drop-transaction-failure-spec

Drop Transaction Failure Handling & Recovery

Task: MC #1191 **Created:** 2026-02-17 **Author:** John (Software Architect Agent) **Status:** DRAFT — Awaiting Alem approval

---

Executive Summary

This specification defines comprehensive transaction failure handling for Drop's fintech payment system. Drop operates as a PSD2 PISP (Payment Initiation Service Provider) — we initiate payments from users' bank accounts but never hold customer money. This creates unique challenges:

- **External dependency:** Every transaction depends on user's bank and Open Banking provider
- **Asynchronous flow:** PISP initiation → bank processing → status callback (can take seconds to days)
- **Failure modes:** Network timeouts, bank declines, partial processing, provider outages
- **Customer impact:** Real money, real trust — failures must be handled gracefully

Core principles:

1. **Clear state machine** — No ambiguous states
2. **Idempotency** — Network retries never cause double-charges
3. **Automatic retry** — Transient failures self-heal
4. **User communication** — Always tell user what's happening
5. **Admin tooling** — Manual intervention when automation can't resolve

---

1. Current State Analysis

1.1 What We Have (Good)

Idempotency keys:

- Both `/api/transactions/remittance/route.ts` and `/api/transactions/qr-payment/route.ts` accept `idempotencyKey`
- Check for existing transaction: `SELECT ... WHERE idempotency_key = ? AND user_id = ?`
- Returns cached response for duplicate requests (prevents double-charge)
- **Status: Production-ready**

Basic error handling:

- `insufficient_balance` error caught and returned as 402
- Rate limiting: IP (10/min) + user (3/min)
- Transaction wrapped in DB transaction (atomic balance check + insert)
- **Status: Good foundation**

30-second timeout:

- PISP API calls have `AbortController` with 30s timeout
- Returns specific timeout error: "Payment request timeout"
- **Status: Implemented**

1.2 What's Missing (Critical Gaps)

 State machine enforcement:

- `transactions.status` has CHECK constraint: `'processing','completed','failed'`
- But no state transition validation (can jump from processing → completed without rules)
- No transition audit (who/when/why status changed)

 Retry logic:

- Timeout errors return failure immediately — no retry
- No exponential backoff
- No max retry counter
- No dead letter queue for permanently failed transactions

 Background reconciliation:

- Transactions stuck in `processing` status stay there forever
- No periodic job to check PISP provider for status updates
- No admin alert when transactions are stuck

 Partial failure handling:

- FX conversion success + transfer failure → no rollback/refund flow
- No compensation logic for partial state

User communication:

- No transaction status page showing real-time progress
- No push notification on status change
- No email on final completion/failure
- Error messages are generic (not user-friendly)

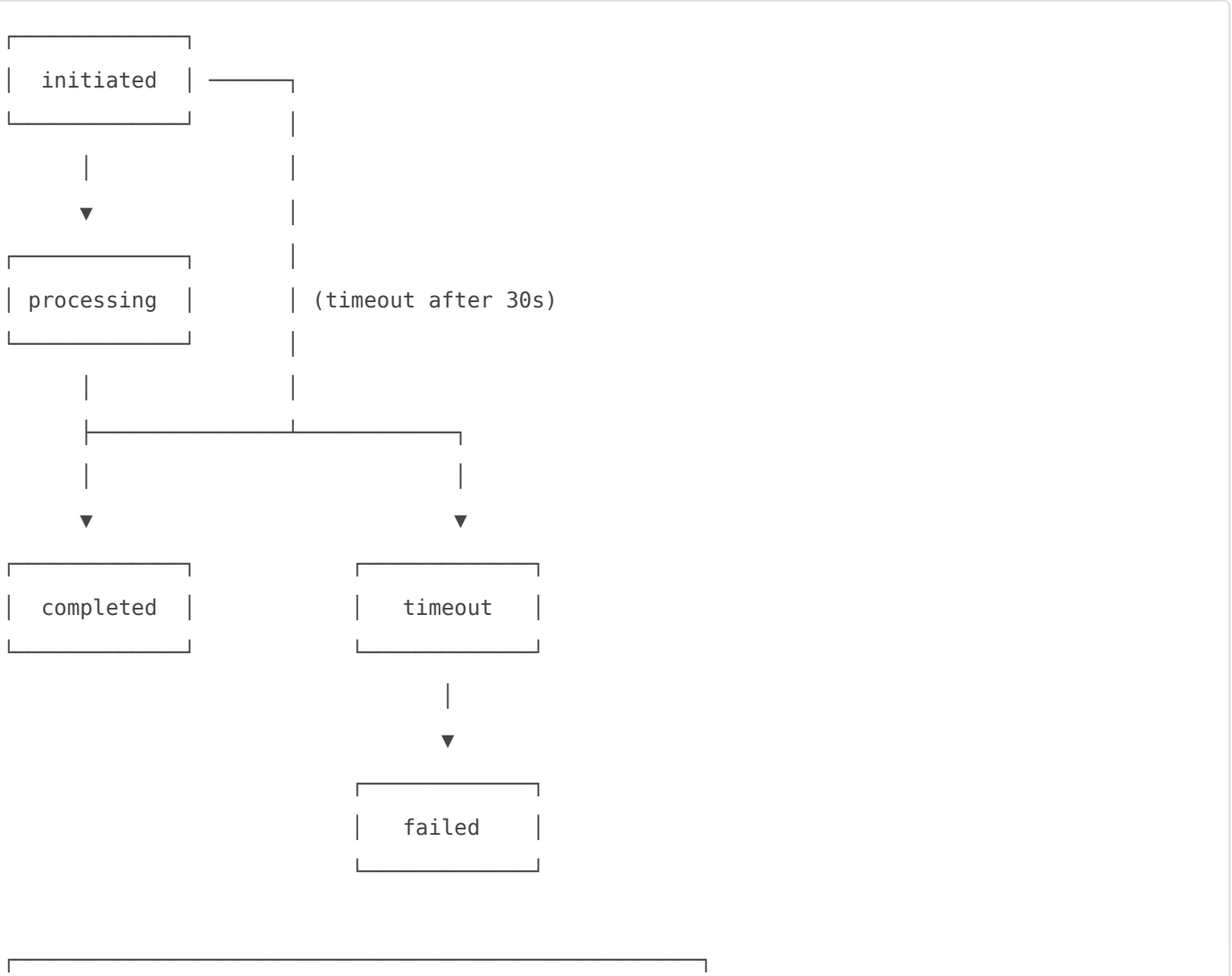
Admin tools:

- No `/api/admin/transactions/stuck` endpoint to list limbo transactions
- No manual retry mechanism
- No manual resolution workflow

---

2. Transaction State Machine

2.1 States



```
│            partially_completed                          │ (future — FX success, transfer fail)
└─────────────────────────────────────────────────────────┘
```

2.2 State Definitions

State	Meaning	Terminal?	User-Facing Message
`initiated`	API request received, validation passed, DB record created	No	"Initiating payment..."
`processing`	PISP provider accepted request, waiting for bank confirmation	No	"Your payment is being processed"
`timeout`	PISP provider didn't respond within 30s, will check status later	No	"Processing your payment — we'll notify you when complete"
`completed`	Bank confirmed payment successful	Yes	"Payment completed"
`failed`	Bank declined, or PISP returned permanent error	Yes	"Payment failed: [reason]"
`partially_completed`	FX conversion succeeded but transfer failed (future)	No	"Processing refund..."

Terminal states: `completed`, `failed` — no further transitions allowed

2.3 Valid Transitions

```
const VALID_TRANSITIONS = {
  initiated: ["processing", "failed"],
  processing: ["completed", "timeout", "failed"],
  timeout: ["completed", "failed", "processing"], // retry
  partially_completed: ["completed", "failed"], // after refund
  completed: [], // terminal
  failed: [], // terminal
};
```

Enforcement: Database CHECK constraint + application-level validation

2.4 Transition Audit

Every status change logged in `audit_log`:

```
INSERT INTO audit_log (
  id, user_id, action, resource_type, resource_id,
  details, ip_address, user_agent, request_id
) VALUES (
  'aud_xyz', 'usr_abc', 'TRANSACTION_STATUS_CHANGE',
  'transaction', 'tx_rem_123',
  '{"from": "processing", "to": "completed", "reason": "PISP callback", "external_id":
"ext_456"}',
  '10.0.1.5', 'Drop-iOS/1.0', 'req_789'
);
```

Compliance: PSD2 requires 5-year audit trail of all payment operations

---

3. Idempotency

3.1 Current Implementation (Keep It)

 Already production-ready:

```
// Check for existing transaction with this idempotency key (scoped to user)
const existing = await getOne<ExistingTx>(
  "SELECT id, type, status, amount, currency, fee, ...
   FROM transactions
   WHERE idempotency_key = ? AND user_id = ?",
  [idempotencyKey, u.id]
);

if (existing) {
  // Return cached response (same payload as successful creation)
  return NextResponse.json({ data: existing }, { status: 200 });
}
```

Key features:

- Scoped to user (prevents IDOR)
- Returns exact same response (status 200, not 201)
- No expiry — idempotency keys valid forever
- Client must generate UUID or similar unique key

3.2 Best Practices

Client implementation:

```
// Generate idempotency key client-side
const idempotencyKey = `${userId}_${Date.now()}_${crypto.randomUUID()}`;

// Send with every payment request
await fetch('/api/transactions/remittance', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
    recipientId: 'rec_123',
    amount: 500,
    idempotencyKey, // ← REQUIRED
  })
});
```

No changes needed — current implementation is correct

---

4. Retry Logic

4.1 Classification of Errors

Error	Type	Retry?	Example
Network timeout	Transient	✅ Yes	`AbortError`, socket timeout
PISP 5xx	Transient	✅ Yes	500 Internal Server Error, 503 Service Unavailable
PISP 4xx client error	Permanent	❌ No	400 Bad Request, 401 Unauthorized
Bank decline	Permanent	❌ No	Insufficient funds (from bank), invalid IBAN
Validation error	Permanent	❌ No	Amount < minimum, KYC not approved

Rule: Only retry errors that are transient (temporary network/server issues)

4.2 Exponential Backoff Strategy

Max retries: 3 attempts **Delays:** 2s → 8s → 32s (exponential) **Jitter:** ±20% to avoid thundering herd

```
const RETRY_CONFIG = {
  maxRetries: 3,
  baseDelayMs: 2000, // 2 seconds
  maxDelayMs: 60000, // 1 minute cap
  jitterPercent: 0.2, // ±20%
};

function calculateDelay(attempt: number): number {
  const exponentialDelay = RETRY_CONFIG.baseDelayMs * Math.pow(4, attempt - 1);
  const cappedDelay = Math.min(exponentialDelay, RETRY_CONFIG.maxDelayMs);
  const jitter = cappedDelay * RETRY_CONFIG.jitterPercent * (Math.random() * 2 - 1);
  return Math.floor(cappedDelay + jitter);
}

// Attempt 1: 2s ± 400ms  = 1.6-2.4s
// Attempt 2: 8s ± 1.6s   = 6.4-9.6s
// Attempt 3: 32s ± 6.4s  = 25.6-38.4s
```

4.3 Retry Implementation

Two approaches:

Option A: In-Process Retry (Simpler, Recommended for MVP)

Retry within the same API request (synchronous):

```
async function callPispWithRetry(
  fn: () => Promise<PaymentResult>,
  txId: string
): Promise<PaymentResult> {
  let lastError: Error | null = null;

  for (let attempt = 1; attempt <= RETRY_CONFIG.maxRetries; attempt++) {
    try {
      const result = await fn();

      // Success — return immediately
      if (result.success) return result;
```

```
    // Permanent error (4xx, bank decline) — don't retry
    if (isPermanentError(result.error)) {
      await logAudit({
        userId: txId,
        action: "PISP_PERMANENT_ERROR",
        resourceType: "transaction",
        resourceId: txId,
        details: { attempt, error: result.error },
      });
      return result;
    }

    // Transient error — prepare to retry
    lastError = new Error(result.error || "Unknown error");

  } catch (error) {
    lastError = error as Error;

    // Non-retryable (validation error, etc.)
    if (!isTransientError(error)) throw error;
  }

  // If not last attempt, wait before retry
  if (attempt < RETRY_CONFIG.maxRetries) {
    const delay = calculateDelay(attempt);
    await logAudit({
      userId: txId,
      action: "PISP_RETRY_SCHEDULED",
      resourceType: "transaction",
      resourceId: txId,
      details: { attempt, nextAttempt: attempt + 1, delayMs: delay },
    });
    await sleep(delay);
  }
}

// All retries exhausted
await logAudit({
  userId: txId,
```

```
    action: "PISP_ALL_RETRIES_FAILED",
    resourceType: "transaction",
    resourceId: txId,
    details: { maxRetries: RETRY_CONFIG.maxRetries, lastError: lastError?.message },
  });

  return {
    success: false,
    status: "failed",
    error: `Payment failed after ${RETRY_CONFIG.maxRetries} attempts`
  };
}
```

Pros:

- Simple — no queue infrastructure needed
- User waits for final result (good UX for fast retries)
- Automatic cleanup (no orphan jobs)

Cons:

- Request can take up to ~40s (blocks thread)
- If server crashes mid-retry, transaction stuck
- No visibility into retry progress

Option B: Background Job Queue (Production-Grade)

Move retries to background worker using job queue:

Tech stack:

- **Job queue:** BullMQ (Redis-backed) or pg-boss (PostgreSQL-backed, no extra infra)
- **Worker:** Separate process polls queue every 5s

Flow:

1. API route creates transaction with status `initiated`
2. Enqueue job: `{ type: "pisp_call", txId: "tx_rem_123", attempt: 1 }`
3. Return to user: `{ status: "processing", txId: "tx_rem_123" }`
4. Worker picks job → calls PISP → updates transaction status
5. On transient failure → re-enqueue with delay + increment attempt
6. On success/permanent failure → mark transaction terminal

Pros:

- Non-blocking (API responds instantly)
- Survives server restarts (jobs persisted in DB)
- Can inspect queue (show pending retries in admin dashboard)

Cons:

- More complex (requires job queue setup)
- More infrastructure (Redis or pg-boss tables)
- User must poll `/api/transactions/[id]` for status updates

Recommendation: Start with Option A (in-process) for MVP. Migrate to Option B when transaction volume increases.

4.4 Dead Letter Queue

After max retries exhausted:

1. **Mark transaction as** `failed` with reason: `"PISP provider unreachable after 3 attempts"`
2. **Create admin alert** in separate table:

```
CREATE TABLE admin_alerts (
  id TEXT PRIMARY KEY,
  alert_type TEXT NOT NULL, -- 'transaction_stuck', 'pisp_failure', etc.
  severity TEXT NOT NULL CHECK(severity IN ('low','medium','high','critical')),
  resource_type TEXT,
  resource_id TEXT,
  title TEXT NOT NULL,
  description TEXT,
  status TEXT DEFAULT 'open' CHECK(status IN ('open','investigating','resolved','dismissed')),
  created_at TEXT DEFAULT (datetime('now')),
  resolved_at TEXT,
  resolved_by TEXT
);

INSERT INTO admin_alerts (
  id, alert_type, severity, resource_type, resource_id,
  title, description
) VALUES (
  'alert_xyz', 'transaction_stuck', 'high', 'transaction', 'tx_rem_123',
  'Transaction failed after 3 retries',
  'Transaction tx_rem_123 (user: usr_abc, amount: 500 NOK) failed to process after 3 attempts.
PISP provider returned: "Service Unavailable". Manual investigation required.'
```

```
);
```

3. **Send Slack/email to ops team** (via webhook or existing notification system)
4. **Admin dashboard shows alert** at `/admin/alerts` with:
   - Transaction details
   - Retry history (from audit_log)
   - Manual actions: "Retry Now", "Refund User", "Mark Resolved"

---

5. Timeout Recovery

5.1 Scenario

User initiates payment → PISP accepts request → network drops → no response after 30s → transaction stuck in `processing`

Current behavior: API returns error, transaction never completes

New behavior: Mark as `timeout`, schedule background reconciliation

5.2 Implementation

Step 1: On timeout, transition to `timeout` status

```
// In payments.ts
if (error instanceof Error && error.name === "AbortError") {
  // Don't fail immediately — schedule status check
  await updateTransactionStatus(txId, "timeout", "PISP request timeout - will check status
later");

  // Enqueue background reconciliation job (runs after 2 min)
  await scheduleStatusCheck(txId, 120000); // 2 minutes

  return {
    success: true, // ← YES! Tell API route we handled it
    status: "timeout",
    message: "Payment is processing — we'll notify you when complete"
  };
}
```

Step 2: Background worker checks status

```
// reconciliation-worker.ts
async function checkTransactionStatus(txId: string) {
  const tx = await getOne("SELECT * FROM transactions WHERE id = ?", [txId]);
  if (!tx) return;

  // Call PISP provider's GET /payments/{id} endpoint
  const status = await pispProvider.getPaymentStatus(tx.external_id);

  if (status.completed) {
    await updateTransactionStatus(txId, "completed", "Confirmed via reconciliation");
    await notifyUser(tx.user_id, "payment_completed", { txId });
  } else if (status.failed) {
    await updateTransactionStatus(txId, "failed", status.reason);
    await notifyUser(tx.user_id, "payment_failed", { txId, reason: status.reason });
  } else {
    // Still processing — check again in 5 min
    await scheduleStatusCheck(txId, 300000); // 5 minutes
  }
}
```

Step 3: Periodic sweep (every 10 minutes)

Find all transactions stuck in `timeout` or `processing` for > 10 minutes:

```
SELECT id FROM transactions
WHERE status IN ('timeout', 'processing')
  AND created_at < datetime('now', '-10 minutes')
LIMIT 100;
```

For each: call `checkTransactionStatus(txId)`

5.3 User Experience

User sees:

1. **Immediate response:** "Processing your payment — we'll send you a notification when it's complete" (status 202)
2. **Push notification (1-2 min later):** "Your 500 NOK payment to Mama Jasmina is complete"
3. **Transaction list updates:** Polling `/api/transactions` or WebSocket push

What if it never completes?

- After 24 hours stuck in `timeout` → mark as `failed` + admin alert
- User can contact support via `/support` page
- Admin manually investigates + refunds if needed

---

6. Partial Failure Handling

6.1 Scenario (Future — FX Conversion)

Remittance flow with FX conversion:

1. User sends 500 NOK → 5,085 RSD
2. FX conversion succeeds (NOK debited from user's bank)
3. International transfer fails (recipient bank rejects)
4. **Problem:** User's money is gone, recipient didn't receive it

Current code: No FX conversion step (demo uses hardcoded exchange rates)

Future risk: When FX provider is added, must handle partial success

6.2 Classification

Scenario	Recoverable?	Action
FX success + transfer success	N/A	✅ Complete
FX success + transfer fail	✅ Yes	Refund converted amount back to NOK
FX fail + transfer not attempted	✅ Yes	Transaction never started, return error
FX timeout + transfer unknown	⚠ Maybe	Check FX provider status, then refund or complete

6.3 Compensation Flow (When FX Added)

Database changes:

Add `compensation_status` field:

```
ALTER TABLE transactions ADD COLUMN compensation_status TEXT CHECK(
  compensation_status IN ('none', 'pending', 'completed', 'failed')
) DEFAULT 'none';
```

Flow:

```
// 1. Attempt FX conversion
const fxResult = await fxProvider.convert({ from: "NOK", to: "RSD", amount: 500 });

if (!fxResult.success) {
  await updateTransactionStatus(txId, "failed", "FX conversion failed");
  return { success: false, status: "failed", error: fxResult.error };
}

// 2. Mark FX complete
await run("UPDATE transactions SET fx_completed_at = datetime('now'), fx_external_id = ? WHERE
id = ?",
  [fxResult.externalId, txId]);

// 3. Attempt international transfer
const transferResult = await pispProvider.transferInternational({ ... });

if (!transferResult.success) {
  // Transfer failed — need to refund FX
  await updateTransactionStatus(txId, "partially_completed", "Transfer failed, initiating
refund");
  await run("UPDATE transactions SET compensation_status = 'pending' WHERE id = ?", [txId]);

  // 4. Initiate refund (convert RSD back to NOK + credit user's bank account)
  const refundResult = await fxProvider.refund({
    originalConversionId: fxResult.externalId,
    recipientBankAccountId: tx.from_bank_account_id
  });

  if (refundResult.success) {
    await updateTransactionStatus(txId, "failed", "Transfer failed, refund completed");
    await run("UPDATE transactions SET compensation_status = 'completed' WHERE id = ?",
[txId]);
  } else {
    // Refund also failed — escalate to manual review
```

```
    await updateTransactionStatus(txId, "failed", "Transfer and refund failed - manual review
required");
    await run("UPDATE transactions SET compensation_status = 'failed' WHERE id = ?", [txId]);
    await createAdminAlert({
      type: "compensation_failed",
      severity: "critical",
      resourceId: txId,
      title: "Refund failed after partial payment",
      description: `Transaction ${txId}: FX conversion succeeded (${fxResult.externalId}) but
transfer and refund both failed. User's 500 NOK is stuck in limbo. URGENT MANUAL INTERVENTION
REQUIRED.`
    });
  }
}
```

SLA: Refund must complete within 24 hours (PSD2 requirement)

6.4 Edge Cases

Q: What if refund takes 48 hours? A: Status remains `partially_completed` until refund clears. User sees: "Processing refund — this may take up to 2 business days"

Q: What if user's bank account is closed? A: Refund fails → admin alert → manual investigation → refund via alternative method (e.g., bank transfer to new account)

Q: What if FX provider goes down during refund? A: Retry with exponential backoff (same logic as Step 4). After max retries → admin alert.

---

7. User Communication

7.1 Transaction Status Page

Route: `/transactions/[id]`

Content:

```
// src/app/transactions/[id]/page.tsx

export default function TransactionDetailPage({ params }: { params: { id: string } }) {
```

```
  const { data: tx } = useSWR(`/api/transactions/${params.id}`, fetcher, {
    refreshInterval: tx?.status === "processing" || tx?.status === "timeout" ? 2000 : 0
  });

  if (!tx) return <div>Loading...</div>;

  return (
    <div className="p-6">
      <StatusBadge status={tx.status} />
      <h1 className="text-2xl font-semibold mt-4">{tx.type === "remittance" ? "Money Transfer"
: "QR Payment"}</h1>

      {/* Real-time status */}
      <div className="mt-6">
        {tx.status === "initiated" && <StatusMessage icon="" message="Initiating payment..."
/>}
        {tx.status === "processing" && <StatusMessage icon="" message="Your payment is being
processed" />}
        {tx.status === "timeout" && <StatusMessage icon="" message="Processing your payment —
we'll notify you when complete" />}
        {tx.status === "completed" && <StatusMessage icon="" message="Payment completed" />}
        {tx.status === "failed" && <StatusMessage icon="" message={`Payment failed:
${tx.failure_reason || "Unknown error"}`} />}
      </div>

      {/* Timeline */}
      <div className="mt-8">
        <h2 className="font-medium mb-4">Timeline</h2>
        <Timeline events={tx.timeline} />
      </div>

      {/* Details */}
      <div className="mt-8 grid grid-cols-2 gap-4">
        <DetailRow label="Amount" value={`${tx.amount} ${tx.currency}`} />
        <DetailRow label="Fee" value={`${tx.fee} ${tx.currency}`} />
        {tx.type === "remittance" && (
          <>
            <DetailRow label="Recipient" value={tx.recipient_name} />
            <DetailRow label="Exchange Rate" value={tx.exchange_rate} />
            <DetailRow label="Recipient Gets" value={`${tx.receive_amount}
```

```
${tx.receive_currency}`} />
            <DetailRow label="ETA" value={tx.eta || "1-2 business days"} />
          </>
        )}
        <DetailRow label="Transaction ID" value={tx.id} />
        <DetailRow label="Created" value={new Date(tx.created_at).toLocaleString("nb-NO")} />
      </div>

      {/* Actions */}
      {tx.status === "failed" && (
        <button className="mt-6 btn-primary" onClick={() => retryTransaction(tx.id)}>
          Try Again
        </button>
      )}
    </div>
  );
}
```

Timeline data:

API response includes `timeline` array:

```
{
  "id": "tx_rem_123",
  "status": "completed",
  "timeline": [
    { "timestamp": "2026-02-17T10:00:00Z", "event": "created", "message": "Payment initiated"
},
    { "timestamp": "2026-02-17T10:00:02Z", "event": "processing", "message": "Sent to bank" },
    { "timestamp": "2026-02-17T10:00:45Z", "event": "completed", "message": "Payment confirmed
by bank" }
  ]
}
```

Fetched from `audit_log` table where `resource_id = tx.id` and `action LIKE 'TRANSACTION_%'`

7.2 Push Notifications

When to send:

Status Change	Title	Body

`processing` → `completed`	"Payment Complete"	"Your 500 NOK payment to Mama Jasmina is complete"
`processing` → `failed`	"Payment Failed"	"Your 500 NOK payment failed. Tap to view details"
`timeout` → `completed`	"Payment Complete"	"Your payment has been confirmed by the bank"
`partially_completed` → `failed`	"Refund Processed"	"Your 500 NOK has been refunded to your account"

Implementation:

```
// lib/services/notifications.ts
export async function sendPushNotification(params: {
  userId: string;
  title: string;
  body: string;
  data: Record<string, string>;
}) {
  // Check user settings
  const settings = await getOne("SELECT push_enabled FROM settings WHERE user_id = ?",
[params.userId]);
  if (!settings?.push_enabled) return;

  // Get user's push tokens (stored in separate table)
  const tokens = await query<{ token: string }>(
    "SELECT token FROM push_tokens WHERE user_id = ? AND active = 1",
    [params.userId]
  );

  // Send via Firebase Cloud Messaging (FCM) or Apple Push Notification Service (APNS)
  for (const { token } of tokens) {
    await fcm.send({
      token,
      notification: { title: params.title, body: params.body },
      data: params.data,
    });
  }

  // Log notification
  await run(
    "INSERT INTO notifications (id, user_id, type, title, body) VALUES (?, ?, ?, ?, ?)",
```

```
    [randomId("ntf"), params.userId, "push", params.title, params.body]
  );
}
```

Call from status update:

```
async function updateTransactionStatus(
  txId: string,
  newStatus: string,
  reason?: string
) {
  const tx = await getOne("SELECT * FROM transactions WHERE id = ?", [txId]);
  if (!tx) throw new Error("Transaction not found");

  // Update status
  await run("UPDATE transactions SET status = ?, updated_at = datetime('now') WHERE id = ?",
    [newStatus, txId]);

  // Log audit
  await logAudit({ ... });

  // Send push notification
  if (newStatus === "completed" || newStatus === "failed") {
    await notifications.sendPushNotification({
      userId: tx.user_id,
      title: newStatus === "completed" ? "Payment Complete" : "Payment Failed",
      body: newStatus === "completed"
        ? `Your ${tx.amount} NOK payment is complete`
        : `Your ${tx.amount} NOK payment failed${reason ? `: ${reason}` : ""}`,
      data: { txId, status: newStatus },
    });
  }
}
```

7.3 Email Notifications

When to send: Only for terminal states (`completed`, `failed`)

Template:

```
<!-- email-templates/transaction-completed.html -->
<html>
<body style="font-family: Inter, sans-serif;">
  <div style="max-width: 600px; margin: 0 auto; padding: 20px;">
    <h1>Payment Complete</h1>
    <p>Your payment of <strong>{{amount}} {{currency}}</strong> to
<strong>{{recipientName}}</strong> has been completed.</p>
    <p><strong>Transaction ID:</strong> {{txId}}</p>
    <p><strong>Date:</strong> {{completedAt}}</p>
    <a href="https://getdrop.no/transactions/{{txId}}" style="display: inline-block; padding:
12px 24px; background: #00E5A0; color: #000; text-decoration: none; border-radius: 8px;
margin-top: 20px;">
      View Transaction
    </a>
  </div>
</body>
</html>
```

Send via existing email service:

```
// lib/services/email.ts
import { email } from "@/lib/services";

await email.send({
  to: user.email,
  subject: "Payment Complete",
  template: "transaction-completed",
  data: {
    amount: tx.amount,
    currency: tx.currency,
    recipientName: tx.recipient_name,
    txId: tx.id,
    completedAt: new Date(tx.completed_at).toLocaleString("nb-NO"),
  },
});
```

7.4 Error Messages (User-Friendly)

Current: Generic errors like "PISP API error: 500"

New: Human-readable messages

Error Code	User-Facing Message (Norwegian)	English
`insufficient_balance`	"Ikke nok dekning på bankkontoen"	"Insufficient funds in your bank account"
`bank_declined`	"Banken din avslo betalingen. Kontakt banken for detaljer."	"Your bank declined the payment. Contact your bank for details."
`invalid_iban`	"Ugyldig kontonummer. Sjekk mottakerens kontoopplysninger."	"Invalid account number. Check recipient's account details."
`pisp_timeout`	"Betalingen tar lengre tid enn vanlig. Vi varsler deg når den er fullført."	"Payment is taking longer than usual. We'll notify you when complete."
`pisp_unavailable`	"Vår betalingsleverandør er midlertidig utilgjengelig. Prøv igjen om noen minutter."	"Our payment provider is temporarily unavailable. Try again in a few minutes."
`max_retries_exceeded`	"Betalingen feilet etter flere forsøk. Kontakt kundestøtte."	"Payment failed after multiple attempts. Contact support."

Implementation:

```
// lib/error-messages.ts
const ERROR_MESSAGES: Record<string, { no: string; en: string }> = {
  insufficient_balance: {
    no: "Ikke nok dekning på bankkontoen",
    en: "Insufficient funds in your bank account"
  },
  // ... all errors above
};

export function getUserFacingError(errorCode: string, language: "no" | "en" = "no"): string {
  return ERROR_MESSAGES[errorCode]?.[language] || ERROR_MESSAGES.default[language];
}
```

---

8. Admin Tools

8.1 Stuck Transactions Endpoint

Route: `GET /api/admin/transactions/stuck`

Access: Requires admin role (check JWT: `user.role === 'admin'`)

Query:

```
SELECT
  t.id,
  t.user_id,
  t.type,
  t.status,
  t.amount,
  t.currency,
  t.created_at,
  t.updated_at,
  u.email AS user_email,
  u.first_name || ' ' || u.last_name AS user_name,
  (julianday('now') - julianday(t.created_at)) * 24 AS hours_stuck
FROM transactions t
JOIN users u ON t.user_id = u.id
WHERE t.status IN ('processing', 'timeout', 'partially_completed')
  AND t.created_at < datetime('now', '-10 minutes')
ORDER BY t.created_at ASC
LIMIT 100;
```

Response:


```
{
  "data": [
    {
      "id": "tx_rem_456",
      "userId": "usr_abc",
      "userName": "Amir Hadžić",
      "userEmail": "amir@example.com",
      "type": "remittance",
      "status": "timeout",
      "amount": 500,
      "currency": "NOK",
      "createdAt": "2026-02-17T08:00:00Z",
      "hoursStuck": 2.5
    }
  ],
  "total": 1
}
```

8.2 Manual Retry Endpoint

Route: `POST /api/admin/transactions/[id]/retry`

Access: Admin only

Action:

1. Validate transaction is in retryable state (`timeout`, `failed` with transient error)
2. Reset retry counter
3. Call PISP provider again (with retry logic from Section 4)
4. Log admin action in audit_log

Implementation:

```
// src/app/api/admin/transactions/[id]/retry/route.ts
export async function POST(
  request: NextRequest,
  { params }: { params: { id: string } }
) {
  const { user, error } = await requireAuth(request);
  if (error) return error;

  if (user.role !== "admin") {
    return jsonError("forbidden", "Admin access required", 403);
  }

  const txId = params.id;
  const tx = await getOne("SELECT * FROM transactions WHERE id = ?", [txId]);

  if (!tx) {
    return jsonError("not_found", "Transaction not found", 404);
  }

  if (!["timeout", "failed"].includes(tx.status)) {
    return jsonError("invalid_state", "Transaction is not retryable", 400);
  }

  // Log admin action
  await logAudit({
    userId: user.id,
```

```
    action: "ADMIN_TRANSACTION_RETRY",
    resourceType: "transaction",
    resourceId: txId,
    details: { previousStatus: tx.status },
    ipAddress: getClientIp(request),
    requestId: getRequestId(request.headers),
  });

  // Reset transaction to initiated
  await run("UPDATE transactions SET status = 'initiated', retry_count = 0 WHERE id = ?",
[txId]);

  // Re-call PISP with retry logic
  const result = tx.type === "remittance"
    ? await payments.initiateRemittance({ ... })
    : await payments.initiateQrPayment({ ... });

  if (result.success) {
    return NextResponse.json({ message: "Retry initiated", status: result.status });
  } else {
    return jsonError("retry_failed", result.error || "Retry failed", 500);
  }
}
```

8.3 Manual Resolution Endpoint

Route: `POST /api/admin/transactions/[id]/resolve`

Body:

```
{
  "action": "mark_completed" | "mark_failed" | "initiate_refund",
  "reason": "Admin manually verified with bank",
  "externalReference": "bank_ref_12345" // optional
}
```

Actions:

Action	Effect
`mark_completed`	Set status to `completed`, add admin note to audit_log

Action	Effect
`mark_failed`	Set status to `failed`, add reason, notify user
`initiate_refund`	Trigger refund flow (for partially_completed), set compensation_status to `pending`

Implementation:

```
export async function POST(request: NextRequest, { params }: { params: { id: string } }) {
  const { user, error } = await requireAuth(request);
  if (error) return error;
  if (user.role !== "admin") return jsonError("forbidden", "Admin access required", 403);

  const body = await request.json();
  const { action, reason, externalReference } = body;

  const txId = params.id;
  const tx = await getOne("SELECT * FROM transactions WHERE id = ?", [txId]);
  if (!tx) return jsonError("not_found", "Transaction not found", 404);

  switch (action) {
    case "mark_completed":
      await run("UPDATE transactions SET status = 'completed', completed_at = datetime('now')
WHERE id = ?", [txId]);
      await logAudit({ userId: user.id, action: "ADMIN_MARK_COMPLETED", resourceId: txId,
details: { reason, externalReference } });
      await notifications.sendPushNotification({ userId: tx.user_id, title: "Payment
Complete", body: "Your payment has been confirmed" });
      return NextResponse.json({ message: "Transaction marked as completed" });

    case "mark_failed":
      await run("UPDATE transactions SET status = 'failed', failure_reason = ? WHERE id = ?",
[reason, txId]);
      await logAudit({ userId: user.id, action: "ADMIN_MARK_FAILED", resourceId: txId,
details: { reason } });
      await notifications.sendPushNotification({ userId: tx.user_id, title: "Payment Failed",
body: reason });
      return NextResponse.json({ message: "Transaction marked as failed" });

    case "initiate_refund":
      // TODO: Call refund provider
```

```
      await run("UPDATE transactions SET compensation_status = 'pending' WHERE id = ?",
[txId]);
      await logAudit({ userId: user.id, action: "ADMIN_INITIATE_REFUND", resourceId: txId,
details: { reason } });
      return NextResponse.json({ message: "Refund initiated" });

    default:
      return jsonError("invalid_action", "Invalid action", 400);
  }
}
```

8.4 Admin Dashboard

Route: `/admin/transactions`

Features:

1. **Overview Cards:**
   - Stuck transactions (count)
   - Failed last 24h (count)
   - Average resolution time
2. **Stuck Transactions Table:**
   - Columns: TX ID, User, Amount, Status, Hours Stuck, Actions
   - Actions: "Retry", "Resolve", "View Audit Log"
3. **Filters:**
   - Status (processing, timeout, partially_completed)
   - Stuck > X hours
   - User search (email, ID)

Screenshot mockup:

```
┌─────────────────────────────────────────────────┐
│  Admin: Stuck Transactions                      │
├─────────────────────────────────────────────────┤
│  [ Stuck: 3 ]  [ Failed 24h: 12 ]  [ Avg: 1.2h ]  │
├─────────────────────────────────────────────────┤
│  Filters: [Status: All ▼] [Stuck > 1h ▼]          │
├─────────────────────────────────────────────────┤
│  TX ID       | User          | Amount | Status   | Hours | Actions        |
│  tx_rem_456  | amir@ex.com   | 500 NOK| timeout  | 2.5   | [Retry][Resolve]|
│  tx_qr_789   | sara@ex.com   | 129 NOK| processing| 0.8   | [Retry][Resolve]|
└─────────────────────────────────────────────────┘
```

9. Database Schema Changes

9.1 New Columns on `transactions` Table

```
-- Retry tracking
ALTER TABLE transactions ADD COLUMN retry_count INTEGER DEFAULT 0;
ALTER TABLE transactions ADD COLUMN last_retry_at TEXT;

-- External references
ALTER TABLE transactions ADD COLUMN external_id TEXT; -- PISP provider's transaction ID
ALTER TABLE transactions ADD COLUMN external_status TEXT; -- Raw status from provider

-- Failure details
ALTER TABLE transactions ADD COLUMN failure_reason TEXT;
ALTER TABLE transactions ADD COLUMN failure_code TEXT; -- Machine-readable error code

-- Compensation (for partial failures)
ALTER TABLE transactions ADD COLUMN compensation_status TEXT CHECK(
  compensation_status IN ('none', 'pending', 'completed', 'failed')
) DEFAULT 'none';
ALTER TABLE transactions ADD COLUMN compensation_completed_at TEXT;

-- Timeline
ALTER TABLE transactions ADD COLUMN updated_at TEXT DEFAULT (datetime('now'));

-- FX tracking (future)
ALTER TABLE transactions ADD COLUMN fx_completed_at TEXT;
ALTER TABLE transactions ADD COLUMN fx_external_id TEXT;
```

9.2 New State: `timeout`

Update CHECK constraint:

```
-- Before:
status TEXT DEFAULT 'processing' CHECK(status IN ('processing','completed','failed'))
```

```
-- After:
status TEXT DEFAULT 'initiated' CHECK(status IN
('initiated','processing','timeout','completed','failed','partially_completed'))
```

Migration (SQLite):

SQLite doesn't support `ALTER TABLE ... MODIFY CONSTRAINT`, so recreate table:

```
-- Create new table with updated constraint
CREATE TABLE transactions_new (
  id TEXT PRIMARY KEY,
  user_id TEXT NOT NULL REFERENCES users(id),
  type TEXT NOT NULL CHECK(type IN ('remittance','qr_payment')),
  status TEXT DEFAULT 'initiated' CHECK(status IN
('initiated','processing','timeout','completed','failed','partially_completed')),
  -- ... all other columns
);

-- Copy data
INSERT INTO transactions_new SELECT * FROM transactions;

-- Drop old, rename new
DROP TABLE transactions;
ALTER TABLE transactions_new RENAME TO transactions;

-- Recreate indexes
CREATE UNIQUE INDEX idx_tx_idempotency ON transactions(idempotency_key) WHERE idempotency_key
IS NOT NULL;
CREATE INDEX idx_transactions_user ON transactions(user_id);
CREATE INDEX idx_transactions_merchant ON transactions(merchant_id);
```

9.3 New Table: `admin_alerts`

```
CREATE TABLE admin_alerts (
  id TEXT PRIMARY KEY,
  alert_type TEXT NOT NULL, -- 'transaction_stuck', 'pisp_failure', 'compensation_failed',
etc.
  severity TEXT NOT NULL CHECK(severity IN ('low','medium','high','critical')),
  resource_type TEXT, -- 'transaction', 'user', 'merchant', etc.
```

```
  resource_id TEXT,
  title TEXT NOT NULL,
  description TEXT,
  status TEXT DEFAULT 'open' CHECK(status IN ('open','investigating','resolved','dismissed')),
  created_at TEXT DEFAULT (datetime('now')),
  resolved_at TEXT,
  resolved_by TEXT, -- user_id of admin who resolved
  resolution_notes TEXT
);

CREATE INDEX idx_admin_alerts_status ON admin_alerts(status);
CREATE INDEX idx_admin_alerts_type ON admin_alerts(alert_type);
CREATE INDEX idx_admin_alerts_created ON admin_alerts(created_at);
```

9.4 New Table: `retry_history`

Optional (if want detailed retry logs separate from audit_log):

```
CREATE TABLE retry_history (
  id TEXT PRIMARY KEY,
  transaction_id TEXT NOT NULL REFERENCES transactions(id),
  attempt INTEGER NOT NULL, -- 1, 2, 3
  started_at TEXT DEFAULT (datetime('now')),
  completed_at TEXT,
  success INTEGER DEFAULT 0, -- 0 = failed, 1 = succeeded
  error_code TEXT,
  error_message TEXT,
  pisp_response TEXT -- Full JSON response from PISP provider
);

CREATE INDEX idx_retry_history_tx ON retry_history(transaction_id);
```

Alternative: Use `audit_log` table (already exists, sufficient for MVP)

---

10. File Structure & Implementation Checklist

10.1 Files to Create

```
src/
├── app/
│   ├── api/
│   │   ├── transactions/
│   │   │   ├── [id]/
│   │   │   │   ├── route.ts          # GET transaction by ID (add timeline)
│   │   │   │   └── retry/route.ts     # NEW: POST retry transaction (user-facing, for failed
txs)
│   │   ├── admin/
│   │   │   ├── transactions/
│   │   │   │   ├── stuck/route.ts     # NEW: GET stuck transactions
│   │   │   │   └── [id]/
│   │   │   │       ├── retry/route.ts  # NEW: POST admin retry
│   │   │   │       └── resolve/route.ts # NEW: POST admin manual resolution
│   │   │   └── alerts/
│   │   │       ├── route.ts            # NEW: GET admin alerts (list)
│   │   │       └── [id]/route.ts       # NEW: PATCH resolve alert
│   ├── transactions/
│   │   └── [id]/
│   │       └── page.tsx                # NEW: Transaction detail page
│   └── admin/
│       ├── transactions/
│       │   └── page.tsx                # NEW: Admin stuck transactions dashboard
│       └── alerts/
│           └── page.tsx                # NEW: Admin alerts dashboard
├── lib/
│   ├── services/
│   │   ├── payments.ts                 # MODIFY: Add retry logic + timeout handling
│   │   ├── reconciliation.ts           # NEW: Background status checks
│   │   └── notifications.ts            # MODIFY: Add transaction notifications
│   ├── db-migrations/
│   │   └── 004-transaction-recovery.sql # NEW: Schema changes
│   ├── retry.ts                        # NEW: Retry logic (exponential backoff)
│   ├── state-machine.ts                # NEW: Transaction state transitions
│   ├── error-messages.ts               # NEW: User-friendly error messages
│   └── admin-alerts.ts                 # NEW: Admin alert creation/management
└── workers/
```

```
    └── reconciliation-worker.ts       # NEW: Background job to check stuck txs
```

10.2 Implementation Phases

Phase 1: State Machine & Audit (Week 1)

- [ ] Update `transactions` table schema (new columns + `timeout` state)
- [ ] Implement state transition validation in `lib/state-machine.ts`
- [ ] Add transition audit logging (every status change → audit_log)
- [ ] Update API routes to use state machine validation
- [ ] **Deliverable:** Status changes are validated + audited

Phase 2: Retry Logic (Week 2)

- [ ] Implement exponential backoff in `lib/retry.ts`
- [ ] Add error classification (transient vs permanent)
- [ ] Update `payments.ts` to use retry wrapper
- [ ] Add retry counter tracking in DB
- [ ] **Deliverable:** Transient errors auto-retry up to 3 times

Phase 3: Timeout Recovery (Week 2-3)

- [ ] Change timeout behavior: return `timeout` status instead of failure
- [ ] Create `reconciliation-worker.ts` background job
- [ ] Implement PISP status polling (every 10 min for stuck txs)
- [ ] Add timeout → completed/failed transitions
- [ ] **Deliverable:** Timeouts self-resolve via background reconciliation

Phase 4: User Communication (Week 3)

- [ ] Create transaction detail page (`/transactions/[id]`)
- [ ] Add real-time status polling (SWR with 2s refresh)
- [ ] Implement push notifications for status changes
- [ ] Add email notifications for terminal states
- [ ] Implement user-friendly error messages
- [ ] **Deliverable:** Users always know transaction status

Phase 5: Admin Tools (Week 4)

- [ ] Create `admin_alerts` table
- [ ] Implement stuck transaction detection (every 10 min sweep)
- [ ] Build admin dashboard (`/admin/transactions`)
- [ ] Add manual retry endpoint (`POST /api/admin/transactions/[id]/retry`)
- [ ] Add manual resolution endpoint (`POST /api/admin/transactions/[id]/resolve`)
- [ ] **Deliverable:** Admins can intervene on stuck transactions

Phase 6: Partial Failure Handling (Future — After FX Provider Integration)

- [ ] Add `compensation_status` field
- [ ] Implement refund flow for partial failures
- [ ] Add FX provider status checks
- [ ] Test compensation scenarios
- [ ] **Deliverable:** Partial failures trigger automatic refunds

---

11. Testing Strategy

11.1 Unit Tests

Retry logic:

```
describe("Retry with exponential backoff", () => {
  test("succeeds on first attempt", async () => {
    const result = await callPispWithRetry(() => Promise.resolve({ success: true }));
    expect(result.success).toBe(true);
  });

  test("retries on transient error", async () => {
    let attempts = 0;
    const result = await callPispWithRetry(async () => {
      attempts++;
      if (attempts < 3) throw new Error("Network timeout");
      return { success: true };
    });
    expect(attempts).toBe(3);
```

```
  });

  test("stops on permanent error", async () => {
    let attempts = 0;
    const result = await callPispWithRetry(async () => {
      attempts++;
      return { success: false, error: "invalid_iban" }; // permanent
    });
    expect(attempts).toBe(1);
  });
});
```

State machine:

```
describe("Transaction state machine", () => {
  test("allows initiated → processing", () => {
    expect(canTransition("initiated", "processing")).toBe(true);
  });

  test("blocks processing → initiated", () => {
    expect(canTransition("processing", "initiated")).toBe(false);
  });

  test("blocks completed → anything", () => {
    expect(canTransition("completed", "failed")).toBe(false);
  });
});
```

11.2 Integration Tests

Scenario: Timeout recovery

1. Mock PISP to timeout on first call
2. Initiate transaction → verify status = `timeout`
3. Run reconciliation worker
4. Mock PISP to return `completed`
5. Verify transaction status = `completed`
6. Verify push notification sent

Scenario: Retry exhaustion

1. Mock PISP to return 503 three times
2. Initiate transaction
3. Verify transaction status = `failed`
4. Verify admin alert created
5. Verify user notified

11.3 End-to-End Tests

User journey:

1. User initiates remittance (500 NOK → RSD)
2. PISP times out after 30s
3. User sees "Processing — we'll notify you"
4. 2 minutes later: background worker checks status
5. PISP returns `completed`
6. User receives push notification
7. User opens transaction detail page → sees "Completed"
8. User receives email confirmation

Admin journey:

1. Transaction stuck in `timeout` for 2 hours
2. Admin opens `/admin/transactions` dashboard
3. Sees transaction in "Stuck" list
4. Clicks "Retry" → transaction re-attempted
5. PISP succeeds → status = `completed`
6. Admin marks alert as "Resolved"

---

12. Acceptance Criteria

12.1 State Machine

- ☑ All status transitions validated against whitelist
- ☑ Invalid transitions blocked at DB + app level
- ☑ Every status change logged in `audit_log` with timestamp + reason
- ☑ Terminal states (`completed`, `failed`) cannot transition

12.2 Idempotency

☑ Duplicate requests with same `idempotencyKey` return cached response (already implemented)

☑ Idempotency keys scoped to user (prevents IDOR) (already implemented)

☑ Response includes identical payload + status 200 (already implemented)

12.3 Retry Logic

☑ Transient errors (5xx, timeout) trigger automatic retry

☑ Exponential backoff: 2s → 8s → 32s (with jitter)

☑ Max 3 retry attempts

☑ Permanent errors (4xx, bank decline) fail immediately (no retry)

☑ After max retries: mark as `failed` + create admin alert

12.4 Timeout Recovery

☑ Timeout returns `timeout` status (not `failed`)

☑ Background worker checks PISP status every 10 min

☑ Stuck transactions (> 10 min) swept periodically

☑ Timeout → completed/failed based on PISP response

☑ User notified when status resolves

12.5 Partial Failure

☑ `compensation_status` field added (for future FX refunds)

☑ Refund flow triggers on transfer failure (when FX provider added)

☑ Compensation failures escalate to admin alert

☑ User sees "Processing refund" status

12.6 User Communication

☑ Transaction detail page (`/transactions/[id]`) shows:
- Real-time status
- Timeline of events
- User-friendly error messages

☑ Push notifications sent on status change

☑ Email sent on terminal status (`completed`, `failed`)

☑ Error messages in Norwegian (primary) + English

12.7 Admin Tools

☑ `/api/admin/transactions/stuck` returns all stuck transactions

☑ `/api/admin/transactions/[id]/retry` manually retries transaction

☑ `/api/admin/transactions/[id]/resolve` manually marks completed/failed

☑ Admin dashboard shows stuck transactions with action buttons

☑ Admin alerts created for:

- Max retries exhausted
- Compensation failure
- Transaction stuck > 24 hours

---

13. Monitoring & Alerting

13.1 Metrics to Track

Metric	Threshold	Alert If
Stuck transactions (count)	5	> 10
Average resolution time (hours)	1	> 4
Failed transactions (last 24h)	50	> 100
PISP timeout rate (%)	5%	> 15%
Retry success rate (%)	80%	< 60%
Compensation failures (count)	0	> 0

13.2 Dashboard Queries

Stuck transactions:

```
SELECT COUNT(*) FROM transactions
WHERE status IN ('processing', 'timeout')
  AND created_at < datetime('now', '-10 minutes');
```

Average resolution time:

```
SELECT AVG(julianday(completed_at) - julianday(created_at)) * 24 AS hours
FROM transactions
WHERE status = 'completed'
  AND completed_at > datetime('now', '-24 hours');
```

PISP timeout rate:

```
SELECT
  SUM(CASE WHEN failure_code = 'pisp_timeout' THEN 1 ELSE 0 END) * 100.0 / COUNT(*) AS
timeout_pct
FROM transactions
WHERE created_at > datetime('now', '-24 hours');
```

13.3 Log Events

Every transaction state change:

```
{
  "level": "info",
  "msg": "Transaction status changed",
  "txId": "tx_rem_123",
  "userId": "usr_abc",
  "from": "processing",
  "to": "completed",
  "reason": "PISP callback received",
  "externalId": "ext_456",
  "timestamp": "2026-02-17T10:00:45Z"
}
```

PISP API call failures:

```
{
  "level": "error",
  "msg": "PISP API call failed",
  "txId": "tx_rem_123",
  "attempt": 2,
  "errorCode": "pisp_timeout",
  "errorMessage": "Request timeout after 30s",
  "willRetry": true,
  "nextRetryIn": "8000ms",
```

```
  "timestamp": "2026-02-17T10:00:30Z"
}
```

Retry exhaustion:

```
{
  "level": "error",
  "msg": "All retries exhausted",
  "txId": "tx_rem_123",
  "maxRetries": 3,
  "lastError": "PISP provider unavailable",
  "adminAlertCreated": "alert_xyz",
  "timestamp": "2026-02-17T10:01:10Z"
}
```

---

14. Security Considerations

14.1 Admin Endpoints

Access control:

- All admin endpoints require `user.role === 'admin'` (checked via JWT)
- Audit every admin action (`ADMIN_TRANSACTION_RETRY`, `ADMIN_MARK_COMPLETED`, etc.)
- Log IP address + user agent for all admin operations

Rate limiting:

- Admin endpoints: 60 requests/min (higher than user endpoints)
- Admin dashboard: no rate limit (internal tool)

14.2 Idempotency Key Security

No vulnerability: Idempotency keys scoped to user → can't replay another user's transaction

Best practice: Client generates key = `${userId}_${timestamp}_${random}` (prevents guessing)

14.3 Transaction Status Leaks

Risk: User A checks `/api/transactions/tx_rem_123` → sees User B's transaction

Mitigation (already implemented):

```
const tx = await getOne(
  "SELECT * FROM transactions WHERE id = ? AND user_id = ?",
  [txId, user.id] // ← Scoped to logged-in user
);
```

Admin endpoints: Bypass user_id check (admin sees all transactions)

---

15. Cost Analysis

15.1 Infrastructure

Component	Cost	Notes
Background worker	$0	Same server process (cron or `setInterval`)
Job queue (pg-boss)	$0	Uses existing PostgreSQL (when migrated from SQLite)
Job queue (BullMQ)	~$20/mo	Redis hosting (if chosen over pg-boss)
Push notifications (FCM)	Free	Up to unlimited (Firebase Cloud Messaging)
Email (SendGrid)	$15/mo	50k emails/month (transactional tier)

Total: $15-35/mo (depending on job queue choice)

15.2 PISP API Costs

Retry costs:

- 3 retries per failed transaction
- If 5% of transactions fail → 5% * 3 = 15% extra PISP API calls
- Assuming 10,000 txs/month, 5% fail = 500 failed → 1,500 retry calls
- Cost: depends on PISP provider (typically $0.01-0.05 per API call)
- **Estimated:** $15-75/mo in extra API fees

Reconciliation costs:

- Background worker checks status every 10 min for stuck txs
- If 1% stuck (100 txs/month) → 100 * 6 status checks/hour * 24h = 14,400 API calls
- **Estimated:** $144-720/mo

Optimization: Only check status for transactions stuck > 10 min (reduces unnecessary calls)

---

16. Open Questions (For Alem)

16.1 Retry Strategy

Q1: Should we do in-process retry (Option A) or background job queue (Option B)?

Recommendation: Start with Option A (simpler, no extra infra). Migrate to Option B when transaction volume > 10k/month.

16.2 Notification Channels

Q2: Email + push notifications both? Or only one?

Recommendation: Both. Email is fallback (if user disabled push). Send email only for terminal states.

16.3 Admin Alert Delivery

Q3: How should admin alerts be delivered?

- Option A: Dashboard only (admin must check `/admin/alerts`)
- Option B: Email to ops team
- Option C: Slack webhook
- Option D: SMS for critical alerts

Recommendation: Option C (Slack) for high/critical alerts. Dashboard for all.

16.4 Stuck Transaction Threshold

Q4: When should we mark a transaction as "stuck"?

- Current spec: 10 minutes
- Alternative: 1 hour (less aggressive)

Recommendation: 10 min for reconciliation sweep, 24h for admin alert (gives time to self-resolve)

16.5 Partial Failure Compensation SLA

Q5: What's acceptable refund time for partial failures?

- PSD2 requires 24h for refunds
- Faster = better UX

Recommendation: Initiate refund immediately, complete within 24h (meet regulatory minimum)

---

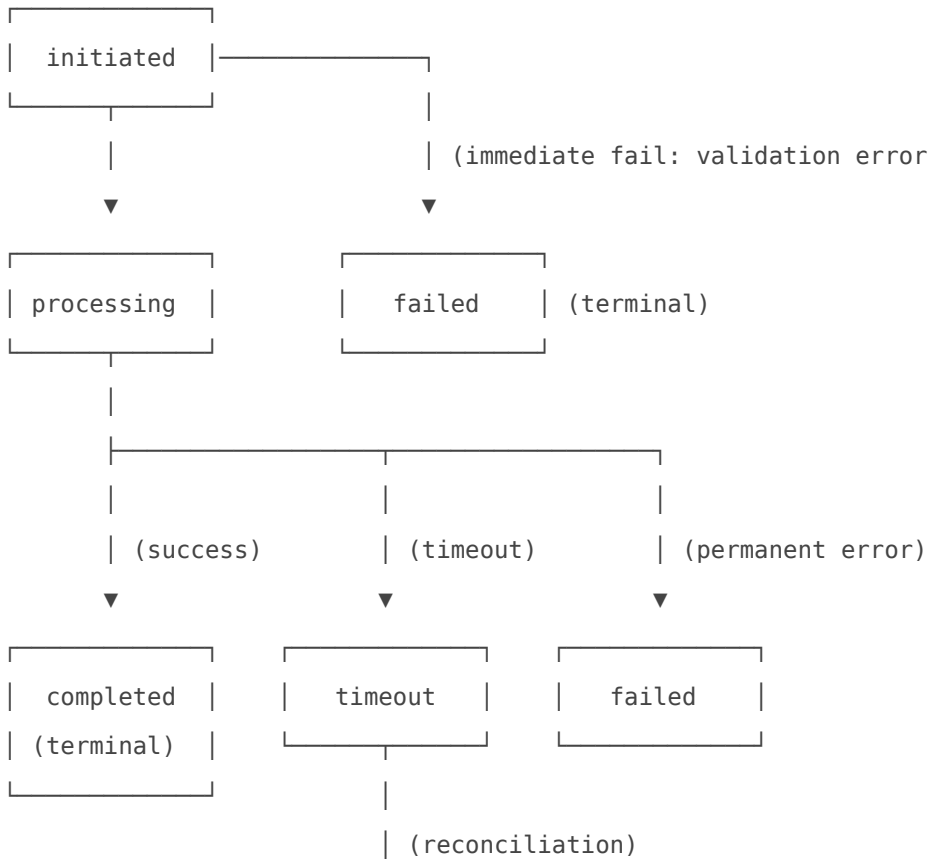
17. Next Steps

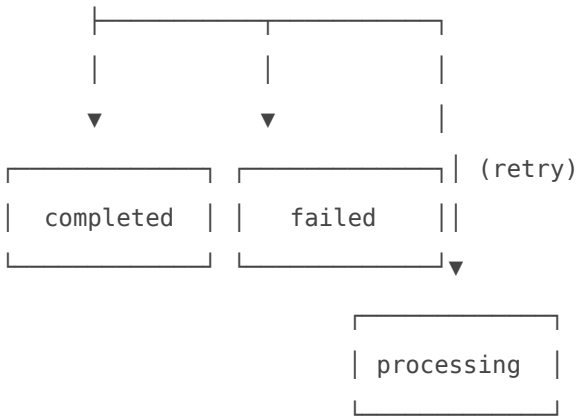
1. **Review this spec** with Alem
2. **Approve/reject each section** (or request changes)
3. **Prioritize phases** (which to implement first?)
4. **Assign to builder agent** (one phase at a time)
5. **Validation after each phase** (validator agent checks implementation)

Estimated timeline: 4 weeks for Phases 1-5, Phase 6 (partial failure) deferred until FX provider integrated

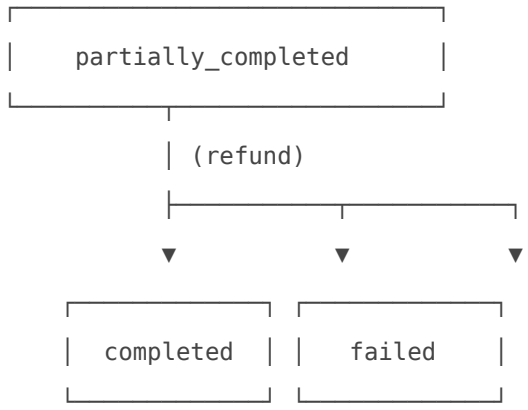
---

Appendix A: State Diagram (ASCII)





Future:



Appendix B: Error Code Reference

Code	Type	Retry?	User Message (NO)	User Message (EN)
`insufficient_balance`	Permanent	No	"Ikke nok dekning på bankkontoen"	"Insufficient funds"
`bank_declined`	Permanent	No	"Banken din avslo betalingen"	"Your bank declined the payment"
`invalid_iban`	Permanent	No	"Ugyldig kontonummer"	"Invalid account number"
`kyc_required`	Permanent	No	"Identitetsverifisering kreves"	"Identity verification required"
`pisp_timeout`	Transient	Yes	"Betalingen tar lengre tid enn vanlig"	"Payment taking longer than usual"
`pisp_unavailable`	Transient	Yes	"Betalingsleverandør midlertidig utilgjengelig"	"Payment provider temporarily unavailable"

Code	Type	Retry?	User Message (NO)	User Message (EN)
`network_error`	Transient	Yes	"Nettverksfeil — prøver igjen automatisk"	"Network error — retrying automatically"
`pisp_5xx`	Transient	Yes	"Betalingsleverandør har tekniske problemer"	"Payment provider experiencing technical issues"
`max_retries_exceeded`	Permanent	No	"Betalingen feilet etter flere forsøk"	"Payment failed after multiple attempts"
`validation_error`	Permanent	No	"Ugyldig forespørsel"	"Invalid request"

End of Specification

Revision #3
Created 2026-02-18 08:44:47 UTC by John
Updated 2026-05-24 20:00:50 UTC by John