Part 2. Architectural Decision Records (ADRs) Documenting Each Selected Tactics/Patterns

I. Availability (A)

ADR 1: Circuit Breaker Pattern (A.1)

- **Title:** Implement Circuit Breaker Pattern for External Payment Service Resilience
- **Status: Accepted**
- **Context (problem & background):** During Flash Sales (Peak Load/Overloaded Mode), the external Payment Service may fail or time out. Unbounded retries against a failed service can cause cascading failures, degrading system availability and performance for all users. The goal is to stop attempting immediate payment upon detecting repeated external faults.
- **Decision:** Implement the **Circuit Breaker pattern** on the External Payment Service Connector. This mechanism will detect service failure and prevent further resource-wasting requests for a defined period.
- **Consequences (pros/cons, trade-offs):**
    - **Pros:** Prevents cascading failures that could take the system out of service. It removes the policy about how many retries to allow from individual components. It quickly identifies and isolates persistent external faults, supporting the availability goal.
    - **Cons:** Requires careful selection of timeout or retry values; if too short, it may trip unnecessarily (a "false positive"), lowering availability and performance. Introduces an intermediary component that must be maintained.
- **Alternatives:**
    - *Unlimited Retries:* Rejected, as this would make the invoking component useless and cause failure to cascade across the whole system during peak load.
    - *Immediately Fail:* Rejected, as this reduces availability by not allowing for transient network issues to resolve, forcing immediate user failure.

ADR 2: Graceful Degradation Tactic (A.1)

- **Title:** Utilize Graceful Degradation for Order Processing during Peak Load
- **Status: Accepted**
- **Context (problem & background):** When the system experiences Flash Sale Peak Load, the Order Submission Endpoint/API handles up to 1,000 requests per second. If the Payment Service fails (as handled by the Circuit Breaker), the system must continue to accept orders to maintain service availability, even if full processing cannot complete immediately. The system must maintain critical functions while dropping less critical ones.

- **Decision:** Implement **Graceful Degradation** by routing orders to a queue for asynchronous processing upon detection of a payment fault, so that 99% of order requests may be successfully accepted (queued).
- **Consequences (pros/cons, trade-offs):**
    - **Pros:** Maintains the most critical system functions (order acceptance) in the presence of component failures, preventing a complete system crash. It allows the system to operate in a degraded but useful mode.
    - **Cons:** Orders accepted into the queue may experience higher latency before final confirmation, impacting the performance quality attribute. Requires additional architecture elements (queues/workers).
- **Alternatives:**
    - *Hard Stop/Fail All:* Rejected, as this directly violates the availability requirement during component failures.
    - *Non-Graceful Shutdown:* Rejected, as this means component failures cause a complete system failure rather than a controlled reduction of functionality.

ADR 3: Rollback Tactic (A.2)

- **Title:** Employ Rollback Tactic for Transaction Integrity during Transient Failures
- **Status: Accepted**
- **Context (problem & background):** The core use case involves an atomic transaction block: payment processing followed by sale persistence and stock decrement. If a transient failure occurs during this critical sequence, such as a database conflict or communication timeout, any partial state changes (e.g., partial log entries, resource reservations) must be undone immediately to maintain data consistency.
- **Decision:** Implement the **Rollback** tactic, ensuring that the system reverts to a previous known good state ("rollback line") upon fault detection. Leveraging PostgreSQL's ACID compliance, the entire transaction is rolled back if the payment or persistence step fails.
- **Consequences (pros/cons, trade-offs):**
    - **Pros:** Ensures atomicity and consistency, preventing data corruption and unintended side effects, achieving **zero observed data corruption**. It prepares the system for a safe retry attempt.
    - **Cons:** Depends on a checkpoint (previous good state) being available. If the checkpoints are complex or large, this mechanism can consume arbitrarily large amounts of memory, potentially affecting performance.
- **Alternatives:**
    - *Forward Error Recovery:* Rejected, as this tactic relies on data redundancy or built-in correction to move *forward* to a desirable state. Rollback is more appropriate for guaranteeing transaction integrity in external interactions where the outcome is uncertain.

ADR 4: Retry Tactic (A.2)

- **Title:** Utilize Retry Tactic for Transient Payment Fault Recovery
- **Status: Accepted**
- **Context (problem & background):** Transient failures (e.g., temporary card processing failure or communication timeout) are common in distributed systems. The system must attempt to mask these failures to the user and successfully complete the transaction.
- **Decision:** Implement the **Retry** tactic for payment and full transaction logic (up to N=3 attempts). This assumes the fault is transient and retrying the operation may succeed.
- **Consequences (pros/cons, trade-offs):**
    - **Pros:** Masks transient faults, increasing availability and improving user experience by successfully completing 99% of transactions that initially fail due to temporary errors.
    - **Cons:** A limit must be placed on retries; otherwise, if the fault is permanent, retrying will add unnecessary latency (negatively impacting performance) and waste resources.
- **Alternatives:**
    - *Ignore Faulty Behavior:* Rejected, as ignoring payment failures results in lost revenue and/or transaction errors, violating functional requirements.
    - *Immediate Failure/Error:* Rejected, as this forces the user to manually re-initiate the transaction, lowering availability/usability for transient issues.

ADR 5: Removal from Service Tactic (A.3)

- **Title:** Implement Removal from Service for Predictive Fault Mitigation in Queue Workers
- **Status: Accepted**
- **Context (problem & background):** Components, such as Order Processing Queue Workers, may develop latent faults (e.g., memory leaks) over time, which are predictive of future crashes, potentially leading to service-affecting failures during degraded operation.
- **Decision:** The system implements **Removal from Service** (software rejuvenation). If an internal monitor detects high memory utilization (predictive fault), the unhealthy worker is automatically taken offline and reset, and a replacement instance is provisioned.
- **Consequences (pros/cons, trade-offs):**
    - **Pros:** Preempts potential system failures caused by accumulating latent faults, ensuring system availability. The automated restart aims for a Mean Time to Repair (MTTR) of less than 60 seconds.
    - **Cons:** Requires external monitoring infrastructure to detect the predictive fault. The brief interruption when the component is removed and replaced may slightly reduce overall throughput (performance) during that period.

- **Alternatives:**
    - *Allow Worker to Crash:* Rejected, as an unmanaged crash is less predictable and takes longer to recover from, increasing MTTR.
    - *Escalating Restart:* Rejected, as removal from service (a complete clean restart) is simpler to manage for stateless queue workers than varying granularities of restart.

II. Security (S)

ADR 6: Authenticate Actors Tactic (S.1)

- **Title:** Implement Authenticate Actors for Partner Catalog Ingest API Security
- **Status: Accepted**
- **Context (problem & background):** External Partners (VARs) interact with the system via the Catalog Ingest API Endpoint. It is critical to protect the system and data from unauthorized access attempts originating externally.
- **Decision:** The system uses the **Authenticate Actors** tactic, requiring a valid, non-expired API key for all catalog ingest requests. If authentication fails, the request is immediately denied, and the attempt is logged (Audit).
- **Consequences (pros/cons, trade-offs):**
    - **Pros:** Ensures confidentiality and integrity by guaranteeing that 100% of unauthorized external attempts are denied access. It provides a strong perimeter defense.
    - **Cons:** Adds overhead to every request (performance cost). Requires a management layer to securely issue, store, and revoke API keys.
- **Alternatives:**
    - *Restrict Access to IP Only:* Rejected, as relying solely on network location is insufficient protection if an attacker compromises a partner's network.
    - *Authorization Only (Post-Authentication):* Rejected, as this would waste resources by processing invalid requests up to the point of authorization, increasing computational overhead.

ADR 7: Validate Input Tactic (S.2)

- **Title:** Implement Validate Input Tactic to Prevent SQL Injection from Partner Feeds
- **Status: Accepted**
- **Context (problem & background):** Incoming external partner feeds (CSV/JSON) are a source of input data. If these data fields contain malicious payloads, such as a known **SQL Injection** payload, the system's integrity is threatened upon persistence.

- **Decision:** Implement the **Validate Input** tactic. All incoming external data fields undergo sanitization, filtering, and canonicalization before they are processed or persisted to the PostgreSQL database.
- **Consequences (pros/cons, trade-offs):**
    - **Pros:** Protects data integrity by ensuring zero malicious payloads successfully reach the database, measured by 100% adherence to database constraints. This is a defense against attacks aiming to change data.
    - **Cons:** Adds computational overhead (performance cost) to the ingestion process, as every external data field must be validated. If the validation logic is complex, it introduces potential bugs (modifiability risk).
- **Alternatives:**
    - *Relying on ORM Parameterization Alone:* Rejected, although ORMs help, custom data fields might still pose issues, and explicit filtering/sanitization is required for a good level of security assurance.
    - *Delayed Detection (Audit/Post-Mortem):* Rejected, as the goal is prevention; detecting corruption after it occurs is too late.

III. Modifiability (M)

ADR 8: Use an Intermediary/Encapsulate Tactic (M.1)

- **Title:** Use Intermediary/Encapsulation for Partner Format Modifiability
- **Status: Accepted**
- **Context (problem & background):** The system must accommodate new partner integration formats (e.g., XML) with low effort, preventing modifications to existing core domain logic or parser modules. This relates to increasing cohesion and reducing coupling.
- **Decision:** Implement the **Use an Intermediary / Encapsulate** tactic in the Partner Catalog Ingest module. This design uses encapsulation (such as introducing an explicit interface) and an intermediary (such as the Adapter pattern) to hide the implementation details of format parsing from the core business logic.
- **Consequences (pros/cons, trade-offs):**
    - **Pros:** Reduces coupling, allowing the new XML format parser to be added without modifying existing code. Functional responsibilities are localized, making changes local and reducing effort to under 20 person-hours.
    - **Cons:** Introducing an intermediary adds a small amount of abstraction and indirection, increasing system complexity initially.
- **Alternatives:**
    - *Direct Integration:* Rejected, as this would require modifying core domain logic for every new format, resulting in high coupling and high maintenance cost.
    - *Module Splitting:* Rejected, as simply splitting modules (Increase Cohesion) wouldn't resolve the coupling issue between the core logic and the various parsers; an intermediary is needed to mediate.

ADR 9: Adapter Pattern (M.1 Context)

- **Title:** Adopt Adapter Pattern to Support Varied Partner Catalog Formats
- **Status: Accepted**
- **Context (problem & background):** Following the decision to use an Intermediary for Modifiability (ADR 8), a specific mechanism is needed to handle the syntactic and data semantic distance between external partner formats (CSV/JSON/XML) and the internal data model.
- **Decision:** Implement the **Adapter pattern** as the intermediary mechanism. An Adapter component will translate the external data format into the internal format expected by the core Order Processing Logic, without modifying the existing business logic.
- **Consequences (pros/cons, trade-offs):**
    - **Pros:** Hides complexity of external formats from internal components. Allows easy addition of new formats (e.g., XML) by simply implementing a new adapter, achieving the modifiability requirement. All three Adapter-like patterns (wrapper, bridge, adapter) allow access to an element without forcing a change to the element.
    - **Cons:** Requires up-front development work. Adds computational overhead to transform data.
- **Alternatives:**
    - *Manual Code Change per Format:* Rejected; high cost and coupling risk.
    - *Bridge Pattern:* Rejected; while Bridge is useful for bridging an abstraction and its (potentially numerous) implementations, Adapter and Bridge are often used at different points in the software lifecycle - Bridge makes them work before they are; the Adapter pattern makes things work after they're designed which is needed in this case. The Adapter pattern is specifically suited for reconciling incompatible interfaces or protocols.

ADR 10: Feature Toggle Mechanism (M.2)

- **Title:** Implement Feature Toggle Mechanism for Instant Feature Disablement
- **Status: Accepted**
- **Context (problem & background):** New features, like the "Flash Sale" pricing logic, may contain unexpected bugs post-deployment. The Product Owner needs the ability to instantly disable this feature at runtime without requiring a zero-downtime hotfix or complete service redeployment.
- **Decision:** Implement a **Feature Toggle** mechanism (realizing the **Defer Binding** tactic via external configuration). This mechanism allows the "Flash Sale" logic to be enabled or disabled instantly via an external configuration flag.
- **Consequences (pros/cons, trade-offs):**
    - **Pros:** Provides a "kill switch". Allows instant feature disablement across all users within 5 seconds, requiring zero code changes or redeployment, fulfilling the modifiability goal. It is useful for testing in production (scaled rollouts).

    - **Cons:** Adds complexity by requiring external configuration management infrastructure. Requires developer to ensure that the code path always respects the toggle flag, leading to slightly more complex logic.
- **Alternatives:**
    - *Rollback:* Rejected, as rollback involves complex orchestration across multiple services and databases and takes considerable time (hours or minutes), violating the requirement for instant disablement.
    - *Hard-coded Configuration:* Rejected, as this requires a full restart or redeployment to change the setting.

IV. Performance (P)

ADR 11: Manage Event Arrival Tactic (P.1)

- **Title:** Implement Manage Event Arrival (Throttling/Queuing) for Flash Sale Load
- **Status: Accepted**
- **Context (problem & background):** The system expects an event stream of 1,000 order placement requests per second during Flash Sale Peak Load. This sudden, high demand risks overloading the Order Submission Endpoint, leading to unpredictable, unbounded latency.
- **Decision:** Implement the **Manage Event Arrival** tactic using Throttling and Queuing mechanisms. This limits concurrent processing to a sustainable rate, prioritizing throughput and ensuring the average latency for 95% of requests remains below 500 milliseconds.
- **Consequences (pros/cons, trade-offs):**
    - **Pros:** Ensures predictable processing for accepted events and bounds overall latency, preventing system collapse during overload. It protects downstream resources, like the database, from being overwhelmed.
    - **Cons:** If events arrive too rapidly, non-processed events must be queued or discarded, potentially requiring a complex policy on how to handle queue overflow. Queuing adds latency to the overall response time compared to immediate processing.
- **Alternatives:**
    - *Increase Resources:* Rejected, as vertically or horizontally scaling hardware (Increase Resources tactic) may not keep pace with massive spikes and is generally more expensive than optimized processing strategies.
    - *Limit Event Response:* Rejected, as simply limiting response might result in discarding events, which is typically unacceptable for financial transactions unless queues are bounded.

ADR 12: Introduce Concurrency Tactic (P.2)

- **Title:** Utilize Introduce Concurrency (Transaction Locking) for Stock Update Efficiency
- **Status: Accepted**
- **Context (problem & background):** During high traffic, multiple concurrent internal processes attempt to modify the stock level for the same product. This resource contention for shared data must be managed to ensure data integrity and minimize the blocked time, which affects performance.
- **Decision:** Implement the **Introduce Concurrency** tactic by utilizing the database transaction locking/isolation levels provided by PostgreSQL. This ensures efficient shared resource access while keeping database lock wait time below 50 milliseconds.
- **Consequences (pros/cons, trade-offs):**
    - **Pros:** Reduces blocked time by allowing efficient processing in parallel. It maintains data integrity (consistency/isolation) during simultaneous stock updates.
    - **Cons:** If lock granularity is too coarse, it can introduce resource contention, leading to bottlenecks. Improper concurrency control can lead to deadlock issues.
- **Alternatives:**
    - *Maintain Multiple Copies of Data (Replication):* Rejected for critical stock updates, as managing data consistency (synchronization) across replicated mutable data stores is complex and risks temporal inconsistency.
    - *Static Scheduling:* Rejected, as dynamic resource allocation is required due to unpredictable, high-volume concurrent access patterns.

V. Integrability (I)

ADR 13: Tailor Interface Tactic (I.1)

- **Title:** Use Tailor Interface Tactic for External Reseller API Onboarding
- **Status: Accepted**
- **Context (problem & background):** Onboarding external reseller APIs (e.g., legacy SOAP/XML) requires connecting systems that use different data semantic protocols and interfaces. These syntactic and semantic differences create distance that must be resolved without modifying the external API or the internal order service.
- **Decision:** Implement the **Tailor Interface** tactic to add translation and buffering capabilities to the interface of the New Reseller API Connector. This allows the external API to interoperate with the internal system seamlessly.
- **Consequences (pros/cons, trade-offs):**
    - **Pros:** Resolves syntactic and data semantic distance during integration. Reduces the effort required for integration to under 40 person-hours. The tailoring is confined to the connector/wrapper, protecting the core system.

    - **Cons:** Typically applied during integration, requiring explicit design support in the architecture. The wrapper component must be maintained.
- **Alternatives:**
    - *Adhere to Standards:* Rejected, as legacy reseller APIs cannot be forced to adhere to internal standards.
    - *Configure Behavior:* Rejected, as configuration is less suited for complex data or protocol translations compared to a dedicated interface modification.

ADR 14: Adapter Pattern (I.1 Context)

- **Title:** Use Adapter Pattern to Resolve External Protocol/Data Mismatches
- **Status: Accepted**
- **Context (problem & background):** As a means to implement the Tailor Interface tactic (ADR 13) for the reseller API onboarding, a specific pattern is needed to translate incompatible external protocols (e.g., SOAP/XML) and internal data formats.
- **Decision:** Implement the **Adapter pattern** to translate data formats and protocol sequences between the external system and the internal order service. The Adapter acts as the dedicated translation layer.
- **Consequences (pros/cons, trade-offs):**
    - **Pros:** Allows the Tailor Interface tactic to be executed efficiently. Provides access to an element without forcing a change to the element or its interface. Makes integration effort predictable.
    - **Cons:** Requires up-front development work. Adds a small latency overhead due to the translation layer.
- **Alternatives:**
    - *Wrapper:* Rejected, as the Adapter is preferred for connecting existing incompatible interfaces.
    - *Bridge Pattern:* Rejected, as the bridge pattern is used to decouple an abstraction from its implementation, whereas the Adapter focuses on interface matching.

ADR 15: Use an Intermediary Tactic (I.2)

- **Title:** Employ Use an Intermediary Tactic for Decoupled Reporting Services
- **Status: Accepted**
- **Context (problem & background):** A new internal reporting service needs to consume incoming Partner Catalog data feed. Directly coupling the new reporting service to the existing Partner Catalog Ingest Module would create dependencies, raising the cost of future modification or introduction of new consumers.
- **Decision:** Implement the **Use an Intermediary** tactic, specifically using a message bus or broker to facilitate the communication. The Partner Catalog Ingest System broadcasts data updates (Publish-Subscribe pattern) rather than knowing about or invoking specific consumers.

- **Consequences (pros/cons, trade-offs):**
    - **Pros:** Reduces dependencies between data producers and consumers, minimizing coupling. Adding the new reporting consumer requires modification of zero lines of code in the existing Ingest module, satisfying the integrability response measure.
    - **Cons:** Introducing an intermediary adds a performance cost (latency) and infrastructure complexity. System performance and resource management become harder to reason about due to implicit invocation.
- **Alternatives:**
    - *Shared Data Repository:* Rejected, although a shared data repository is an intermediary, coupling still exists through the schema and timing dependencies (temporal distance).
    - *Direct Invocation:* Rejected, high coupling and high cost for future changes.

ADR 16: Publish-Subscribe Pattern (I.2)

- **Title:** Adopt Publish-Subscribe Pattern for Asynchronous Data Broadcast
- **Status: Accepted**
- **Context (problem & background):** Following the decision to use an Intermediary (ADR 15), a specific pattern is required for asynchronous distribution of Partner Catalog data updates to multiple unknown internal consumers (like the new reporting service).
- **Decision:** Implement the **Publish-Subscribe pattern**. The Ingest Module acts as a publisher, communicating via asynchronous messages (events or topics). Consumers subscribe to these messages, remaining unaware of the publisher's identity or existence.
- **Consequences (pros/cons, trade-offs):**
    - **Pros:** Promotes extreme decoupling (publisher has no knowledge of subscribers). Increases integrability and modifiability by making it easy to add new consumers with zero code changes to the source module. Events can be easily logged for audit purposes.
    - **Cons:** May negatively impact performance (latency) due to asynchronous communication; mitigating this requires a distributed coordination mechanism. It is difficult for a component to be sure when a message will be received.
- **Alternative:**
    - *Client-Server:* Rejected, as the Client (consumer) would need a priori knowledge of the Server (publisher), leading to high coupling.

VI. Testability (T)

ADR 17: Record/Playback Tactic (T.1)

- **Title:** Implement Record/Playback Tactic for Flash Sale Load Simulation Reproducibility
- **Status: Accepted**
- **Context (problem & background):** Load tests simulating Flash Sale traffic (500 transactions) often produce faults or performance degradation that are difficult to re-create manually. The testing infrastructure must efficiently reproduce the exact workload condition (including system state and input data).
- **Decision:** Implement the **Record/Playback** tactic. This involves capturing the state when data crosses an interface (recording) and using that captured state to re-create the fault condition (playback).
- **Consequences (pros/cons, trade-offs):**
    - **Pros:** Saves substantial time in bug reproduction, reducing the effort required to replicate the workload condition to feasibly less than 1 hour. Makes tests repeatable. Enables analysis of the exact state that caused degradation.
    - **Cons:** Requires additional infrastructure and instrumentation to capture the state crossing the interface. Recording and capturing state adds a small performance overhead during the test itself.
- **Alternatives:**
    - *Executable Assertions:* Rejected, as assertions detect *when* a program is faulty but do not inherently provide the ability to replay the complex external inputs that led to that state.

ADR 18: Dependency Injection Pattern (T.2)

- **Title:** Adopt Dependency Injection Pattern for Isolating Payment Service Dependencies
- **Status: Accepted**
- **Context (problem & background):** The developer needs to verify the logic of the Order Processor when the external Payment Service returns various failure modes (e.g., transient timeouts). Testing this logic against the real external service is slow, unreliable, and requires complex setup.
- **Decision:** Implement the **Dependency Injection (DI) pattern**. This pattern separates the client's dependencies from its behavior, allowing a mock object simulating a transient failure to be substituted (injected) in place of the real Payment Service during testing.
- **Consequences (pros/cons, trade-offs):**
    - **Pros:** Greatly enhances controllability and observability during development. Allows the complex retry/rollback logic to be executed and validated quickly (under 5 seconds) without external network dependencies. Promotes high cohesion by writing client code with no knowledge of how it is tested.
    - **Cons:** Adds a small amount of up-front complexity and may require developer to retrain to use the inversion of control paradigm. May make runtime performance slightly less predictable.
- **Alternatives:**

    - *Specialized Interfaces:* Rejected, although effective, Dependency Injection is a more standardized and systematic way to manage and isolate dependencies for unit testing.
    - *Stubbing/Hardcoding:* Rejected, as DI offers a clean, architectural approach to dependency management rather than ad-hoc stubbing within the code.

VII. Usability (U)

ADR 19: Minimize Impact of User Errors Tactic (U.1)

- **Title:** Employ Minimize Impact of User Errors for Declined Payment Feedback
- **Status: Accepted**
- **Context (problem & background):** When the simulated Payment Service declines a transaction, this is an instance of a user error (U.1 Stimulus: Minimize impact of errors) that can result in user frustration and abandonment. The system must ensure the user can successfully complete a modified transaction quickly.
- **Decision:** Implement the **Minimize Impact of User Errors** tactic. The User Interface provides clear, immediate, actionable error feedback that suggests an alternative payment method, enabling the user to recover in less than 90 seconds.
- **Consequences (pros/cons, trade-offs):**
    - **Pros:** Increases user satisfaction and task completion rates by guiding the user through recovery from errors. It uses the system's ability to cancel a command issued incorrectly.
    - **Cons:** Requires careful design of the user interface flow to handle recovery gracefully. Implementing complex error logging (Audit) adds logging overhead (performance cost).
- **Alternatives:**
    - *Silent Failure/Generic Error:* Rejected, as this fails to provide actionable feedback and leads to high user dissatisfaction.

ADR 20: Maintain System Model Tactic (U.2)

- **Title:** Use Maintain System Model Tactic (Progress Indicator) for Long-Running Tasks
- **Status: Accepted**
- **Context (problem & background):** Under Peak Load, when a user clicks "Confirm Order," the subsequent sequence of stock checks and database transactions can be lengthy. Lack of immediate feedback decreases user confidence and satisfaction.
- **Decision:** Implement the **Maintain System Model** tactic by displaying a Progress Indicator (e.g., a wait screen or progress bar). This explicit model of the system's state predicts the time needed to complete the activity and provides immediate feedback.
- **Consequences (pros/cons, trade-offs):**

    - **Pros:** Increases confidence and satisfaction by giving immediate, appropriate feedback during long-running tasks. It helps users understand why they are waiting, helping to keep satisfaction scores above 80%.
    - **Cons:** Requires the system to accurately predict completion time; inaccurate predictions can undermine user confidence. Maintaining and updating the system model adds a small, ongoing computation cost (performance trade-off).
- **Alternatives:**
    - *Pause/Resume Tactic:* Rejected, as this tactic is typically reserved for highly resource-intensive or interruptible operations (like large file downloads). Order confirmation, due to its transactional nature, is better suited to a progress indicator.
    - *No Feedback:* Rejected, this directly violates the usability goal and can decrease satisfaction.