**Mingus Production Integration Plan**

**Database Schema Optimization**

**Recommended daily\_cashflow Table Structure**

Based on your existing naming convention (user\_income\_due\_dates, user\_expense\_due\_dates), here's the optimized structure:

-- Optimized daily\_cashflow table

CREATE TABLE daily\_cashflow (

id uuid PRIMARY KEY DEFAULT gen\_random\_uuid(),

user\_id uuid REFERENCES auth.users(id) NOT NULL,

forecast\_date date NOT NULL,

opening\_balance decimal(12,2) NOT NULL,

total\_income decimal(12,2) DEFAULT 0.00,

total\_expenses decimal(12,2) DEFAULT 0.00,

closing\_balance decimal(12,2) NOT NULL,

net\_daily\_change decimal(12,2) NOT NULL,

balance\_status text CHECK (balance\_status IN ('positive', 'negative', 'critical')),

created\_at timestamp DEFAULT now(),

updated\_at timestamp DEFAULT now(),

UNIQUE(user\_id, forecast\_date)

);

-- Index for performance

CREATE INDEX idx\_daily\_cashflow\_user\_date ON daily\_cashflow(user\_id, forecast\_date);

CREATE INDEX idx\_daily\_cashflow\_status ON daily\_cashflow(user\_id, balance\_status);

**Column Name Rationale**

* **forecast\_date** instead of date (avoids SQL reserved word conflicts)
* **total\_income** and **total\_expenses** (clear aggregation purpose)
* **net\_daily\_change** instead of net\_change (more specific)
* **balance\_status** instead of status\_color (semantic meaning, color derived in frontend)
* **Decimal(12,2)** precision for financial accuracy up to $999,999,999.99

**Enhanced Schedule Generation System**

**Production-Ready Python Backend**

import os

import asyncio

from datetime import datetime, timedelta

from dateutil.rrule import rrule, WEEKLY, MONTHLY, YEARLY

from supabase import create\_client, Client

import logging

class MingusCashFlowEngine:

def \_\_init\_\_(self):

self.supabase = create\_client(

os.environ.get("SUPABASE\_URL"),

os.environ.get("SUPABASE\_SERVICE\_KEY") # Service key for server operations

)

def generate\_schedule\_enhanced(self, start\_date\_str, frequency, amount,

category, forecast\_months=12, business\_days\_only=False):

"""Enhanced schedule generation with business day logic"""

start\_date = datetime.strptime(start\_date\_str, "%Y-%m-%d")

end\_date = start\_date + timedelta(days=30 \* forecast\_months)

freq\_map = {

'weekly': rrule(WEEKLY, dtstart=start\_date, until=end\_date),

'bi-weekly': rrule(WEEKLY, interval=2, dtstart=start\_date, until=end\_date),

'monthly': rrule(MONTHLY, dtstart=start\_date, until=end\_date),

'quarterly': rrule(MONTHLY, interval=3, dtstart=start\_date, until=end\_date),

'annually': rrule(YEARLY, dtstart=start\_date, until=end\_date)

}

schedule = []

for dt in freq\_map[frequency]:

payment\_date = dt

# Business day adjustment

if business\_days\_only:

payment\_date = self.adjust\_to\_business\_day(payment\_date)

schedule.append({

'due\_date': payment\_date.strftime("%Y-%m-%d"),

'amount': amount,

'category': category,

'frequency': frequency

})

return schedule

def adjust\_to\_business\_day(self, date):

"""Move weekend dates to next Monday, handle holidays later"""

while date.weekday() > 4: # 5=Saturday, 6=Sunday

date += timedelta(days=1)

return date

async def populate\_income\_schedules(self, user\_id):

"""Populate user\_income\_due\_dates table"""

try:

# Get user's income data

income\_response = self.supabase.table('user\_income')\

.select('\*')\

.eq('user\_id', user\_id)\

.execute()

# Clear existing future dates

future\_date = datetime.now().strftime("%Y-%m-%d")

self.supabase.table('user\_income\_due\_dates')\

.delete()\

.eq('user\_id', user\_id)\

.gte('due\_date', future\_date)\

.execute()

# Generate new schedules

for income in income\_response.data:

schedule = self.generate\_schedule\_enhanced(

income.get('start\_date', datetime.now().strftime("%Y-%m-%d")),

income['frequency'],

income['amount'],

'income',

forecast\_months=12,

business\_days\_only=True # Paychecks on business days

)

# Insert new dates

for entry in schedule:

self.supabase.table('user\_income\_due\_dates').insert({

'user\_id': user\_id,

'due\_date': entry['due\_date'],

'amount': entry['amount'],

'frequency': entry['frequency'],

'category': 'income'

}).execute()

return True

except Exception as e:

logging.error(f"Error populating income schedules for user {user\_id}: {e}")

return False

async def populate\_expense\_schedules(self, user\_id):

"""Populate user\_expense\_due\_dates table"""

try:

# Get all expense categories for user

expense\_tables = [

'rent\_or\_mortgage\_expense', 'auto\_gas\_expense', 'auto\_insurance\_expense',

'health\_insurance\_expense', 'mobile\_phone\_expense', 'daycare\_expense',

'entertainment\_expense', 'restaurant\_meals\_expense', 'gift\_others\_expense',

'selfcare\_expense', 'pet\_expense', 'child\_support\_expense', 'career\_expense',

'personal\_care\_expense', 'healthcare\_expense', 'fraternity\_sorority\_expense',

'credit\_card1\_expense', 'credit\_card2\_expense', 'credit\_card3\_expense',

'loan\_1\_expense', 'loan\_2\_expense', 'loan\_3\_expense', 'rideshare\_expense',

'auto\_payment\_expense', 'cash\_to\_family\_expense', 'bus\_fare\_expense'

]

# Clear existing future dates

future\_date = datetime.now().strftime("%Y-%m-%d")

self.supabase.table('user\_expense\_due\_dates')\

.delete()\

.eq('user\_id', user\_id)\

.gte('due\_date', future\_date)\

.execute()

for table\_name in expense\_tables:

try:

expense\_response = self.supabase.table(table\_name)\

.select('\*')\

.eq('user\_id', user\_id)\

.execute()

for expense in expense\_response.data:

if expense.get('amount', 0) > 0: # Only process if amount exists

category = table\_name.replace('\_expense', '')

schedule = self.generate\_schedule\_enhanced(

expense.get('due\_date', datetime.now().strftime("%Y-%m-%d")),

expense.get('frequency', 'monthly'),

-abs(expense['amount']), # Expenses are negative

category,

forecast\_months=12,

business\_days\_only=False # Bills can be any day

)

# Insert new dates

for entry in schedule:

self.supabase.table('user\_expense\_due\_dates').insert({

'user\_id': user\_id,

'due\_date': entry['due\_date'],

'amount': entry['amount'],

'frequency': entry['frequency'],

'category': category

}).execute()

except Exception as table\_error:

logging.warning(f"Could not process {table\_name}: {table\_error}")

continue

return True

except Exception as e:

logging.error(f"Error populating expense schedules for user {user\_id}: {e}")

return False

async def calculate\_daily\_cashflow(self, user\_id, days\_ahead=365):

"""Calculate daily cash flow for the next year"""

try:

# Get opening balance

user\_response = self.supabase.table('users')\

.select('opening\_balance')\

.eq('id', user\_id)\

.execute()

if not user\_response.data:

logging.error(f"User {user\_id} not found")

return False

opening\_balance = user\_response.data[0].get('opening\_balance', 0)

# Get date range

start\_date = datetime.now()

end\_date = start\_date + timedelta(days=days\_ahead)

# Get all income dates

income\_response = self.supabase.table('user\_income\_due\_dates')\

.select('\*')\

.eq('user\_id', user\_id)\

.gte('due\_date', start\_date.strftime("%Y-%m-%d"))\

.lte('due\_date', end\_date.strftime("%Y-%m-%d"))\

.execute()

# Get all expense dates

expense\_response = self.supabase.table('user\_expense\_due\_dates')\

.select('\*')\

.eq('user\_id', user\_id)\

.gte('due\_date', start\_date.strftime("%Y-%m-%d"))\

.lte('due\_date', end\_date.strftime("%Y-%m-%d"))\

.execute()

# Build daily transaction map

daily\_transactions = {}

# Add income

for income in income\_response.data:

date\_key = income['due\_date']

if date\_key not in daily\_transactions:

daily\_transactions[date\_key] = {'income': 0, 'expenses': 0}

daily\_transactions[date\_key]['income'] += income['amount']

# Add expenses

for expense in expense\_response.data:

date\_key = expense['due\_date']

if date\_key not in daily\_transactions:

daily\_transactions[date\_key] = {'income': 0, 'expenses': 0}

daily\_transactions[date\_key]['expenses'] += abs(expense['amount']) # Store as positive

# Calculate daily balances

current\_balance = opening\_balance

current\_date = start\_date

daily\_records = []

# Clear existing future forecasts

self.supabase.table('daily\_cashflow')\

.delete()\

.eq('user\_id', user\_id)\

.gte('forecast\_date', start\_date.strftime("%Y-%m-%d"))\

.execute()

while current\_date <= end\_date:

date\_str = current\_date.strftime("%Y-%m-%d")

# Get transactions for this date

day\_income = daily\_transactions.get(date\_str, {}).get('income', 0)

day\_expenses = daily\_transactions.get(date\_str, {}).get('expenses', 0)

net\_change = day\_income - day\_expenses

# Calculate new balance

new\_balance = current\_balance + net\_change

# Determine status

if new\_balance < 0:

status = 'critical'

elif new\_balance < 100: # Configurable threshold

status = 'negative'

else:

status = 'positive'

daily\_record = {

'user\_id': user\_id,

'forecast\_date': date\_str,

'opening\_balance': current\_balance,

'total\_income': day\_income,

'total\_expenses': day\_expenses,

'closing\_balance': new\_balance,

'net\_daily\_change': net\_change,

'balance\_status': status

}

daily\_records.append(daily\_record)

current\_balance = new\_balance

current\_date += timedelta(days=1)

# Batch insert daily records

if daily\_records:

self.supabase.table('daily\_cashflow').insert(daily\_records).execute()

return True

except Exception as e:

logging.error(f"Error calculating daily cashflow for user {user\_id}: {e}")

return False

async def refresh\_user\_forecast(self, user\_id):

"""Complete forecast refresh for a user"""

success = True

success &= await self.populate\_income\_schedules(user\_id)

success &= await self.populate\_expense\_schedules(user\_id)

success &= await self.calculate\_daily\_cashflow(user\_id)

return success

# Usage example

async def main():

engine = MingusCashFlowEngine()

user\_id = "your-user-uuid-here"

success = await engine.refresh\_user\_forecast(user\_id)

print(f"Forecast refresh {'successful' if success else 'failed'}")

if \_\_name\_\_ == "\_\_main\_\_":

asyncio.run(main())

**Automated Daily Processing**

**Recommendation: Supabase Edge Functions + Cron**

For automatic daily updates, I recommend using **Supabase Edge Functions** with cron triggers:

// Supabase Edge Function: refresh-forecasts

import { serve } from "https://deno.land/std@0.168.0/http/server.ts"

import { createClient } from 'https://esm.sh/@supabase/supabase-js@2'

serve(async (req) => {

try {

const supabase = createClient(

Deno.env.get('SUPABASE\_URL') ?? '',

Deno.env.get('SUPABASE\_SERVICE\_ROLE\_KEY') ?? ''

)

// Get all active users (or users who need updates)

const { data: users } = await supabase

.from('users')

.select('id')

.eq('active', true)

let successCount = 0

let errorCount = 0

for (const user of users || []) {

try {

// Call your Python function (deployed as separate service)

const response = await fetch('YOUR\_PYTHON\_SERVICE\_URL/refresh-forecast', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ user\_id: user.id })

})

if (response.ok) {

successCount++

} else {

errorCount++

}

} catch (error) {

errorCount++

console.error(`Error refreshing forecast for user ${user.id}:`, error)

}

}

return new Response(

JSON.stringify({

success: true,

processed: successCount,

errors: errorCount

}),

{ headers: { "Content-Type": "application/json" } }

)

} catch (error) {

return new Response(

JSON.stringify({ success: false, error: error.message }),

{ headers: { "Content-Type": "application/json" }, status: 500 }

)

}

})

**Alternative: GitHub Actions (Recommended for Budget)**

# .github/workflows/daily-forecast.yml

name: Daily Forecast Update

on:

schedule:

- cron: '0 6 \* \* \*' # Run at 6 AM UTC daily

workflow\_dispatch: # Allow manual trigger

jobs:

update-forecasts:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v3

- uses: actions/setup-python@v4

with:

python-version: '3.9'

- name: Install dependencies

run: |

pip install supabase python-dateutil

- name: Run forecast updates

env:

SUPABASE\_URL: ${{ secrets.SUPABASE\_URL }}

SUPABASE\_SERVICE\_KEY: ${{ secrets.SUPABASE\_SERVICE\_KEY }}

run: python scripts/daily\_forecast\_update.py

**Implementation Timeline**

**Week 1: Database & Core Logic**

* **You Build**: Update daily\_cashflow table structure, test basic Python functions
* **Upwork ($400)**: Database migration, optimization, error handling

**Week 2: Integration & Testing**

* **You Build**: Connect Python functions to existing Supabase data
* **Upwork ($300)**: Advanced error handling, performance optimization

**Week 3: Automation Setup**

* **You Build**: GitHub Actions workflow, testing automation
* **Upwork ($300)**: Production deployment setup, monitoring

**Week 4: Frontend Connection**

* **You Build**: Vanilla JS dashboard connecting to daily\_cashflow table
* **Upwork ($500)**: Advanced visualizations, responsive design

**Budget Allocation ($5,000)**

* **Backend Integration**: $1,500 (Python + Supabase optimization)
* **Frontend Development**: $1,200 (Dashboard, charts, mobile-responsive)
* **Infrastructure**: $800 (hosting, automation, monitoring)
* **Health Integration**: $800 (correlation algorithms, health check-ins)
* **Contingency**: $700 (unexpected issues, additional features)

**Key Advantages of This Approach**

**1. Scalable Architecture**

* Handles thousands of users with efficient batch processing
* Separate concerns: schedule generation → daily calculation → frontend display

**2. Reliable Automation**

* GitHub Actions is free and reliable
* Easy monitoring and error handling
* Manual trigger capability for testing

**3. Performance Optimized**

* Pre-calculated daily balances for instant dashboard loading
* Indexed database queries for fast user experience
* Batch operations minimize database load

The combination of your existing schema + this production-ready backend puts you **2-3 months ahead** of typical personal finance app development timelines!