Venus-System Production Fixes

Prioritized by Severity

Date: 2026-01-17
Status: Pre-Production Action Plan

🔴 CRITICAL SEVERITY (Block Launch)

These issues MUST be fixed before any production deployment with real money or customers.

C1. Race Condition in Sales - Atomic Sale Creation

Severity: 🔴 CRITICAL
Effort: 2-3 days
Risk if Unfixed: Inventory goes negative, overselling physical stock

Problem

Stock validation and sale creation are separate operations allowing concurrent transactions to oversell:

```
Cashier A: Check stock (10kg) ✓ → Create sale (8kg)
Cashier B: Check stock (10kg) ✓ → Create sale (8kg)
Result: 16kg sold from 10kg stock
```

Files Affected

- `backend/app/routers/poultry_retail/sales.py` (lines 72-141)
- `supabase/migrations/049_sales.sql`

Solution

Create a new PostgreSQL function that atomically:

1. Locks the inventory row using `SELECT ... FOR UPDATE`
2. Validates stock availability
3. Creates the sale record
4. Creates all sale items
5. Commits or rollbacks as a single unit

```sql
CREATE OR REPLACE FUNCTION create_sale_atomic(
    p_store_id INTEGER,
    p_cashier_id UUID,
    p_payment_method TEXT,
    p_sale_type TEXT,
    p_items JSONB,
    p_customer_id UUID DEFAULT NULL,
    p_customer_name TEXT DEFAULT NULL,
    p_customer_phone TEXT DEFAULT NULL,
    p_idempotency_key UUID DEFAULT NULL,
```

```
    p_notes TEXT DEFAULT NULL
) RETURNS UUID AS $$
DECLARE
    v_sale_id UUID;
    v_receipt_number TEXT;
    v_total DECIMAL(12,2) := 0;
    v_item RECORD;
    v_sku RECORD;
    v_available DECIMAL(10,3);
BEGIN
    -- Check idempotency first
    IF p_idempotency_key IS NOT NULL THEN
        SELECT id INTO v_sale_id FROM sales
        WHERE idempotency_key = p_idempotency_key;
        IF FOUND THEN
            RETURN v_sale_id;
        END IF;
    END IF;

    -- Lock and validate all inventory in one pass
    FOR v_item IN SELECT * FROM jsonb_to_recordset(p_items)
        AS x(sku_id UUID, weight DECIMAL, price_snapshot DECIMAL)
    LOOP
        SELECT * INTO v_sku FROM skus WHERE id = v_item.sku_id;

        -- Lock the inventory rows for this store/bird_type/inventory_type
        SELECT COALESCE(SUM(quantity_change), 0) INTO v_available
        FROM inventory_ledger
        WHERE store_id = p_store_id
          AND bird_type = v_sku.bird_type
          AND inventory_type = v_sku.inventory_type
        FOR UPDATE;  -- This locks the aggregation

        IF v_available < v_item.weight THEN
            RAISE EXCEPTION 'Insufficient stock for SKU %. Available: %, Required:
%',
                v_sku.code, v_available, v_item.weight;
        END IF;

        v_total := v_total + (v_item.weight * v_item.price_snapshot);
    END LOOP;

    -- Generate receipt number
    v_receipt_number := generate_receipt_number(p_store_id);

    -- Create sale
    INSERT INTO sales (
        store_id, cashier_id, total_amount, payment_method, sale_type,
        receipt_number, idempotency_key, customer_id, customer_name,
        customer_phone, notes
    ) VALUES (
        p_store_id, p_cashier_id, v_total, p_payment_method::payment_method_enum,
```

```
        p_sale_type::sale_type_enum, v_receipt_number, p_idempotency_key,
        p_customer_id, p_customer_name, p_customer_phone, p_notes
    ) RETURNING id INTO v_sale_id;

    -- Create sale items (trigger will deduct inventory)
    FOR v_item IN SELECT * FROM jsonb_to_recordset(p_items)
        AS x(sku_id UUID, weight DECIMAL, price_snapshot DECIMAL)
    LOOP
        INSERT INTO sale_items (sale_id, sku_id, weight, price_snapshot)
        VALUES (v_sale_id, v_item.sku_id, v_item.weight, v_item.price_snapshot);
    END LOOP;

    RETURN v_sale_id;
END;
$$ LANGUAGE plpgsql;
```

Backend Change

```
# In sales.py create_sale endpoint
result = supabase.rpc("create_sale_atomic", {
    "p_store_id": sale.store_id,
    "p_cashier_id": current_user["user_id"],
    "p_payment_method": sale.payment_method.value,
    "p_sale_type": sale.sale_type.value,
    "p_items": [item.model_dump(mode="json") for item in sale.items],
    "p_idempotency_key": str(idempotency_key),
    # ... other params
}).execute()
```

Verification Steps

1. Create two concurrent API calls with same stock
2. Confirm only one succeeds
3. Verify inventory matches sales

C2. Non-Atomic Multi-Step Operations - Transaction Compensation

Severity: 🔴 CRITICAL
Effort: 3-5 days
Risk if Unfixed: Orphan records, data inconsistency, financial discrepancies

Problem

Multiple Supabase calls without transaction boundaries:

- Sale created → some items fail → orphan sale exists
- Payment created → ledger entry fails → financial mismatch

Files Affected

- `backend/app/routers/poultry_retail/sales.py`
- `backend/app/routers/poultry_retail/payments.py`

- `backend/app/routers/poultry_retail/transfers.py`
- `backend/app/routers/poultry_retail/settlements.py`

Solution

A. Create Atomic RPC Functions (Preferred)

Similar to C1, create PostgreSQL functions for:

- `create_payment_atomic()` - payment + ledger entry
- `approve_transfer_atomic()` - status + both ledger entries
- `submit_settlement_atomic()` - settlement + variance calculation

B. Add Cleanup Worker (Required Either Way)

```
# backend/app/workers/orphan_cleanup.py
async def cleanup_orphan_records():
    """Run every 5 minutes to find and flag orphan records"""
    supabase = get_supabase()

    # Find sales with no items older than 5 minutes
    orphans = supabase.table("sales").select("id, created_at").is_(
        "id", "not.in",
        supabase.table("sale_items").select("sale_id")
    ).lt("created_at", datetime.utcnow() - timedelta(minutes=5)).execute()

    for orphan in orphans.data:
        # Flag for review, don't auto-delete financial records
        supabase.table("sales").update({
            "notes": "ORPHAN_FLAGGED: No items found"
        }).eq("id", orphan["id"]).execute()

        logger.warning(f"Orphan sale flagged: {orphan['id']}")
```

Verification Steps

1. Simulate failure mid-transaction
2. Run cleanup worker
3. Verify orphans are flagged
4. Manual review process documented

C3. Crash Recovery - Saga Pattern Implementation

Severity: 🔴 CRITICAL
Effort: 3-4 days
Risk if Unfixed: Lost transactions, unrecoverable state after server crashes

Problem

If backend crashes during multi-step operation, no recovery mechanism exists.

Solution

A. Transaction Log Table

```sql
CREATE TABLE transaction_log (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    transaction_type VARCHAR(50) NOT NULL, -- SALE, PAYMENT, TRANSFER, etc.
    payload JSONB NOT NULL,
    status VARCHAR(20) DEFAULT 'PENDING', -- PENDING, COMPLETED, FAILED, COMPENSATED
    created_at TIMESTAMPTZ DEFAULT NOW(),
    completed_at TIMESTAMPTZ,
    error_message TEXT,
    retry_count INTEGER DEFAULT 0
);
```

B. Transaction Wrapper

```python
# backend/app/services/transaction_service.py
class TransactionService:
    async def execute_with_recovery(
        self,
        transaction_type: str,
        payload: dict,
        execute_fn: Callable,
        compensate_fn: Callable
    ) -> Any:
        supabase = get_supabase()

        # Log transaction start
        log_result = supabase.table("transaction_log").insert({
            "transaction_type": transaction_type,
            "payload": payload,
            "status": "PENDING"
        }).execute()
        log_id = log_result.data[0]["id"]

        try:
            result = await execute_fn(payload)

            # Mark completed
            supabase.table("transaction_log").update({
                "status": "COMPLETED",
                "completed_at": datetime.utcnow().isoformat()
            }).eq("id", log_id).execute()

            return result

        except Exception as e:
            # Mark failed
            supabase.table("transaction_log").update({
                "status": "FAILED",
                "error_message": str(e)
```

```
            }).eq("id", log_id).execute()

            # Attempt compensation
            try:
                await compensate_fn(payload)
                supabase.table("transaction_log").update({
                    "status": "COMPENSATED"
                }).eq("id", log_id).execute()
            except Exception as comp_error:
                logger.critical(f"Compensation failed for {log_id}: {comp_error}")

            raise
```

C. Recovery Worker

```
async def recover_pending_transactions():
    """Run on startup and every minute"""
    supabase = get_supabase()

    # Find PENDING transactions older than 2 minutes
    pending = supabase.table("transaction_log").select("*").eq(
        "status", "PENDING"
    ).lt("created_at", datetime.utcnow() - timedelta(minutes=2)).execute()

    for tx in pending.data:
        if tx["retry_count"] < 3:
            # Retry
            await retry_transaction(tx)
        else:
            # Flag for manual intervention
            await flag_for_manual_review(tx)
```

C4. Correlation IDs - Request Tracing

Severity: 🔴 CRITICAL
Effort: 1 day
Risk if Unfixed: Cannot debug production issues, impossible to trace transactions

Files Affected

- `backend/app/main.py`
- `backend/app/utils/logger.py`
- All routers

Solution

A. Middleware for Correlation ID

```
# backend/app/middleware/correlation.py
import uuid
```

```python
from contextvars import ContextVar
from starlette.middleware.base import BaseHTTPMiddleware

correlation_id: ContextVar[str] = ContextVar('correlation_id', default='')

class CorrelationMiddleware(BaseHTTPMiddleware):
    async def dispatch(self, request, call_next):
        # Get from header or generate
        corr_id = request.headers.get('X-Correlation-ID', str(uuid.uuid4()))
        correlation_id.set(corr_id)

        response = await call_next(request)
        response.headers['X-Correlation-ID'] = corr_id

        return response
```

B. Update Logger

```python
# backend/app/utils/logger.py
import logging
from app.middleware.correlation import correlation_id

class CorrelationFilter(logging.Filter):
    def filter(self, record):
        record.correlation_id = correlation_id.get('')
        return True

def setup_logging():
    formatter = logging.Formatter(
        fmt='%(asctime)s [%(correlation_id)s] %(name)s %(levelname)s - %(message)s',
        datefmt='%Y-%m-%d %H:%M:%S'
    )
    # ... rest of setup
    root_logger.addFilter(CorrelationFilter())
```

C. Register Middleware

```python
# In main.py
from app.middleware.correlation import CorrelationMiddleware
app.add_middleware(CorrelationMiddleware)
```

C5. Error Alerting - Sentry Integration

Severity: 🔴 CRITICAL
Effort: 1 day
Risk if Unfixed: Production errors go unnoticed until customer complaints

Solution

A. Install Sentry

```
pip install sentry-sdk[fastapi]
```

B. Configure

```
# backend/app/main.py
import sentry_sdk
from sentry_sdk.integrations.fastapi import FastApiIntegration

sentry_sdk.init(
    dsn=settings.SENTRY_DSN,
    environment=settings.ENVIRONMENT,
    integrations=[FastApiIntegration()],
    traces_sample_rate=0.1,  # 10% of transactions
    profiles_sample_rate=0.1,
)

# Add to exception handler
@app.exception_handler(Exception)
async def general_exception_handler(request: Request, exc: Exception):
    sentry_sdk.capture_exception(exc)
    # ... existing handling
```

C. Environment Variable

```
SENTRY_DSN=https://xxx@sentry.io/xxx
```

🟡 HIGH SEVERITY (Fix Within 2 Weeks)

H1. N+1 Query Pattern in Ledger Views

Severity: 🟡 HIGH
Effort: 0.5 days
Risk if Unfixed: Slow performance with many suppliers/customers

Files Affected

- `backend/app/routers/poultry_retail/ledger.py`

Solution

```
@router.get("/suppliers")
async def get_supplier_ledger(...):
    supabase = get_supabase()

    # Single aggregated query instead of N+1
```

```
    result = supabase.rpc("get_supplier_balances", {
        "p_limit": limit,
        "p_offset": offset
    }).execute()

    return result.data
```

```
CREATE OR REPLACE FUNCTION get_supplier_balances(p_limit INT, p_offset INT)
RETURNS TABLE (...) AS $$
    SELECT
        s.id as entity_id,
        s.name as entity_name,
        s.phone,
        COALESCE(SUM(fl.debit), 0) as total_debit,
        COALESCE(SUM(fl.credit), 0) as total_credit,
        COALESCE(SUM(fl.credit), 0) - COALESCE(SUM(fl.debit), 0) as outstanding
    FROM suppliers s
    LEFT JOIN financial_ledger fl ON fl.entity_id = s.id::text
        AND fl.entity_type = 'SUPPLIER'
    WHERE s.status = 'ACTIVE'
    GROUP BY s.id, s.name, s.phone
    ORDER BY outstanding DESC
    LIMIT p_limit OFFSET p_offset;
$$ LANGUAGE sql;
```

H2. Mobile POS Offline Support

Severity: 🟡 HIGH
Effort: 3-5 days
Risk if Unfixed: Failed sales when network drops, lost revenue

Files Affected

- `expo-pos/lib/usePOS.ts`
- `expo-pos/lib/api.ts`

Solution

```
// expo-pos/lib/useOfflineQueue.ts
import AsyncStorage from '@react-native-async-storage/async-storage';

interface QueuedSale {
    id: string;
    data: SaleCreate;
    timestamp: number;
    retryCount: number;
}

export function useOfflineQueue() {
    const [queue, setQueue] = useState<QueuedSale[]>([]);
```

```
    const [isOnline, setIsOnline] = useState(true);

    // Queue sale when offline
    const queueSale = async (sale: SaleCreate) => {
        const queuedSale: QueuedSale = {
            id: uuid.v4(),
            data: sale,
            timestamp: Date.now(),
            retryCount: 0
        };

        const existing = await AsyncStorage.getItem('saleQueue');
        const queue = existing ? JSON.parse(existing) : [];
        queue.push(queuedSale);
        await AsyncStorage.setItem('saleQueue', JSON.stringify(queue));

        return queuedSale.id;
    };

    // Sync when back online
    const syncQueue = async () => {
        const stored = await AsyncStorage.getItem('saleQueue');
        if (!stored) return;

        const queue: QueuedSale[] = JSON.parse(stored);
        const remaining: QueuedSale[] = [];

        for (const item of queue) {
            try {
                await api.createSale({
                    ...item.data,
                    idempotency_key: item.id  // Use queue ID as idempotency key
                });
            } catch (error) {
                if (item.retryCount < 3) {
                    remaining.push({...item, retryCount: item.retryCount + 1});
                } else {
                    // Flag for manual review
                    await flagFailedSale(item);
                }
            }
        }

        await AsyncStorage.setItem('saleQueue', JSON.stringify(remaining));
    };

    return { queueSale, syncQueue, isOnline, queueLength: queue.length };
}
```

H3. Redis-Backed Rate Limiting

Severity: 🟡 HIGH
Effort: 1 day
Risk if Unfixed: Rate limits lost on restart, doesn't scale horizontally

Solution

```
# backend/app/middleware/rate_limiter.py
import redis.asyncio as redis

class RedisRateLimitStore:
    def __init__(self):
        self.redis = redis.from_url(settings.REDIS_URL)

    async def record_request(self, user_id: str) -> None:
        pipe = self.redis.pipeline()

        minute_key = f"rate:{user_id}:minute:{int(time.time() // 60)}"
        hour_key = f"rate:{user_id}:hour:{int(time.time() // 3600)}"

        pipe.incr(minute_key)
        pipe.expire(minute_key, 120)  # 2 minute TTL
        pipe.incr(hour_key)
        pipe.expire(hour_key, 7200)  # 2 hour TTL

        await pipe.execute()

    async def get_request_counts(self, user_id: str) -> Tuple[int, int]:
        minute_key = f"rate:{user_id}:minute:{int(time.time() // 60)}"
        hour_key = f"rate:{user_id}:hour:{int(time.time() // 3600)}"

        minute_count = await self.redis.get(minute_key) or 0
        hour_count = await self.redis.get(hour_key) or 0

        return int(minute_count), int(hour_count)
```

H4. Settlement Optimistic Locking

Severity: 🟡 HIGH
Effort: 1 day
Risk if Unfixed: Double-approval of settlements possible

Solution

A. Add Version Column

```
ALTER TABLE daily_settlements ADD COLUMN version INTEGER DEFAULT 1;
```

B. Update Approval Logic

```python
@router.post("/{settlement_id}/approve")
async def approve_settlement(settlement_id: UUID, ...):
    supabase = get_supabase()

    # Get current version
    existing = supabase.table("daily_settlements").select(
        "*, version"
    ).eq("id", str(settlement_id)).execute()

    current_version = existing.data[0]["version"]

    # Update with version check
    result = supabase.table("daily_settlements").update({
        "status": "APPROVED",
        "approved_by": current_user["id"],
        "version": current_version + 1
    }).eq("id", str(settlement_id)).eq(
        "version", current_version  # Only succeeds if version unchanged
    ).execute()

    if not result.data:
        raise HTTPException(
            status_code=409,
            detail="Settlement was modified by another user. Please refresh."
        )
```

H5. Inventory Balance Snapshots

Severity: 🟡 HIGH
Effort: 2 days
Risk if Unfixed: Stock queries become slow as ledger grows

Solution

A. Snapshot Table

```sql
CREATE TABLE inventory_balance_snapshots (
    id SERIAL PRIMARY KEY,
    store_id INTEGER NOT NULL,
    bird_type bird_type_enum NOT NULL,
    inventory_type inventory_type_enum NOT NULL,
    balance DECIMAL(10,3) NOT NULL,
    bird_count INTEGER DEFAULT 0,
    snapshot_date DATE NOT NULL,
    last_ledger_id UUID REFERENCES inventory_ledger(id),
    created_at TIMESTAMPTZ DEFAULT NOW(),
    UNIQUE(store_id, bird_type, inventory_type, snapshot_date)
);
```

B. Nightly Snapshot Job

```python
async def create_daily_snapshots():
    """Run at midnight for each store"""
    supabase = get_supabase()

    stores = supabase.table("shops").select("id").eq("status", "ACTIVE").execute()

    for store in stores.data:
        # Get current balances
        balances = supabase.rpc("get_current_stock", {
            "p_store_id": store["id"]
        }).execute()

        for balance in balances.data:
            supabase.table("inventory_balance_snapshots").upsert({
                "store_id": store["id"],
                "bird_type": balance["bird_type"],
                "inventory_type": balance["inventory_type"],
                "balance": balance["current_qty"],
                "snapshot_date": date.today().isoformat()
            }).execute()
```

C. Optimized Stock Query

```sql
CREATE OR REPLACE FUNCTION get_current_stock_fast(p_store_id INTEGER, ...)
RETURNS TABLE (...) AS $$
DECLARE
    v_snapshot_date DATE;
    v_snapshot_id UUID;
BEGIN
    -- Find most recent snapshot
    SELECT snapshot_date, last_ledger_id
    INTO v_snapshot_date, v_snapshot_id
    FROM inventory_balance_snapshots
    WHERE store_id = p_store_id
    ORDER BY snapshot_date DESC LIMIT 1;

    IF v_snapshot_date IS NULL THEN
        -- No snapshot, use full aggregation
        RETURN QUERY SELECT * FROM get_current_stock(p_store_id);
    ELSE
        -- Snapshot + delta
        RETURN QUERY
        SELECT
            s.store_id,
            s.bird_type,
            s.inventory_type,
            s.balance + COALESCE(SUM(il.quantity_change), 0) as current_qty
        FROM inventory_balance_snapshots s
```

```
        LEFT JOIN inventory_ledger il ON
            il.store_id = s.store_id
            AND il.bird_type = s.bird_type
            AND il.inventory_type = s.inventory_type
            AND il.created_at > v_snapshot_date
        WHERE s.store_id = p_store_id
        GROUP BY s.store_id, s.bird_type, s.inventory_type, s.balance;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

🟢 MEDIUM SEVERITY (Fix Before Scale)

M1. Receipt Number Gap Prevention

Effort: 1 day

Use sequence inside atomic function (already covered in C1 solution).

M2. Bulk Sale Permission Check

Effort: 0.5 days

```
# In sales.py
if sale.sale_type == SaleType.BULK:
    if "sales.bulk" not in current_user.get("permissions", []):
        raise HTTPException(
            status_code=403,
            detail="Bulk sales require 'sales.bulk' permission"
        )
```

M3. Store Access Check Before Fetch

Effort: 1 day

```
# Pattern to use everywhere
@router.get("/{sale_id}")
async def get_sale(sale_id: UUID, ...):
    # First: Minimal fetch with store_id only
    store_check = supabase.table("sales").select("store_id").eq(
        "id", str(sale_id)
    ).execute()

    if not store_check.data:
        raise HTTPException(status_code=404, detail="Sale not found")

    # Second: Validate access
```

```
    if not validate_store_access(store_check.data[0]["store_id"], current_user):
        raise HTTPException(status_code=403, detail="Access denied")

    # Third: Full fetch only after access confirmed
    result = supabase.table("sales").select("*").eq("id", str(sale_id)).execute()
```

M4. Device Fingerprinting for Mobile POS

Effort: 2-3 days

```
// expo-pos/lib/useDeviceAuth.ts
import * as Device from 'expo-device';
import * as Crypto from 'expo-crypto';

export async function getDeviceFingerprint(): Promise<string> {
    const deviceInfo = {
        brand: Device.brand,
        modelName: Device.modelName,
        osName: Device.osName,
        osVersion: Device.osVersion,
        deviceId: Device.deviceId,  // Unique device ID
    };

    const fingerprint = await Crypto.digestStringAsync(
        Crypto.CryptoDigestAlgorithm.SHA256,
        JSON.stringify(deviceInfo)
    );

    return fingerprint;
}

// Send with every request
api.defaults.headers['X-Device-Fingerprint'] = await getDeviceFingerprint();
```

M5. Prometheus Metrics Integration

Effort: 2 days

```
# backend/app/middleware/metrics.py
from prometheus_client import Counter, Histogram, generate_latest
from starlette.middleware.base import BaseHTTPMiddleware

REQUEST_COUNT = Counter(
    'http_requests_total',
    'Total HTTP requests',
    ['method', 'endpoint', 'status']
)

REQUEST_LATENCY = Histogram(
```

```
    'http_request_duration_seconds',
    'HTTP request latency',
    ['method', 'endpoint']
)

SALE_COUNTER = Counter(
    'sales_total',
    'Total sales created',
    ['store_id', 'payment_method']
)

INVENTORY_GAUGE = Gauge(
    'inventory_balance_kg',
    'Current inventory balance',
    ['store_id', 'bird_type', 'inventory_type']
)

class MetricsMiddleware(BaseHTTPMiddleware):
    async def dispatch(self, request, call_next):
        start_time = time.time()
        response = await call_next(request)
        duration = time.time() - start_time

        REQUEST_COUNT.labels(
            method=request.method,
            endpoint=request.url.path,
            status=response.status_code
        ).inc()

        REQUEST_LATENCY.labels(
            method=request.method,
            endpoint=request.url.path
        ).observe(duration)

        return response

@app.get("/metrics")
async def metrics():
    return Response(generate_latest(), media_type="text/plain")
```

Implementation Checklist

Week 1 (Critical)

- [ ] C1: Atomic sale creation RPC
- [ ] C4: Correlation IDs
- [ ] C5: Sentry integration

Week 2 (Critical + High)

- [ ] C2: Atomic functions for payments/transfers
- [ ] C3: Transaction log + recovery worker
- [ ] H1: Fix N+1 queries

Week 3 (High)

- [ ] H3: Redis rate limiting
- [ ] H4: Optimistic locking
- [ ] H2: Begin mobile offline support

Week 4 (High + Medium)

- [ ] H5: Balance snapshots
- [ ] H2: Complete mobile offline
- [ ] M2: Bulk sale permission
- [ ] M3: Access check pattern

Week 5+ (Medium)

- [ ] M4: Device fingerprinting
- [ ] M5: Prometheus metrics
- [ ] M1: Receipt number refinement

Document Version: 1.0
Last Updated: 2026-01-17