New Revision” event. The UI could show a message: “Maximum revision rounds reached – additional changes require a new contract.” The deliverable might move to a Finalization state automatically at this point.
- *Payment Missing:* Attempt to deliver or download final files triggers check `if (invoice.status == Paid)`. If not, block. If a client somehow finds a direct asset URL, the backend serving the file also double-checks payment status before allowing it. This double safety ensures no leak (defense in depth: both UI and backend enforce the rule).
- *Retainer Expired:* If someone tries to book work on March’s retainer hours in April, the rule engine sees that March’s retainer allocation is expired. The action “log design time to retainer” fails with

"Retainer period closed – cannot log time. Please use current month allocation or renew contract." This might be implemented by marking the allocation object as inactive after its end_date, and any reference to it in new task entries is disallowed.

- *RAW Request without Entitlement:* If a client user attempts to click "Download source file" for a deliverable, the rule engine will check the Contract or Deliverable flags: `if (deliverable.rawAccess == true)` or similar. By default it's false, so it blocks. Possibly the UI doesn't even show the button unless `rawAccess` is true. If shown (say the client found an API endpoint), the server still rejects and logs it. The client could then be prompted "RAW files are not included in your agreement" rather than a silent failure. This emphasizes that **the system doesn't negotiate or bend the rules** – it outright refuses the action, backing it with contract policy.

- **Hard-Coded vs Configurable (Evolution):** In MVP, rules like max revisions, required waiver, etc., might be constants or simple contract fields set by devs. Over time, we will introduce a configuration UI where an admin can set these per client/contract. Our rule engine will then fetch rule parameters from the database instead of constants. For instance, the guard becomes `if (deliverable.revisionCount < deliverable.contract.max_revisions)` rather than `< 3`. That `max_revisions` lives in a config. Similarly, "payment required for delivery" might eventually be a yes/no toggle in case a specific agency chooses to allow net terms (though our core principle suggests it's always yes). We suspect most rules are universal for this product (because it's our differentiator to enforce them), so we won't expose toggles to turn them *off* – but we may allow adjusting values (like 2 vs 3 revision rounds). We also foresee adding new rule types as needed (e.g. maybe a rule for "no work starts before deposit paid" could be parameterized by a deposit percentage).

- **Logging & Messaging:** The rule engine will integrate with the UI messaging system. When a rule blocks an action, we log it (for devs/admins) and also give user feedback where appropriate. E.g., if Ops tries to mark delivered with no payment, we immediately show an error "Invoice not paid – cannot deliver." In YouTrack's workflow, they advise using a message to explain blocked transitions [11]; we'll do the same so users understand why an action is unavailable. This messaging is part of user experience to reinforce the authoritative tone (clear, calm statement of the rule in effect).

In design, the rule engine ensures **the system's authoritative stance is consistent** – it doesn't rely on user memory or manual checks. Every rule the agency wants to enforce is encoded and automatically applied by the software. This reduces human error and ensures operational safety (e.g. you literally *cannot* forget to collect payment, the system won't let you proceed). The system's logic is like a referee: impartial and strict, which ultimately improves trust both internally and with clients (the system can even be cited: "Our system doesn't allow us to do that without X, it's standard policy.").

Phase 4: Permission & Access Model (State-Based + Role-Based)

The permission model combines **Role-Based Access Control (RBAC)** with **state-based restrictions**. Roles define *who* can in general do certain categories of actions, while state-based rules define *when* and *to what* those actions apply. This dual approach ensures that even authorized users can only act when the context (object state) permits.

Why RBAC Alone Is Insufficient: Traditional RBAC assigns privileges by user role (e.g. a "Designer" role can edit deliverables, a "Client" role can view and comment). However, RBAC doesn't consider the *state of*

a particular item. In our scenario, a Client might normally have permission to download files (role-based permission), but we *also* need to restrict that by state (only if the deliverable is in Delivered state and paid). RBAC can't express "the client can download if the invoice is paid" because that's dynamic business logic, not just user attribute. Similarly, an Ops user might generally be allowed to transition project states, but we don't want them marking a project as completed if key steps are unfinished. We need **attribute-based conditions** on top of roles. This is essentially an **Attribute-Based Access Control (ABAC)** approach where attributes include the object's state and other flags [12]. State is a prime attribute to check in policies: it introduces context. By combining RBAC with state, we achieve a more fine-grained and situation-aware security model.

State + Role Matrix: Each action in the system will be allowed based on an intersection of: - **User Role** – does this role have the capability *in principle*? (e.g. Only internal roles can edit internal data; Clients cannot change internal statuses. Only Client role can approve deliverables, etc.) - **Object State** – is the object in a state where this action makes sense and is permitted? (e.g. You can only "submit work for client review" if deliverable state is In Progress; you can only "approve" if it's In Review; you can only "deliver files" if deliverable is Approved and invoice Paid.) - **Optional Condition** – sometimes additional conditions like "user is assigned to project" or "deliverable not locked for some reason" could apply, but primarily it's role + state.

The system will likely implement this via checks before executing any action: 1. Check the role of the user vs an action's allowed roles. 2. Check the state of the target entity vs the allowed state(s) for that action. 3. If both pass (and any other guard conditions), allow; otherwise deny.

Dynamic UI and Actions: The UI will reflect this combined permission logic by dynamically enabling, disabling, or hiding controls: - If a user lacks the role for an action, that action button or page is not visible at all. For instance, a Client user won't even see internal project status change buttons. - If the user has the role but the item's state isn't appropriate, the button may be visible but disabled or annotated (e.g. "Deliver Files" button is shown for an Ops user but greyed out with a tooltip "Waiting for payment" until payment received). This way, the UI communicates *state-related locks* clearly. - Example: A **Designer (Digital role)** opens a deliverable. They normally can upload files and mark as ready for review. But if the deliverable's current state is "Awaiting Client Approval" (meaning it's with client), the designer shouldn't alter it. The UI might disable the "Upload Revision" button until the state goes back to In Progress (if client requested changes). - Example: A **Client** viewing a deliverable: Before approval, they might have a "Request Revision" or "Approve" option. Once they click Approve (deliverable goes to Approved), those options disappear (can't request revision after approval unless perhaps an Ops reopens it). The "Download Files" option might remain disabled until Ops actually marks it Delivered (or until invoice is paid). - The UI thus always queries both user permissions and object state to decide what to show.

Preventing Accidental Leaks: State+role permissions are critical for things like asset access: - A **Client role** user should only access assets when deliverable state = Delivered. The system will enforce this on the backend (e.g. checking JWT claims for role and then checking DB for deliverable state before serving file). But also on the frontend, the download link only appears once conditions are met. This two-layer check prevents any accidental or malicious access. For instance, even if a client somehow guesses a file URL before delivery, the file service will verify "is this deliverable delivered & paid & does user have client role on this project?" before allowing it. If not, it returns a 403. - **Internal confidentiality:** Perhaps RAW assets or internal notes are visible to internal team roles (Studio, Digital, Ops) but never to Clients. Those UI elements are entirely hidden from Client view (role-based hidden) and server will double-check if someone tries an API call. This stops any data leak of something like an Adobe source file or an internal comment thread. - Another example: **Communication logs** (if any) might be internal until a

project is in a state ready to share with client. Only when project state = “Client Portal Open” or such do we show certain info to client role.

Role Hierarchies and Exceptions: Typically Owner would have the broadest access (including viewing all financial info, overriding rules). Ops would have broad project control but maybe not see billing details depending on how we define. Designers (Digital) have edit access on creative content but not on contract or billing modules. Studio role maybe only sees the studio schedule and relevant project info. Client obviously only sees their own projects’ high-level progress and deliverables, not internal workflows or other clients. - However, **state can override role** in the sense that even an Owner cannot perform an action that is disallowed by state without explicitly doing an override (which we treat as a separate action). E.g., if a deliverable is locked due to an overdue invoice, an Owner won’t see the normal “Deliver” button active either – they would instead use an “Override and Deliver” action. This is deliberate to maintain consistency; it prevents mistakes. The system basically says “this item is locked” to everyone, but provides the Owner a special key if needed. - Conversely, state can grant a temporary permission: e.g., when a deliverable is in “Client Review” state, the Client role gains access to an “Approve” action (which wouldn’t be relevant in other states). That’s a state-scoped permission for that role.

Implementing State+Role Rules: - At the code level, one can implement this by attaching role requirements to transition definitions. For example, the state machine transition “Approve deliverable” requires: from In Review -> to Approved, guard: user.role == Client. If a designer accidentally calls that transition, the guard fails (or the UI never offered it). - We could also implement a generic permission check function that every API call or UI action triggers: `canUserDo(action, object)` which internally checks `user.roles ∈ allowedRoles && object.state ∈ allowedStates && otherConditions`. This could be data-driven by a policy matrix for maintainability.

Example Scenarios: - **Deliverable Delivery:** Allowed if user.role ∈ {Ops, Owner} *and* deliverable.state == Approved *and* invoice.paid == true. If role passes but invoice not paid, it’s blocked by state/condition. - **Client Feedback Submission:** Allowed if user.role == Client *and* deliverable.state == In Review. Outside that state, a client shouldn’t be able to submit feedback (e.g. if deliverable is not yet shared or already finalized). - **Moving Project to Archive:** Maybe only Owner role can archive, and only if project.state == Completed. So an Ops user cannot archive mid-project; they’d need to complete it first. - **Viewing Financial Info:** Perhaps only Ops/Owner can see budget utilization, invoices, etc. Those UI sections are permission-limited by role, not by state (since an invoice’s state doesn’t grant more people access; it’s purely role-based in that case). So we do still have pure RBAC areas (like modules only accessible to certain roles).

Why State+Role is Secure: It effectively implements a form of **policy-based access control** where policies can include resource attributes (state, client ownership, etc.) in addition to subject attributes (role). This dynamic model is known to be more flexible and secure than static RBAC [12]. It reduces the need to create overly granular roles to capture combinations of conditions (e.g. we don’t need a special “PaidClient” role to represent clients who have paid – we just check payment status attribute each time). Instead of role explosion, we keep roles relatively coarse and rely on runtime checks.

In summary, **RBAC gives the base authority, State gives the situational authority.** Both must align for an action to happen. The UI will be a direct reflection of this: it will *always* show the user what they can do at the moment, nothing more. This model prevents both unauthorized access and also *mistakes by authorized users* by disabling actions at the wrong time. It treats the UI as an extension of the rule engine – essentially making the interface state-aware and context-sensitive, which contributes to operational safety (users are guided to do the right thing and cannot easily do the wrong thing).

Phase 5: MVP Decomposition (Realistic Scope)

We define a **true MVP** that a small team can build and deliver quickly, focusing on solving the most pressing pains (enforcing core rules) and deferring more complex or nice-to-have features. The MVP will prove the concept of rule-enforced operations without trying to cover every edge case or customization.

MVP-Critical Modules & Features: 1. **Project & Deliverable Workflow Enforcement:** The primary pain point is uncontrolled scope creep and payment issues. So MVP must include the project/deliverable entities with state machines enforcing “no payment, no delivery” and “limited revisions” from day one. This means: - Deliverable creation, uploading final files, client approval workflow, and final delivery gating by payment. The UI for deliverables (internal view to manage, client view to review/approve, download link when allowed) is critical. - Basic Project tracking with status, mainly to roll up deliverable statuses and possibly to block project completion until all payments done. - **Revision management:** simple counter of revision rounds with a setting per project/contract. If max is reached, system stops further revision requests. (For MVP, this max can be a constant or a contract field set via admin – no need for a full UI for client to pick, it can be part of contract entry by an Ops user.) 2. **Contracts & Payment Integration (Simplified):** The MVP should ensure that an invoice or payment status can be recorded and tied to deliverables. This could be as simple as: - An admin interface for Ops to mark an invoice as paid (initially, we might not integrate a payment gateway in MVP; Ops can flip a “Paid” toggle when they confirm money arrived). - Linking deliverables to an invoice ID or payment required flag. Possibly just a checkbox “Paid” on deliverable for MVP to simulate the effect. However, better to at least have an Invoice object even if created manually. Ensuring that the deliverable delivery button is disabled until that paid flag is true. This addresses the biggest pain of not delivering before payment. 3. **Basic Rule Engine/State Enforcement:** Implement the core state machine logic for deliverables and revisions in code. This is MVP-critical to demonstrate the category’s value. Hard-code the rules that: - Deliverable cannot go to Delivered state if not paid. - Only X revisions allowed. - (Potentially include retainer expiry if applicable to the first target use-case, but if not immediately needed, could defer.) - Essentially, the core “gatekeeper” behavior must be present for the main content delivery process. 4. **Client & Team Portal:** Need minimal UI for two sides: - Internal view: Ops/Designers can create projects, upload work-in-progress, send for client review, etc. This interface must enforce sequential handoff (e.g. a designer marks deliverable done -> triggers client review state). - Client view: where client can log in, see their deliverables, perhaps a simple Kanban or list with statuses, download final files when available, and provide approvals or revision requests. The client portal must reflect the enforced rules (if something’s waiting on payment, show a notice or disable actions). - **Authentication and Role management** is needed (at least stubbed) to differentiate client vs staff users in the UI. 5. **Audit Logging (basic):** At least have a backend log of actions for debugging. A full UI for audit logs is not MVP, but ensure we record data like state changes and who did what in the database (so we don’t lose info). This can later surface in an admin panel. 6. **Design System implementation:** Use the NOCO design standard for the interface from the start (colors, typography etc. from Phase 6). This is important because the product tone must be established early (serious, professional). The MVP UI should already reflect clarity and authority. We won’t build every component variation, but ensure the main pieces (navigation, buttons, forms, modals) follow the style guide.

Deferred (Post-MVP) Modules: - **Studio Booking Module:** Managing studio schedules and waivers can be complex and might not be the most immediate pain for MVP. We can defer the full scheduling calendar, etc., unless NOCO’s primary operations involve constant studio bookings. We assume project deliverables and payments are a bigger issue to solve first. Studio booking could come in a later iteration with its rule enforcement (waiver signing workflow). - **Advanced Retainer Management:** For MVP, we might handle retainer hours in a simplistic way or even manually. If current pain is just deliverables and revisions, retainer expiry enforcement could be Phase 2. We might not implement automatic month resets initially – an Ops person could manually track hours in MVP. However, if NOCO

specifically sells retainers, we might include a basic counter that resets monthly. It depends on immediate need; we optimize MVP for core flow first. - **Configurability & Multi-tenancy:** MVP will have rules hard-coded or configured via config files. A polished UI to edit rule settings (like changing revision limit per project via GUI) can be added later. Also, multi-tenant support (multiple agencies with isolated data) is not needed if launching just for NOCO. So we'll assume one tenant in MVP. Down the line, we will add an organization model to separate data. - **Payment Gateway Integration:** Instead of integrating Stripe/PayPal in MVP, we might simulate payment (Ops marks as paid when money is received externally). The core need is to enforce gating, which we can show with a manual toggle. Actual online payment integration can be added once the workflow is validated. - **Comprehensive Notifications & Emails:** While some basic email (like "deliverable ready for download") may be useful, we can start with just in-app status changes. Fancy email sequences or reminders (like "invoice overdue" reminders) can come later. - **Mobile Optimizations:** The MVP can be desktop-focused for agency and client portal. Mobile-responsive design is a should-have given many clients might want to approve on phone, but if resources are tight, we ensure the layout is responsive but perhaps not perfect on MVP. (Since the design system is defined, making it responsive is straightforward with the given grid). - **Full Audit Log UI:** Initially, logs in database or console is fine. Later, an admin interface to view history can be built.

Hard-Coded First vs Configurable Later: - As mentioned, things like **max revision rounds, included or not** – MVP: perhaps a global constant (e.g. 2 rounds for all projects) or set per contract via JSON. Later: a field in a Contract creation form. - **Enforcement on/off toggles** – We likely won't allow turning off critical rules, since that defeats the product's purpose. But if needed, those could be config flags in future (e.g. an agency wants to allow delivery before payment for certain trusted clients – they could turn off that rule for that client's contract as a special case in future, or use override). - **User role definitions** will be somewhat static in MVP (predefined roles with specific code checks). Later, we might allow custom roles or changing permissions if needed by different organizations. - **UI Text and Language:** The product language is Turkish, so MVP would ideally have all UI text in Turkish (since NOCO and clients are presumably Turkish-speaking in Istanbul). For MVP, developers can hard-code Turkish text strings following the tone guidelines. Later, if going SaaS internationally, we'd externalize strings for localization. But MVP specifically should just use Turkish consistently from the start to avoid later overhaul. (This is a content localization aspect rather than functionality.)

By focusing on **deliverables, revisions, approvals, and payment gating**, the MVP directly addresses scope creep and payment leakage – likely the biggest pains. This immediately provides value: even with minimal bells and whistles, if the system simply *stops the team from delivering unpaid work and limits revision rounds*, it's solving a real problem. Other features, while useful, can be layered on once this core is validated.

This MVP approach avoids over-engineering: we're not building a full CRM or scheduling app first, we're building the unique enforcement mechanism. It's **simple and functional** – deliver the core value quickly [1]. That also de-risks the project; we prove that the concept works for NOCO's own workflow before expanding.

Phase 6: UI/UX System (NOCO Digital Standard Compliance)

The UI design will strictly follow the NOCO DIGITAL design standards provided. The aesthetic is serious, clear, and authoritative – visually reinforcing that this system means business and enforces rules. Below we outline the key style guidelines and how the UI will use them:

Color System

- **Primary Color:** `#329FF5` – a strong blue used for primary actions, highlights, and active states. All primary buttons and links use this color (or its variants on hover). It conveys trust and authority in the interface.
- **Secondary/Accent Color:** `#00F5B0` – a teal green accent for secondary actions or highlights (e.g. toggle switches, accent borders). Use it sparingly (~15% of the interface) to draw attention to interactive elements or status highlights that are not primary actions.
- **Highlight Color:** `#F6D73C` – a yellow highlight used to call attention to warnings or important info (e.g. "payment overdue" badge, or highlight text for emphasis). This should be used even more sparingly (like for a warning icon background or emphasis text).
- **Neutral Colors (Ink/Surface):**
- Ink (main text color): `#111827` which is a near-black for high contrast text on light backgrounds. Almost all text will be this color for maximum readability.
- Sub-ink (secondary text): `#374151` – a dark grey for secondary information or labels, to create hierarchy in typography.
- Surface (app background): `#F8FAFC` – a very light grey for page backgrounds. It's easy on the eyes and not pure white, providing a subtle warmth.
- Card/Panel background: `#FFFFFF` (white) for cards and surfaces that sit on the background. This ensures cards stand out on the slightly grey surface.
- Border color: `#E5E7EB` – light grey for dividers and component borders (inputs, card outlines). It's subtle to avoid a heavy boxed look, but enough to delineate sections.
- **Feedback Colors:**
- Success: `#10B981` (green) for success messages or status (e.g. "Payment Received" or "All Good" indicators).
- Warning: `#F59E0B` (orange) for non-critical warnings (perhaps for things like "Retainer hours expiring soon").
- Error: `#EF4444` (red) for errors or rule violations (e.g. "Payment Required" error state). Red will be used carefully to draw attention when something is wrong or blocked.
- **Brand Gradient (optional use):** A gradient from Primary to Accent (`#329FF5 → #00F5B0`) at a 135° angle, 60% opacity on top layer. This could be used in the application's branding, like the login page background or as a highlight overlay in graphics. It's not a main UI element but can appear in marketing or splash screens for a modern touch.
- **Color Usage Ratios:** Approximately: Ink/dark text 40% of UI, light surfaces ~35%, Primary ~15%, Accent/Highlight combined ~8%, others ~2%. This means the interface will appear clean and predominantly neutral with measured splashes of color for calls-to-action and status. Nothing should be overly colorful or decorative – color always has a purpose (like indicating an actionable element or a status that needs attention).
- **Contrast and Accessibility:** All text will have sufficient contrast: body text color on white is ~#111827 which yields a contrast ratio well above 4.5:1 on white (meeting WCAG AA for normal text). Headings (which might use a slightly lighter dark grey for aesthetic) will still exceed 3:1 as required for large text. We will test key color combinations (e.g., Primary blue on white for buttons, etc.) to ensure accessibility standards are met.

Typography

We use a combination of **Darker Grotesque** for display headings and **Inter** (or **Manrope**) for body and UI text: - **Headings (Display font):** Darker Grotesque, weight 600–700 (semi-bold to bold). This grotesk sans-serif gives a strong, confident feel to titles. - H1 – **56px** font size, **60px** line-height, with -1% letter spacing (a slight tightening). Used for top-level page titles (e.g. "Project Overview" page title). - H2 – **40px** size, **44px** line-height (used for section headers or important headlines on a page like

“Deliverables” section). - H3 – **32px** size, **36px** line-height (subsection headings or card titles perhaps). - All headings are single-line (no multi-line fancy display; if text is long we’ll use H2 or H3 instead of cramming into one H1). - **Body and UI text:** Inter or Manrope (both are clean, modern sans-serifs known for readability on screens). Various weights (regular 400, medium 500 for emphasis, semi-bold 600 for small headers or important labels). - Body (M) – **16px** font, **24px** line-height. This is the default paragraph text, used for descriptions, regular content. It’s comfortably readable. - Body (S) – **14px** font, **22px** line-height. Used for secondary text, notes, or perhaps the client comment text, etc. - Caption – **12px** font, **18px** line. For helper text, timestamps, or very secondary info. - Button text – **14px** font, **16px** line-height, usually medium or semi-bold (600). Buttons should have clear, short labels in sentence case. - **Scale for Mobile:** On smaller screens, we scale down: - H1 ~40px, H2 ~32px, H3 ~24px for better fit. The rest of text can remain at 16px/14px as those are already mobile-friendly for body. - **Styling:** - Headings will use the ink color (#111827 or maybe slightly lighter for aesthetic but still high contrast) for maximum readability. We may use the sub-ink color (#374151) for subheadings or less prominent headings. - Secondary text (like meta info under a heading) will be in #6B7280 (a grey from Tailwind’s palette) as given – that is our designated secondary font color for descriptions or labels, keeping them distinct from main content. - Links: They will appear in the Primary blue (#329FF5), at a medium weight (500–600). No underlines by default (to keep it clean), but will have a distinguishable style (the color itself, and maybe an icon). On hover, make them slightly bolder (e.g. 700) or a shade darker, with an underline on hover if needed for clarity. This way they stand out but maintain the clean look.

Overall, typography will be clean and professional, with adequate line spacing to avoid crowding. We’ll ensure no giant blocks of text; break paragraphs and use lists to improve scan-ability (aligning with the short paragraph rule).

Grid, Spacing, and Radius

The design follows an **8pt grid system**: - Spacing increments are multiples of 8px for consistent padding/margins (e.g. 8, 16, 24, 32px etc.). This creates visual harmony and predictability. For instance, form fields might have 16px vertical spacing between them (2 units), sections might have 32px top margin, etc. - **Layout Container:** Max width ~1200px. Content will center in the viewport, so on large screens it doesn’t stretch too wide (ensuring line lengths are readable). - **Grid system:** 12-column grid with 24px gutters. We’ll use this for page layouts – e.g. a dashboard might have cards spanning certain columns. On smaller screens, columns collapse. This standard responsive grid helps arrange data-dense UI (like tables, lists, sidebars). - **Border Radius:** We use subtle rounding to keep the UI modern but not overly whimsical: - XS radius: 4px (for small elements like badges, maybe input corners). - S: 8px – likely our default for buttons, input fields (slight rounding). - M: 12px – maybe for cards or modals. - L: 16px – larger containers or the overall panel corners. - XL: 24px – perhaps used in very large surfaces or special cases (maybe a hero banner or something, though in such a business app not many large rounded containers are needed). - Likely, we’ll consistently use one radius for most components (e.g. 8px or 12px) to keep a uniform look, but the values above define the available scale. - **Elevation (Shadow):** Minimal use: - Z1 (small elevation): shadow `0 1px 2px 0 rgba(0,0,0,0.06)` – a very light shadow for things like active buttons or input focus or small cards. Barely noticeable but adds depth. - Z2 (medium elevation): shadow `0 4px 12px 0 rgba(0,0,0,0.10)` – a slightly stronger shadow for modals or dropdowns to lift them above content. - We avoid heavy or multiple layered shadows to maintain a clean, flat aesthetic with just enough depth to separate layers. - All spacing decisions (padding in cards, margin between elements) will stick to the 8pt rule (e.g. a card might have 24px padding, a form group maybe 16px padding inside a section, etc.) which ensures rhythmic whitespace and a calm appearance (no awkward tight or too loose spots).

Component Styles

Buttons: - **Primary Button:** Filled with the Primary blue `#329FF5` as background, and white (#FFFFFF) text. This is used for the main CTAs (e.g. “Deliver Now”, “Send for Approval”). It stands out strongly on the neutral background. Example: a “Approve” button for client will be primary style. - **Secondary Button:** White background with a 1px solid border in `#E5E7EB` (grey) and text in ink color `#111827`. This looks like a neutral or cancel action button. E.g. a “Cancel” or “Close” or less important actions use this style. It blends with cards because it’s white, but the border and text make it clearly a button. - **Hover states:** Buttons will darken by ~8% on hover – Primary blue becomes slightly darker blue, Secondary button’s background might become slightly off-white/grey to show it’s active, border might darken. No excessive effects, just a subtle color darken to indicate interactivity. - **Focus state:** We will show a focus ring (for accessibility when using keyboard or focus): a 2px outline in the primary color at 40% opacity (#329FF5 at 0.4). This creates a visible blue glow around the button indicating focus, without shifting layout (we can use outline or box-shadow). This focus style is for both primary and secondary buttons. - **Sizes:** - Large: 48px height, with ~20px horizontal padding (so a large button might be 48px tall, padding ensures wide enough click area even if label is short). Font likely 14px bold as mentioned. - Medium: 40px height, 16px horizontal padding. This will be the default size for most buttons (good for forms, dialogs). - (Small not explicitly listed, but if needed, could be 32px height with 12px padding, but we’ll mainly stick to M and L to ensure the >=40px hit area as required.) - Buttons will have slight radius (maybe 8px for medium buttons, and maybe the same 8px for large – consistent look). They’ll also possibly use the shadow Z1 when hovered or active to give a slight lift impression.

Forms (Inputs, Selects, etc.): - We apply a consistent style: 1px border in `#E5E7EB` (light grey) around input fields. Background white. Text in ink color. - Padding inside inputs: 12px padding (vertical and horizontal) inside the field, so text isn’t jammed against edges. This also makes the input height around 40px (if 12px top+bottom padding + ~16px text). - Border radius maybe S (8px) so inputs have slightly rounded corners matching buttons’ look. - On focus: the border (or an outer outline) will change to a 2px solid #329FF5 at 40% opacity (similar to button focus ring). This highlights the field that’s active in a calm way. Also possibly show a subtle shadow (0 0 0 4px rgba(50,159,245,0.2)) for a glow effect. We won’t rely on just color change (which might be hard to see); a thicker border or shadow indicates focus clearly. - Labels for inputs will use the secondary text color (#6B7280) and maybe Body S size (14px) and placed above the field for clarity. - Placeholder text, if any, in inputs will be #9CA3AF (a lighter grey) for hinting.

Tables: - Tables will be common for listing projects, tasks, etc., so we style them for clarity: - Table header row background: use the Surface color (#F8FAFC or perhaps a slightly stronger grey like #F3F4F6) to distinguish it. Header text in sub-ink (#374151) bold. - Table grid lines: use Border color (#E5E7EB) for cell borders or row separators. Or we might opt for zebra striping instead of full grid lines for a cleaner look. - Zebra stripes: alternate row background as #F3F4F6 (a very light grey) for better readability in wide tables. That color is given as a zebra in guidelines. This helps track rows without heavy grid lines. - Alignment: Text content left-aligned (especially names, descriptions), numeric or quantitative data right-aligned (e.g. hours, amounts) so they’re easy to compare in columns. This is explicitly noted and we will adhere to it for a professional look. - Table cell padding: about 12px horizontal and 8px vertical padding inside cells to give breathing room. - No overly condensed tables – ensure line length is reasonable. Possibly limit the number of columns or use responsive stacks on mobile.

Cards: - Cards are used for grouping information (e.g. a Project summary card). - Background: White (#FFF) with maybe slight shadow (Z1) to lift above the page background. - Border radius: L (16px) per standard, giving a softer rectangle. - Inside padding: likely 16px or 24px depending on card size, to ensure content isn’t cramped. - If cards include images (like perhaps a thumbnail of an asset or client

logo), maintain a consistent aspect ratio like 16:9 or 4:3 as recommended for media within cards. This ensures a tidy, grid-aligned appearance. - Use the provided color palette for any status tags or icons inside the card (e.g. a status chip “Approved” in green background #10B981, or “Overdue” in error red #EF4444 etc., with white text, small rounded label).

Icons & Imagery: - Icons will use a **1.5px stroke width** (meaning line icons, not solid fills, matching a modern UI style). Rounded line caps and joins for a friendly but professional look (no sharp spike edges). - We’ll use a consistent icon pack (perhaps Feather icons or similar) that meets these criteria. Icons in buttons or labels help clarity (like a small lock icon next to a disabled Download button to indicate restriction). - Icon color: typically use the ink color (#111827) for default icon, or if icon signifies something with a status, use the status color (e.g. red error triangle icon for an error message, green check for success). - Photos (if used, say in case study or user avatars) should be natural, high-contrast, no funky filters – aligning with the “no-nonsense” tone. Likely, user avatars might just be initials in a circle given the B2B nature, but any imagery (perhaps on an onboarding screen) should look real and clear. - Company Logo usage: ensure there’s clear space around it equal to at least the height of the logotype. On colored backgrounds, use a white version of the logo as specified (common branding rule to maintain visibility).

Motion and Interaction: - Animations should be minimal and purposeful. Use standardized easing and duration: - **Duration:** ~160ms to 240ms for most UI transitions. Short enough to feel snappy, but not instant. For example, a modal appearing, a dropdown opening, or a toast message fading in. - **Easing:** use `cubic-bezier(0.2, 0, 0.2, 1)` which is an ease-out with slight acceleration at start and deceleration at end (this is similar to Material Design’s standard easing). It gives a natural feel. - **Transition style:** when elements appear/disappear, prefer a *fade (dissolve) combined with a slight upward motion of 8px*. So a menu might fade in and move up a tiny bit from below, which is a gentle entrance. This matches the specified “dissolve + 8px up” guideline. It creates a polished feel without being distracting. - No fancy parallax or overly complex animations – they would contradict the calm, professional tone and also can confuse or slow down a user in an operational tool. Keep it simple: things should appear and disappear cleanly, give feedback like button click ripple maybe very subtly, etc. - Focus transitions: when user navigates or completes an action, we might highlight the result (e.g. after clicking “Send for Approval”, maybe the deliverable card gets a brief highlight or slides into the “Waiting for Approval” column). But again, subtlety is key – maybe a brief background highlight (in the highlight yellow #F6D73C at 20% opacity) that fades out to draw attention to the changed item.

Accessibility Considerations: - All interactive elements have at least a 40x40px hit area. We’ll ensure even small icons in a button have padding to meet this. This is crucial for both touch and for users with limited dexterity. - Keyboard navigation: The focus ring (primary color 2px outline) will be implemented on all focusable elements (buttons, links, inputs). We won’t remove outlines without replacement. This ensures users navigating via keyboard can see where they are. - Color use will always consider color-blind users: never rely solely on color to indicate a status. For example, an error message not only is red, it also has an icon or text label “Error:” so it’s clear even if the user can’t see red. Same with required fields (we might use an asterisk and not just red border). - Typography line length: We will keep paragraphs to a comfortable width (45–75 characters per line). Our 1200px container and typical 16px font should naturally enforce this, but if a text block is too wide, we might arrange it in columns or limit the width of text elements. Long form text isn’t a huge aspect of this app (it’s more data and labels), but we ensure any descriptions or contract terms displayed follow this rule for readability. - Headings will be concise (aim for 6–10 words max) per guideline, which aligns with clarity – we don’t want heading text wrapping multiple lines or being verbose. - Language: since the app is in Turkish, we will use clear formal Turkish phrasing, no slang, to maintain professionalism. We ensure the text is understandable and not overly technical for clients.

Light & Dark Mode: The design will support both, though MVP might focus on Light mode. Still, we define it: - **Light Mode (default):** - Backgrounds: Card #FFFFFF, overall app background #F8FAFC (near white). - Ink text #111827 as primary on light background. - Border #E5E7EB as used above. - This yields a bright, clean interface. - **Dark Mode:** - Backgrounds become dark navy/black tones: Card #0B1220 (very dark blue) as surface, and a slightly lighter (but still dark) #111827 for page background maybe – here they gave Card specifically #0B1220. - Ink text becomes #E5E7EB (the light grey) for high contrast on dark (so essentially we invert the scheme). - Border: #243041 (a dark grey) for outlines and dividers on dark mode (this is subtle but visible against the almost black card). - The Primary (#329FF5) and Accent (#00F5B0) colors remain the same in hex (we don't change them for dark, which is okay because they are quite vivid and will pop on dark backgrounds too). They may visually appear brighter on dark, but that's usually fine at 100% usage. We might tone them down slightly if needed by opacity. - Images in dark mode: increase brightness by 2–4% if needed to ensure they don't appear too dim against the dark background. (E.g., a logo or any photograph might need a slight gamma adjustment in CSS or providing a brighter asset). - We'll supply a white version of the company logo on any dark toolbar or login screen background to ensure it's visible (per brand rule). - All other UI components invert accordingly (e.g. inputs on dark mode would be dark background with lighter border, etc.). We will ensure contrast still meets standards (white text on #0B1220 background is fine). - The design tokens we define will have both modes values, making it easier to switch. Users can choose dark mode if they prefer, without losing clarity or professionalism.

Throughout the UI, we enforce a **calm and authoritative tone**. This means: - No gaudy graphics or gimmicks. The interface is content-first with a light aesthetic layer from the color scheme. - Messaging is clear and direct. For example, instead of "Oops, you can't do that right now :)", we'll say "Bu eylem, ödeme yapılmadan gerçekleştirilemez." ("This action cannot be performed before payment.") – polite but firm. - The overall look should give users (especially clients) confidence that this system is robust and that the agency is organized and professional. Internally, it gives team members clarity on what's next and what's blocked.

By adhering to the provided design rules, we ensure consistency and a high-quality UI from the start. It also accelerates development (tokens and components are pre-defined in spec). The serious but modern design will set this product apart from generic tools – it visually communicates *"this is a reliable, professional operations system, not a casual app."*

Phase 7: Figma Structure & Design Tokens

To implement the above design systematically, we will set up our design files and token libraries as follows:

Figma Styles Structure: We'll create shared styles in Figma categorized by usage, naming them in a hierarchical style for clarity (as suggested): - **Color Styles:** Organized by role/purpose and name. For example: - `Color/Role/Primary` – #329FF5
- `Color/Role/Accent` – #00F5B0
- `Color/Role/Highlight` – #F6D73C
- `Color/Role/Ink` – #111827 (main text)
- `Color/Role/SubInk` – #374151 (secondary text)
- `Color/Role/Surface` – #F8FAFC (bg)
- `Color/Role/Card` – #FFFFFF
- `Color/Role/Border` – #E5E7EB
- `Color/Feedback/Success` – #10B981
- `Color/Feedback/Warning` – #F59E0B

- `Color/Feedback/Error` – #EF4444
- and similarly color styles for dark mode (Figma can handle separate libraries or a toggle with “Color/Mode/Dark/Card” etc., or using variables in new Figma update). - **Text Styles:** Under a prefix like `Type/` : - `Type/H1` (56px, Darker Grotesque 700, line 60px, tracking -1%)
- `Type/H2` (40px, DG 700, 44px line)
- `Type/H3` (32px, DG 600, 36px line)
- `Type/Body M` (16px Inter 400, 24px line)
- `Type/Body S` (14px Inter 400, 22px line)
- `Type/Caption` (12px Inter 400, 18px line)
- `Type/Button` (14px Inter 600, 16px line, all-caps or sentence-case depending on decision, likely sentence-case per clarity)
- Additional styles for maybe `Type/Label` (if we want a semi-bold small label style). - **Effect Styles:** e.g. `Effect/Shadow/Z1` and `Effect/Shadow/Z2` as defined: - Z1: 0px 1px 2px rgba(0,0,0,0.06)
- Z2: 0px 4px 12px rgba(0,0,0,0.10)
These styles can be applied to frames or components as needed. - **Grid Styles:** We might set up a Figma layout grid for frames: - `Layout/Desktop 12col` – 1200px container, 12 cols, 24px gutter, margins auto. - `Layout/Mobile 4col` (for example) – something like 4 columns, 16px gutter, 16px margin for smaller screens. - **Components in Figma:** We will create a library of components (buttons, inputs, modals, etc.) that use the above styles and can be reused. Naming might follow something like `Components/Button/Primary/Large` etc. All components will adhere to the design tokens, so if a token color changes, it updates everywhere.

Design Tokens (for code): We will extract the finalized style values into a JSON (or similar) token file for developers. Named as `NOCO_UI_Tokens_vX.Y.json` (for example, `NOCO_UI_Tokens_v1.0.json` for the first version). This file would include:

```
{
  "color": {
    "primary": "#329FF5",
    "accent": "#00F5B0",
    "highlight": "#F6D73C",
    "ink": "#111827",
    "subInk": "#374151",
    ...
    "error": "#EF4444"
  },
  "font": {
    "heading": "Darker Grotesque",
    "body": "Inter"
  },
  "textStyle": {
    "H1": { "fontFamily": "Darker Grotesque", "fontSize": 56, "lineHeight":
60, "fontWeight": 700, "letterSpacing": -0.01 },
    ...
    "BodyM": { "fontFamily": "Inter", "fontSize": 16, "lineHeight": 24,
"fontWeight": 400 }
  },
  "radius": { "xs": 4, "s": 8, "m": 12, "l": 16, "xl": 24 },
  "shadow": {
    "z1": "0 1px 2px rgba(0,0,0,0.06)",
```

```
    "z2": "0 4px 12px rgba(0,0,0,0.10)"
  },
  "spacing": 8,
  "breakpoints": { "sm": 640, "md": 768, "lg": 1024, "xl": 1280 }
}
```

This is just illustrative – the idea is to have a single source of truth for all these constants that both Figma and the code refer to. Tools exist to sync Figma styles to such token files, or we can manually maintain it initially.

Versioning & File Naming: - The token file name includes version so we track changes in the design system over time (e.g. if we adjust colors or spacing in future, we bump version). - We'll also prepare a design presentation deck for stakeholders using the brand. The file naming for that is given as `NOCO_Present_vYYMMDD.pptx` – likely a PowerPoint template or pitch deck showing the design. For example, if we present the UI concept on Jan 30, 2026, the file might be `NOCO_Present_v260130.pptx`. This naming convention ensures all presentations are versioned by date. We will follow that for any slides we prepare (though the main blueprint deliverable is this document, the PPT might be for design review or onboarding material). - The Figma file itself could be named similarly or just stored in the design team's space. The main point is to keep a tidy structure: - Figma pages for each major part (e.g. "Style Guide", "Components", "Screens", etc.). The style guide page will list all these tokens visually for quick reference.

By establishing this structure, we enable efficient collaboration: designers can tweak a style in Figma and quickly update the tokens; developers can refer to the JSON or Figma inspect for exact values. It also makes scaling the design easier (e.g. when dark mode or new themes come, we add tokens and variants rather than redesigning from scratch).

This tokens approach is part of treating the design as a **system**, which aligns with the overall product philosophy of rules and consistency. Just as our app has strict rules, our design has strict standards – and we implement them systematically so the UI always remains consistent and professional.

Phase 8: Evolution & Failure Modes

Finally, we consider how the system will evolve from a single-agency solution into a broader SaaS product, what pitfalls to avoid, and how to ensure longevity without becoming an unfocused tool.

Evolution from Single-Agency to Multi-Agency (SaaS): - In the beginning, the system is tailored for NOCO's internal use. To evolve into a multi-tenant SaaS, we must introduce **tenant isolation**. This means every record (clients, projects, users) will be scoped by an Agency or Organization ID. Architecturally, this can be done by adding an `agency_id` column in tables and filtering data by it for each logged-in user. We'll add an admin layer to manage multiple agencies eventually. - Configuration per agency: The rule engine should evolve to allow different settings per tenant. For instance, Agency A might allow 3 revisions by default, Agency B only 1. This likely involves creating a settings panel where each agency's admin can set their rules (within allowed parameters). Initially, we might hard-code NOCO's rules. As we onboard another agency, we refactor those into a config model. We should design the code to make adding this straightforward (e.g. no scattered magic numbers; use constants that later map to database fields). - Multi-agency brings scaling concerns: more users, more data. We should ensure from the start that heavy operations (like audit logs or file storage) can scale. Perhaps adopt cloud storage for assets (AWS S3 or equivalent) rather than local, to handle multiple clients. Use an architecture that can move to a cluster if needed. But not over-engineer now – just keep it in mind. - A

SaaS offering will also require robust **access control** where each agency only sees their data. We have to thoroughly test that and maybe implement an organization switch or separate subdomains for each client in the future. - **Onboarding new agencies:** possibly provide templated rule sets (like a default that matches NOCO's policies, which presumably are best practices). Agencies could then tweak if needed (maybe some agencies would turn off certain rules if they work differently, but that might reduce the core value – this will be a business decision whether rules are flexible or non-negotiable platform standards). - As it evolves, documentation and training will be needed since not all agencies will be used to a strict system. We might build in more guidance UI (tooltips explaining rules, etc.) to help adoption across different organizations.

What Not to Build Early (Avoid Premature Complexity): - **Don't build a full Project Management suite:** It's tempting to add tasks, Gantt charts, resource planning, etc. But many such tools (ClickUp, Asana) exist. Our unique value is the enforcement of policies. So we should not spend time early on building generic PM features like subtasks, time tracking, chat, etc., unless they directly tie into the rules. For example, we include revision tracking because it's a rule, but we might not include a general task Kanban board for internal process beyond what's needed to enforce sequential flow. Avoid creeping into being a generic tool. - **Avoid overly flexible workflows:** The system should be opinionated. We shouldn't early on build a "workflow builder" where agencies can define every state themselves – that could weaken the enforced standard and also is a huge build effort. Initially, define a best-practice flow (maybe allow minimal customization later). This is not a generic BPM engine for any process; it's targeted to creative operations. - **No extensive reporting/analytics at start:** Useful, but not core to rule enforcement. Basic reports (like revision counts, hours used) might come for internal use, but fancy dashboards and profitability charts can wait. - **No multi-language/internationalization (beyond Turkish) initially:** Focus on the Turkish use case for NOCO. Internationalization can be done later if SaaS expands globally, but doing it now adds overhead (translating all UI, etc.). We keep it in Turkish for now as specified. - **No mobile app initially:** A responsive web app is fine. A separate mobile app is a project of its own; likely unnecessary until we have many users or specific mobile needs. If clients absolutely need quick approvals on phone, the responsive web can cover it in MVP. Native apps can come much later if demand exists.

Where Teams Usually Break Such Systems: - **Bypassing the System:** The biggest risk is the team reverting to old habits when the system feels too strict. For example, if an Ops person is under pressure, they might be tempted to send files via email outside the system to bypass a payment block. This undermines the system. To avoid this, we need to make the system easy enough to use and *clearly beneficial* so they stick with it. Also, management buy-in is crucial – leadership must enforce that "if it's not in the system, it didn't happen." We'll encourage process discipline. - **Overriding too often:** If overrides (meant to be rare) become common, the system's authority erodes. This can happen if rules are set unrealistically (too rigid) or not updated with business changes. To mitigate: make sure rules configured match real agreements (e.g. if sometimes payment is Net 15, allow that rather than forcing upfront). Also, track override frequency via audit logs and address if high. - **Poor UX causing confusion:** If the UI doesn't clearly indicate why something is blocked, users might get frustrated or blame the system for being "broken". We must design the UX to communicate clearly (e.g. disabled button with info tooltip "Disabled until invoice is paid"). Transparency in UI prevents users from feeling the system is capricious. - **Scalability issues if usage grows:** Sometimes systems built for one team crumble when opened to many. E.g., performance might drop if many projects and logs accumulate. We should foresee some of that: use proper database indexing, perhaps paginate large tables, archive old projects, etc. Early identification of potential bottlenecks (like the audit log growing huge) allows planning (maybe use an Elasticsearch for logs later, but not needed at start). - **Security negligence:** As more clients use it, data protection is key. We should not postpone basic security in MVP (like proper authentication, authorization checks as discussed, encryption of sensitive data). A breach or data leak

(like one client seeing another's files) would break trust. We already plan strict isolation; we must implement it carefully. Also ensure using HTTPS, secure file storage, etc., from the beginning.

Avoiding "Another ClickUp" Syndrome: ClickUp is known for trying to do everything (tasks, docs, goals, mind maps) which can lead to complexity and lack of focus. Our system should remain **focused on creative operation rules**: - We continuously ask, "Does this feature enforce or facilitate a business rule or critical workflow for creative studios?" If not, maybe it doesn't belong. For instance, a document collaboration feature might be nice, but does it contribute to enforcement? Not really – so probably skip or integrate with existing tools instead of building. - Simplicity in UI: ClickUp can overwhelm users with options. We will keep the interface streamlined – showing users only what they need at that step. Each user role will have a simplified view relevant to them (clients see deliverables, approvals; designers see tasks assigned and next steps; ops sees project overviews and statuses). We avoid burying them in settings or menus. - Customization vs Convention: ClickUp offers a million ways to customize statuses and fields. That's power-user friendly but can create chaos. Our approach is more convention-over-configuration – provide a standard way that works out of the box. That's part of our product philosophy (it enforces best practices rather than letting every team reinvent the wheel). While we allow some configuration (especially as SaaS for different agencies), we will be opinionated in core flows. This makes the software *opinionated software* which, if the opinions align with industry best practices, is a strength. Agencies will either align with it or not use it – and that's okay. We'd rather deeply satisfy those who want structure than try to please everyone and become a generic tool. - **UI clutter:** We will adhere to the design system to ensure a clean, professional look. Resist adding excessive decorative elements or experimental UI components that don't serve a purpose. Every element should have a reason (this echoes the clarity principle in Nielsen's heuristics – no irrelevant info). - **Continuous Improvement:** We plan to gather feedback from the NOCO team's use and later beta clients. If some rules are too strict or too lenient, adjust them systematically. We won't chase every feature request that is unrelated to our core value (for example, someone asks "Can it also do invoicing or CRM?" – we might integrate with existing tools via API rather than building an invoice generator if that's outside our scope of enforcement logic). - We must guard against scope creep: it's easy for software to keep expanding its feature set (the ClickUp effect). Instead, we define our boundaries: *This is a Creative Operations Enforcement system*. It's not a full PM, not a chat tool, not a design tool – it interfaces with those perhaps (like linking to Google Drive or Figma for design files, etc.), but its unique selling point is the rule engine and workflow control.

Maintaining Correctness Over Comfort: We design all decisions (both technical and UX) around correctness and enforcing the rules, even if sometimes it may inconvenience a user in the short term. For example, forcing a client to sign a waiver digitally before booking might add a step, but it's critical for legal safety – so we do it and explain why rather than skipping it to reduce clicks. As long as we communicate the rationale (in onboarding or tooltips), users will understand it's for everyone's benefit. We avoid the temptation to add "soft bypasses" just to reduce complaints. If the rule is correct, we uphold it and find a way to make the user experience around it as smooth as possible (like making payment easy rather than allowing skipping payment).

With this comprehensive blueprint, we have defined how the system works under the hood (state machines, rule engine), how data is structured around enforcing rules, how the UI is designed to reflect state and authority, and how to implement it incrementally. The focus remains sharply on **operational enforcement** – making sure the software actively guides and controls the process according to business rules, rather than being a passive tracker. This blueprint is build-ready: each phase can be translated into development tasks, from setting up state models and database schemas to coding the rule checks and styling the frontend. By following this plan, we ensure the end product is not another

generic SaaS, but a specialized tool that embodies “process as policy,” delivering immediate value to creative agencies by protecting their time, scope, and revenue through software.

---

1 From MVP to Full-Scale SaaS Platform: Lessons Learned | by Jbarnesora | Medium
https://medium.com/@jbarnesora536/from-mvp-to-full-scale-saas-platform-lessons-learned-f9c28c7d610c

2 Understanding State Machines: A Developer’s Guide to Predictable Application Logic | by Melek Charradi | Medium
https://medium.com/@melekcharradi/understanding-state-machines-a-developers-guide-to-predictable-application-logic-d3df50e3e621

3 4 5 11 State-machine Rules | YouTrack Server Documentation
https://www.jetbrains.com/help/youtrack/server/state-machine-per-issue-type.html

6 Audit Log: Definition and Guide
https://chronosphere.io/learn/audit-log-definition-guide/

7 Death to Three Options and Two Rounds of Revisions | by JD Graffam | Medium
https://medium.com/@jdgraffam/death-to-three-options-and-two-rounds-of-revisions-9e1261f29934

8 9 I'm refusing to hand over project files until I'm paid, am I doing the right thing? : r/vfx
https://www.reddit.com/r/vfx/comments/jiejy6/im_refusing_to_hand_over_project_files_until_im/

10 Retainer Agreements & Contracts Demystified | Runn
https://www.runn.io/blog/retainer-agreements-demystified

12 Role-Based Access Control vs. Attribute-Based Access Control
https://www.immuta.com/blog/attribute-based-access-control/