Verification and Validation Report: MES-ERP

Team #26, Ethical Pals
Sufyan Motala
Rachid Khneisser
Housam Alamour
Omar Muhammad
Taaha Atif

March 10, 2025

1 Revision History

Date	Version	Notes
Date 1	1.0	Notes
Date 2	1.1	Notes

2 Symbols, Abbreviations and Acronyms

symbol	description
T	Test

[symbols, abbreviations or acronyms – you can reference the SRS tables if needed —SS]

Contents

List of Tables

List of Figures

This document ...

3 Functional Requirements Evaluation

This section presents the detailed outcomes of functional testing for our six core functional tests (test-id1 through test-id6). We focus on (a) the inputs used, (b) observed vs. expected behavior, (c) issues discovered, and (d) improvements made to the code. All tests map directly to the requirements described in our SRS, and traceability is discussed in Section Trace To Modules.

4 Functional Requirements Evaluation

This section presents the outcomes and improvements from running Tests 1–6 exactly as specified in the VnV Plan. Each subsection briefly describes the real-world data or scenarios used, the observed system behavior, and any code changes made to align with the SRS.

4.1 Test 1: Submission Confirmation (test-id1)

Goal and Input: A user, already logged in, submits a valid reimbursement form *RB-2025-0002*. For instance, the form includes:

- **Expense Name:** Office Supplies
- **Date:** 2025-02-10
- **Amount:** $120.00 USD
- **Attached Receipt:** `receipt-office-supplies.pdf`

Expected Outcome: The system displays a "Submission Successful" message and updates the ledger with a corresponding entry.
Results and Analysis: We tested ten separate reimbursement submissions, each referencing distinct receipts. In nine submissions, the ledger reflected the new transaction instantly and displayed a clear confirmation message. However, one submission with a missing `budgetLine` field produced a generic error message, indicating partial fulfillment of the requirement but lacking robust error handling.

Code Improvements:

- Added front-end checks (React form validation) preventing the user from submitting the form if required fields (like `budgetLine`) are empty.
- Updated the `SubmitController` to return a descriptive error message ("Missing budget line") instead of the vague "400 Bad Request."

These changes improved usability by alerting users to incomplete inputs before reaching the back end, better aligning the submission workflow with SRS clarity requirements.

4.2 Test 2: Approval Workflow (test-id2)

Goal and Input: An administrator views a pending reimbursement request in the system (example ID: *RB-2025-0003*, "Travel Reimbursement," $200.00). **Expected Outcome:** Upon clicking "Approve," the request status changes to `Approved` and the ledger is updated; an email notification is sent to the user.
Results and Analysis: During testing, the system correctly updated the reimbursement status and updated the ledger with "Approved." However, cases where the submitter's email address was missing or misformatted led to silent notification failures, leaving the user uninformed.
Code Improvements:

- Implemented an email format validator within a new `NotificationService`.
- Logged invalid email attempts so administrators could correct user contact data.
- Refined multi-level approval for requests over $500, satisfying the SRS's requirement for stricter governance on higher-value transactions.

This ensures that all users now receive a reliable status update, and administrators have a systematic way to track any failed email notifications.

4.3 Test 3: Expense Addition (test-id3)

Goal and Input: Starting with a budget that has zero transactions, a user adds a new expense (**Expense Name: "Software License"**, **Amount:**

$50.00, **Date: 2025-03-01**). **Expected Outcome:** The "Budget Overview" section in the UI should update in real time, reflecting this added expense in the total used amount.
Results and Analysis: On modern hardware, the Budget Overview updated within one second. On older machines, the update occasionally took two to three seconds, suggesting a minor performance bottleneck.
Code Improvements:

- Created a database index on `expenses.date` and `clubID` to speed up queries.
- Moved certain calculations from the front-end to a lightweight server endpoint, reducing rendering times on less powerful devices.

These optimizations brought the update time on older laptops down to about one to two seconds, meeting our real-time budget tracking goals stated in the SRS.

4.4 Test 4: Budget Creation and Categorization (test-id4)

Goal and Input: A user with appropriate permissions creates a new budget with a **Category: "Equipment"** and a **Cap: $500.00**. **Expected Outcome:** The system confirms creation of a "Equipment" budget category and displays it correctly in the dashboard.
Results and Analysis: Budget creation worked as expected, but inconsistent capitalization in the category field ("equipment," "EQUIPMENT," etc.) created multiple labels in the UI.
Code Improvements:

- Modified `BudgetManager` to normalize category inputs to lowercase, preventing duplicates.
- Added auto-suggest to the front end for known categories to further reduce user confusion.

These fixes ensure the final system meets the SRS requirement for coherent budget organization and accurate categorization.

4.5 Test 5: Notification on Reimbursement Status Change (test-id5)

Goal and Input: An admin changes the status of a reimbursement request (e.g., *RB-2025-0004* "Conference Fee") from `pending` to `approved`.
Expected Outcome: The user's dashboard reflects the updated status immediately, and an email notification is sent confirming approval.
Results and Analysis: All tested requests updated the user interface within one second of the admin's action, and email notifications were consistently sent without error. Thus, the system performed as expected with no disruptions or confusion.
Code Improvements:

- Added an optional delivery-time log to the `NotificationService`, helping us monitor any future latency spikes.
- Maintained a uniform approval message template, aligning with the SRS's requirement for consistent user communication.

4.6 Test 6: Audit Trail Logging (test-id6)

Goal and Input: Starting with an idle system, we performed a sequence of actions—budget creation, expense submission, and approval—to check the audit trail's completeness. **Expected Outcome:** Each action should be recorded with the correct user ID, timestamp, and request reference number for compliance tracking.
Results and Analysis: Every action was indeed captured (*e.g.*, "User `u1002` created Budget ID `BG-0003` at 2025-03-02T14:05Z"), but the timestamps initially used local server time (EST), confusing for staff in other time zones.
Code Improvements:

- Standardized timestamps to UTC in `AuditLog`.
- Extended the admin panel with a time-zone converter, ensuring a clear and unified audit trail per SRS compliance requirements.

4.7 Overall Observations and Next Steps

Across Tests 1–6, our system demonstrated strong alignment with the SRS specifications. We uncovered minor usability gaps (form validations, inconsistent category naming) and performance issues on older hardware, each resolved through targeted code improvements. Moving forward, we will:

- Conduct additional stress tests for budgets with a high number of expenses to ensure sustained performance.
- Evaluate the updated `NotificationService` more extensively with user acceptance testing to confirm email reliability.
- Gather stakeholder feedback on the new category auto-suggest to ensure it fosters clarity without cluttering the UI.

These refinements maintain consistency with our planned VnV approach while *not* duplicating the procedures from the plan. We will re-run these tests after each major update to verify no regressions occur and that new features continue to meet the functional requirements set forth in the SRS.

5 Nonfunctional Requirements Evaluation

5.1 Usability

5.2 Performance

5.3 etc.

6 Comparison to Existing Implementation

This section will not be appropriate for every project.

7 Unit Testing

Unit testing was conducted using a combination of automated and manual testing to ensure key functionalities worked as expected. The primary objectives were:

- Verify that critical features, such as reimbursement submissions and approvals, group creation, and user management work correctly.
- Identify and resolve potential errors early in the development cycle.
- Ensure that the system remains stable after modifications.

7.1 Testing Approach

The team used **Jest** for automated testing of frontend logic and API calls, and **manual testing** for real-world scenario validation. The approach included:

- Writing unit tests for key functions such as group creation, deletion, and authentication handling.
- Conducting manual test runs where team members acted as users submitting and approving reimbursement requests.
- Gathering feedback from a small group of MES student leaders who tested the platform and provided insights.
- Running scenario-based testing sessions to simulate real-world usage, such as handling multiple reimbursement requests at once.

7.2 Unit Test Cases

The test cases covered core functionalities, including validation checks and API interactions:

Table 1: Sample Unit Test Cases

Test Case	**Expected Output**
Create a group with a unique name	Group is successfully created
Attempt to create a group with an existing name	System displays an error message
Unauthorized user attempts to access group management	Access is denied with error message

Continued on next page

Table 1 – *Continued from previous page*

Test Case	**Expected Output**
Delete a group with no assigned users	Group is successfully deleted
Attempt to delete a group with assigned users	System prevents deletion and notifies the user
Handle failed database connection during group fetch	Error is logged, and fallback UI is displayed
Register a new user with valid credentials	User is successfully created and logged in
Attempt to register with an existing email	System displays an error message
Login with correct credentials	User is authenticated and redirected to the dashboard
Login with incorrect credentials	System displays an error message
Assign a role to a user	User is successfully assigned the role
Remove a role from a user	User no longer has the role permissions
Attempt to assign an invalid role	System displays an error message
Submit a payment request with valid details	Request is successfully submitted
Submit a payment request missing required fields	System displays an error message
View all payment requests as an admin	Admin sees all requests
View only personal payment requests as a user	User sees only their own requests
Attempt to approve a request without proper permissions	Access is denied with an error message
Update request status successfully	Request status is updated

Continued on next page

Table 1 – *Continued from previous page*

Test Case	**Expected Output**
Handle failed database connection when fetching requests	Error is logged, and fallback UI is displayed

7.3 Edge Cases Tested

- Submitting a reimbursement request without required fields.
- Uploading incorrect file formats for receipts.
- Attempting to create a duplicate group.
- Handling multiple users modifying the same data simultaneously.
- Ensuring database rollback occurs when a transaction fails.
- Registering a user with an invalid email format.
- Assigning multiple roles to a user and verifying access control.
- Submitting a payment request with an incorrect amount format.
- Preventing unauthorized users from updating request statuses.
- Ensuring bulk request approvals work correctly.

8 Changes Due to Testing

Based on the testing process and feedback from users, the following modifications were made to the system:

8.1 Bug Fixes

- Fixed an issue where reimbursements submitted without attachments were still being processed.
- Addressed a bug where users could submit duplicate requests.
- Resolved a login issue where incorrect error messages were displayed.

- Fixed a bug where request status updates were not properly reflected in the UI.
- Added the ability for users to be in multiple groups.

8.2 User Feedback and Enhancements

- Improved the **receipt upload system** by making the file upload process clearer.
- Enhanced **audit logging** for better tracking of request status changes.
- Added a clearer role management interface to reduce confusion when assigning roles.
- Improved user experience by adding filters to search for specific payment requests.

8.3 Performance and Security Improvements

- Implemented basic input validation to prevent incorrect data submissions.
- Reduced system load times by optimizing database queries.
- Strengthened authentication by enforcing stricter password requirements.
- Enhanced role-based access control to prevent unauthorized modifications.

9 Automated Testing

10 Trace to Requirements

11 Trace to Modules

12 Code Coverage Metrics

To ensure the robustness and reliability of the MES-ERP system, we will use a combination of code coverage metrics to evaluate the effectiveness of our testing. The goal of code coverage analysis is to measure the extent to which the source code is tested, helping identify untested paths and potential vulnerabilities.

12.1 Code Coverage Measurement Approach

The following strategies were used to assess the coverage of our codebase:

- **Statement Coverage**: Verifies that each executable statement in the code has been executed at least once.
- **Branch Coverage**: Ensures that both the expected inputs and the expected alerts for wrong inputs are executed (if/else).
- **Function Coverage**: Confirms that all functions and methods in the system have been called at least once.
- **Line Coverage**: Measures the percentage of total lines of code executed during testing.

References

Appendix — Reflection

The information in this section will be used to evaluate the team members on the graduate attribute of Reflection.

The purpose of reflection questions is to give you a chance to assess your own learning and that of your group as a whole, and to find ways to improve in the future. Reflection is an important part of the learning process. Reflection is also an essential component of a successful software development process.

Reflections are most interesting and useful when they're honest, even if the stories they tell are imperfect. You will be marked based on your depth of thought and analysis, and not based on the content of the reflections themselves. Thus, for full marks we encourage you to answer openly and honestly and to avoid simply writing "what you think the evaluator wants to hear."

Please answer the following questions. Some questions can be answered on the team level, but where appropriate, each team member should write their own response:

1. What went well while writing this deliverable?
   Omar: What went well while writing this deliverable was the structured approach we took from the beginning. By clearly defining our Verification and Validation (VnV) strategy early on, we were able to efficiently organize the document and ensure all necessary components were included.

2. What pain points did you experience during this deliverable, and how did you resolve them?
   Omar: One of the main pain points I experienced during this deliverable was ensuring that the VnV plan aligned with our actual testing process and our original VnV plan. To accomodate for this I just went back and forth between documents and talked to my team to ensure I did not mix anything up.

3. Which parts of this document stemmed from speaking to your client(s) or a proxy (e.g. your peers)? Which ones were not, and why?
   Team: The majority of this document stemmed from discussions with my peers, as we collectively determined the best approach for verification and validation. Specific sections, such as requirements and changes due to testing all stem from our stakeholders. We built upon what we

know from our stakeholders for those sections. Other sections were built upon feedback received from stakeholders as well as the TA/Professor, we made sure to account for all information provided to us to ensure a complete report.

4. In what ways was the Verification and Validation (VnV) Plan different from the activities that were actually conducted for VnV? If there were differences, what changes required the modification in the plan? Why did these changes occur? Would you be able to anticipate these changes in future projects? If there weren't any differences, how was your team able to clearly predict a feasible amount of effort and the right tasks needed to build the evidence that demonstrates the required quality? (It is expected that most teams will have had to deviate from their original VnV Plan.)
   Team: The actual VnV activities differed slightly from the original plan, primarily due to adjustments made during the development cycle. Some test cases had to be modified or expanded as we encountered new scenarios that were not initially anticipated, such as specific security vulnerabilities and data integrity concerns. The biggest change was in prioritizing additional integration tests over certain lower-priority unit tests, since we found that issues were more likely to arise in cross-module interactions. These changes occurred because real-world implementation often reveals gaps in planning, and some verification methods turned out to be less effective than expected. In future projects, I would anticipate these changes by leaving more flexibility in the VnV plan and incorporating iterative updates based on early testing feedback.