Recommendations for Telegram Bot Project Issues

State Management Improvements

Current Issues:

Recommendations:

```
// Enhance session middleware for persistence
const enhancedSessionMiddleware = (ctx, next) => {
  // Save session before scene transitions
  const originalEnter = ctx.scene.enter;
  ctx.scene.enter = async (sceneId, ...rest) => {
    await ctx.session.save(); // Force save before transition
    return originalEnter.call(ctx.scene, sceneId, ...rest);
  };

  // Add session recovery mechanism
  if (!ctx.session || Object.keys(ctx.session).length === 0) {
    console.log(`Recovering session for user ${ctx.from.id}`);
    ctx.session = sessionStore.getBackup(ctx.from.id) || {};
  }

  return next();
};

bot.use(enhancedSessionMiddleware);
```

2. Improve scene management:

```
// In campaignCreation.js
const campaignScene = new Scenes.WizardScene('campaignCreation',
  // Step 1: Campaign details
  async (ctx) => {
    try {
      // Store critical state in both session and scene state
      ctx.scene.state.campaignData = ctx.scene.state.campaignData || {};
      ctx.session.campaignData = ctx.scene.state.campaignData;
      // Add checkpoint for verification
      ctx.scene.state.checkpoint = 'details';
      await ctx.reply('Enter campaign name:');
      return ctx.wizard.next();
    } catch (error) {
      console.error('Campaign scene step 1 error:', error);
      await ctx.reply('An error occurred. Please try /start again.');
      return ctx.scene.leave();
    }
  },
  // Additional steps...
);

// Add leave handler to clean up
campaignScene.leave((ctx) => {
  // Clean up scene state but preserve important data
  if (ctx.scene.state.completed) {
    ctx.session.lastCompletedCampaign = ctx.scene.state.campaignData;
  }
  delete ctx.scene.state;
});

// Add middleware to validate progression
campaignScene.use((ctx, next) => {
  const step = ctx.wizard.cursor;
  const requiredFields = getRequiredFieldsForStep(step);

  if (step > 0 && !validateRequiredFields(ctx.scene.state.campaignData, requiredFields)) {
    return ctx.reply('Please complete all required fields before continuing.');
  }

  return next();
});
```

3. Create state recovery mechanism:

```
// Add to bot initialization
const stateRecoveryMiddleware = (ctx, next) => {
  // Add recovery command handler
  if (ctx.message && ctx.message.text === '/recover') {
    const backupState = stateManager.getLastValidState(ctx.from.id);
    if (backupState) {
      ctx.session = {...ctx.session, ...backupState};
      return ctx.reply('Your previous session has been recovered.');
    }
    return ctx.reply('No previous session found to recover.');
  }

  // Create periodic state backups
  if (ctx.session) {
    stateManager.createBackup(ctx.from.id, ctx.session);
  }

  return next();
};

bot.use(stateRecoveryMiddleware);
```

Database Operation Safety

Current Issues:

Recommendations:

```
// In campaignHandlers.js
async function updateCampaignParticipants(campaignId, userId, action) {
  const client = await pool.connect();
  try {
    await client.query('BEGIN');

    // Get current campaign with FOR UPDATE to lock the row
    const { rows } = await client.query(
      'SELECT * FROM campaigns WHERE id = $1 FOR UPDATE',
      [campaignId]
    );

    if (!rows.length) {
      throw new Error('Campaign not found');
    }

    const campaign = rows[0];
    let participants = [];

    // Safely parse participants array
    try {
      participants = campaign.participants ? JSON.parse(campaign.participants) : [];
    } catch (e) {
      console.error('Error parsing participants JSON:', e);
      participants = [];
    }

    // Perform action
    if (action === 'add' && !participants.includes(userId)) {
      participants.push(userId);
    } else if (action === 'remove') {
      participants = participants.filter(id => id !== userId);
    }

    // Update with new participants array
    await client.query(
      'UPDATE campaigns SET participants = $1 WHERE id = $2',
      [JSON.stringify(participants), campaignId]
    );

    await client.query('COMMIT');
    return true;
  } catch (error) {
    await client.query('ROLLBACK');
    console.error('Error updating campaign participants:', error);
    throw error;
  } finally {
    client.release();
  }
}
```

2. Improve connection pool management:

```
// In setup-db.js
const { Pool } = require('pg');

let poolConfig = {
  connectionString: process.env.DATABASE_URL,
  max: 20,
  idleTimeoutMillis: 30000,
  connectionTimeoutMillis: 5000,
};

const pool = new Pool(poolConfig);

// Add event listeners for connection issues
pool.on('error', (err, client) => {
  console.error('Unexpected error on idle database client', err);
});

// Add health check function
const checkDatabaseConnection = async () => {
  let client;
  try {
    client = await pool.connect();
    await client.query('SELECT 1');
    return true;
  } catch (error) {
    console.error('Database connection check failed:', error);
    // Attempt reconnection by recreating the pool
    if (pool.totalCount === 0) {
      console.log('Attempting to recreate connection pool...');
      pool.end().catch(console.error);
      pool = new Pool(poolConfig);
    }
    return false;
  } finally {
    if (client) client.release();
  }
};

// Run periodic health checks
setInterval(checkDatabaseConnection, 60000);

module.exports = { pool, checkDatabaseConnection };
```

3. Implement safe JSON handling:

```
// Add utility functions for safe JSON operations
const safeJsonParse = (jsonString, defaultValue = {}) => {
  try {
    return jsonString ? JSON.parse(jsonString) : defaultValue;
  } catch (error) {
    console.error('JSON parse error:', error);
    return defaultValue;
  }
};

const safeJsonStringify = (object, defaultValue = '{}') => {
  try {
    return JSON.stringify(object || {});
  } catch (error) {
    console.error('JSON stringify error:', error);
    return defaultValue;
  }
};

// Use in models
// In Campaign.js
async function updateCampaignStats(campaignId, newStats) {
  try {
    const { rows } = await pool.query('SELECT stats FROM campaigns WHERE id = $1',
[campaignId]);
    if (!rows.length) return false;

    // Safely merge stats
    const existingStats = safeJsonParse(rows[0].stats, {});
    const mergedStats = {...existingStats, ...newStats};

    await pool.query(
      'UPDATE campaigns SET stats = $1 WHERE id = $2',
      [safeJsonStringify(mergedStats), campaignId]
    );
    return true;
  } catch (error) {
    console.error('Error updating campaign stats:', error);
    throw error;
  }
}
```

Error Handling Robustness

Current Issues:

Recommendations:

```
// Add to index.js
process.on('unhandledRejection', (reason, promise) => {
  console.error('Unhandled Rejection at:', promise, 'reason:', reason);
  // Optionally notify admin or send to error tracking system
});

// Add middleware for Telegram API errors
bot.catch((err, ctx) => {
  console.error(`Error in bot update ${ctx.update.update_id}:`, err);

  // Categorize errors
  if (err.code === 403) {
    console.log(`User ${ctx.from?.id} has blocked the bot`);
    return;
  }

  // For database errors
  if (err.message.includes('database') || err.message.includes('sql')) {
    ctx.reply('A database error occurred. Our team has been notified.');
    // Log to special channel/system
    return;
  }

  // Generic error message
  ctx.reply('An error occurred. Please try again or contact support with code: ' +
          ctx.update.update_id).catch(console.error);
});
```

2. Add try-catch blocks to all handlers:

```
// Example improved handler in basicHandlers.js
async function startHandler(ctx) {
  try {
    const user = await User.findByTelegramId(ctx.from.id);

    if (!user) {
      console.log(`New user ${ctx.from.id} started the bot`);
      await ctx.scene.enter('userRegistration');
      return;
    }
    // Send welcome message with error handling
    await ctx.reply(`Welcome back, ${user.name}!`, {
      reply_markup: getMainMenu()
    }).catch(error => {
      console.error('Error sending welcome message:', error);
      ctx.reply('Welcome back! (simplified message due to error)');
    });

  } catch (error) {
    console.error('Error in start handler:', error);
    // Try minimal response that's less likely to fail
    ctx.reply('Welcome! Type /help if you need assistance.').catch(console.error);

    // Log critical errors
    if (error.message.includes('ETIMEDOUT') || error.message.includes('connection')) {
      notifyAdmins(`Critical error in startHandler: ${error.message}`);
    }
  }
}
```

3. Create user-friendly error handlers for each scene:

```
// In xVerification.js
const xVerificationScene = new Scenes.WizardScene(
  'xVerification',
  // Step 1
  async (ctx) => {
    try {
      ctx.scene.state.attempts = 0;
      ctx.scene.state.error = null;
      await ctx.reply('Please enter your X (Twitter) username:');
      return ctx.wizard.next();
    } catch (error) {
      console.error('Error in X verification step 1:', error);
      await ctx.reply('Unable to start verification. Please try again later.');
      return ctx.scene.leave();
    }
  },
  // Step 2 with error handling
  async (ctx) => {
    try {
      const username = ctx.message?.text;
      if (!username || !username.match(/^[A-Za-z0-9_]{1,15}$/)) {
        await ctx.reply('Invalid username format. Please enter a valid X username:');
        return; // Stay on current step
      }

      ctx.scene.state.username = username;

      // Attempt API call with timeout
      const verificationResult = await Promise.race([
        verifyXAccount(username),
        new Promise((_, reject) =>
          setTimeout(() => reject(new Error('Verification timed out')), 10000)
        )
      ]);

      // Continue with successful verification
      ctx.scene.state.verificationCode = verificationResult.code;
      await ctx.reply(`Please post the following code on X: ${verificationResult.code}`);
      return ctx.wizard.next();

    } catch (error) {
      console.error('Error in X verification step 2:', error);

      // Handle specific errors
      if (error.message.includes('timed out')) {
        await ctx.reply('Verification is taking too long. Please try again later.');
      } else if (error.code === 429) {
        await ctx.reply('Too many verification attempts. Please try again in 15 minutes.');
      } else {
        await ctx.reply('Error during verification. Please try again.');
      }

      // Allow retry or exit based on attempt count
      ctx.scene.state.attempts = (ctx.scene.state.attempts || 0) + 1;
      if (ctx.scene.state.attempts >= 3) {
        await ctx.reply('Too many failed attempts. Please try again later.');
        return ctx.scene.leave();
      }

      return; // Stay on current step for retry
    }
  }
  // Additional steps...
);
```

User Experience Continuity

Current Issues:

Recommendations:

```
// Create a validation middleware factory
const createValidationMiddleware = (requiredFields, errorMessage) => {
  return (ctx, next) => {
    const data = ctx.scene?.state || {};
    const missing = requiredFields.filter(field => !data[field]);

    if (missing.length > 0) {
      return ctx.reply(`Please complete the following required fields: ${missing.join(', ')}`);
    }

    return next();
  };
};

// Apply to campaign creation scene
campaignScene.use(createValidationMiddleware(
  ['name', 'description', 'reward', 'duration'],
  'Please complete all required campaign details before proceeding.'
));
```

2. Improve permission checking:

```
// In middleware/auth.js
const projectOwnerMiddleware = async (ctx, next) => {
  try {
    // Skip for admins
    if (await isAdmin(ctx.from.id)) {
      ctx.state.isAdmin = true;
      return next();
    }

    // Get project ID from context
    const projectId = ctx.scene?.state?.projectId ||
                    ctx.session?.currentProject?.id ||
                ctx.callbackQuery?.data?.match(/project_(\d+)/)?.[1];

    if (!projectId) {
      return ctx.reply('No project specified. Please select a project first.');
    }

    // Check ownership with retries
    let attempts = 0;
    let isOwner = false;

    while (attempts < 3 && !isOwner) {
      try {
        const result = await Project.checkOwnership(projectId, ctx.from.id);
        isOwner = result;
        break;
      } catch (error) {
        console.error(`Ownership check failed (attempt ${attempts + 1}):`, error);
        attempts++;
        if (attempts < 3) await new Promise(r => setTimeout(r, 500)); // Wait before retry
      }
    }

    if (!isOwner) {
      return ctx.reply('You do not have permission to manage this project.');
    }

    // Store permission in context
    ctx.state.isProjectOwner = true;
    ctx.state.projectId = projectId;

    return next();
  } catch (error) {
    console.error('Error in project owner middleware:', error);
    return ctx.reply('Could not verify project ownership. Please try again.');
  }
};
```

3. Create better user feedback mechanisms:

```
// In utils/userInterface.js
const sendProgressUpdate = async (ctx, stage, total, message) => {
  try {
    const progressBar = Array(10).fill('○');
    const filledCount = Math.floor((stage / total) * 10);
    for (let i = 0; i < filledCount; i++) {
      progressBar[i] = '●';
    }

    const progressText = progressBar.join(' ');
    const percentComplete = Math.floor((stage / total) * 100);

    await ctx.reply(
      `${message}\n\nProgress: ${progressText} ${percentComplete}%\nStep ${stage} of ${total}`
    );
  } catch (error) {
    console.error('Error sending progress update:', error);
    // Fallback to simple message
    await ctx.reply(`${message} (Step ${stage} of ${total})`).catch(console.error);
  }
};

// Use in campaign creation
await sendProgressUpdate(
  ctx,
  ctx.wizard.cursor + 1,
  ctx.wizard.steps.length,
  'Please enter campaign description:'
);
```

Performance Optimization

Current Issues:

Recommendations:

```
// In utils/cache.js
const NodeCache = require('node-cache');

const cache = new NodeCache({
  stdTTL: 300, // 5 minutes
  checkperiod: 60 // Check for expired entries every minute
});

const getCachedData = async (key, fetchFunction, ttl = 300) => {
  const cachedData = cache.get(key);
  if (cachedData !== undefined) {
    return cachedData;
  }

  try {
    const freshData = await fetchFunction();
    cache.set(key, freshData, ttl);
    return freshData;
  } catch (error) {
    console.error(`Cache miss and fetch error for key ${key}:`, error);
    throw error;
  }
};

module.exports = { cache, getCachedData };

// Use in handlers
async function getProjectStats(projectId) {
  return getCachedData(
    `project_stats_${projectId}`,
    async () => {
      const stats = await Project.getDetailedStats(projectId);
      return stats;
    },
    60 // Cache for 1 minute
  );
}
```

2. Implement rate limiting for external APIs:

```
// In services/verification.js
const { RateLimiter } = require('limiter');

// Create rate limiter for Twitter API: 300 requests per 15 min window = 20 per minute
const twitterLimiter = new RateLimiter({
  tokensPerInterval: 20,
  interval: "minute"
});
```

```
async function verifyXAccount(username) {
  // Check if we have tokens available
  const remainingTokens = await twitterLimiter.removeTokens(1);

  if (remainingTokens < 0) {
    const retryAfter = Math.ceil(twitterLimiter.msToNextToken() / 1000);
    const error = new Error('Twitter API rate limit exceeded');
    error.retryAfter = retryAfter;
    error.code = 429;
    throw error;
  }

  try {
    // Proceed with API call now that we have a token
    const result = await twitterClient.verifyUser(username);
    return result;
  } catch (error) {
    // If Twitter returns a rate limit error, update our limiter
    if (error.code === 429) {
      const resetTime = parseInt(error.headers?.['x-rate-limit-reset'] || '0') * 1000;
      if (resetTime > 0) {
        const now = Date.now();
        const waitMs = Math.max(0, resetTime - now);
        twitterLimiter.tokenBucket.waitMs = waitMs;
      }
    }
    throw error;
  }
}
```

3. Implement background processing for non-critical tasks:

```
// In services/scheduler.js
const { Worker, isMainThread, parentPort, workerData } = require('worker_threads');
const Queue = require('better-queue');

// Create processing queue with concurrency control
const processingQueue = new Queue(async (task, cb) => {
  try {
    switch (task.type) {
      case 'updateStats':
        await updateCampaignStats(task.campaignId);
        break;
      case 'distributeRewards':
        await distributeRewards(task.campaignId);
        break;
      case 'syncEngagementMetrics':
        await syncEngagementMetrics(task.campaigns);
        break;
      default:
        throw new Error(`Unknown task type: ${task.type}`);
    }
    cb(null, { success: true });
  } catch (error) {
    console.error(`Error processing task ${task.type}:`, error);
    cb(error);
  }
}, {
  concurrent: 3,
  maxRetries: 3,
  retryDelay: 5000
});

// Add helper to schedule tasks
const scheduleTask = (type, data, priority = 0) => {
  return new Promise((resolve, reject) => {
    processingQueue.push({
      type,
      ...data,
      createdAt: Date.now()
    }, { priority }, (err, result) => {
      if (err) return reject(err);
      resolve(result);
    });
  });
};

// Use in handlers
async function completeCampaignAction(ctx) {
  try {
    // Handle immediate user feedback
    await ctx.reply('Action recorded! Processing your rewards...');

    // Schedule background processing
    await scheduleTask('distributeRewards', {
      campaignId: ctx.scene.state.campaignId,
      userId: ctx.from.id
    });

    // Schedule metrics update with lower priority
    await scheduleTask('syncEngagementMetrics', {
      campaigns: [ctx.scene.state.campaignId]
    }, 5);

    return ctx.scene.leave();
  } catch (error) {
    console.error('Error in campaign action:', error);
    return ctx.reply('There was an error processing your action. Please try again.');
  }
}
```

4. Optimize database queries:

```
// In Campaign.js
async function getCampaignsForUser(userId, options = {}) {
  const { limit = 10, offset = 0, status = 'active' } = options;

  // Create optimized query with proper indexing
  const query = `
    SELECT c.id, c.name, c.description, c.reward, c.created_at,
           p.name as project_name, p.id as project_id,
           (SELECT COUNT(*) FROM unnest(c.participants) as p WHERE p = $1) > 0 as
is_participant
    FROM campaigns c
    JOIN projects p ON c.project_id = p.id
    WHERE c.status = $2
      AND (c.is_public = true OR c.project_id IN (
        SELECT project_id FROM project_members WHERE user_id = $1
      ))
    ORDER BY c.created_at DESC
    LIMIT $3 OFFSET $4
  `;

  try {
    const { rows } = await pool.query(query, [userId, status, limit, offset]);

    // Transform results in JS rather than additional queries
    return rows.map(row => ({
      id: row.id,
      name: row.name,
      description: row.description,
      reward: row.reward,
      createdAt: row.created_at,
      project: {
        id: row.project_id,
        name: row.project_name
      },
      isParticipant: row.is_participant
    }));
  } catch (error) {
    console.error('Error getting campaigns for user:', error);
    throw error;
  }
}
```

These recommendations address the specific issues you've identified while maintaining your current codebase structure. Implementing these changes should significantly improve the stability, reliability, and user experience of your Telegram bot.