🛡️ SOFTWARE STABILITY & QUALITY ASSURANCE GUIDE

Ensuring Bug-Free Temple Management Software

For: MandirSync/MandirConnect Development
Target: Zero bugs after installation
Audience: Beginner-friendly with step-by-step guidance
Last Updated: November 2024

🎯 CORE PRINCIPLE

"A temple cannot afford software failures. Ever."

Unlike other software where a bug might be inconvenient, temple software failures mean:

- ❌ Devotees can't donate during festivals (revenue loss)
- ❌ Wrong accounting entries (audit problems)
- ❌ Lost bookings (devotee frustration)
- ❌ Incorrect 80G certificates (legal issues)
- ❌ Data corruption (disaster!)

Therefore: Stability is NOT optional. It's MANDATORY.

TABLE OF CONTENTS

12. Tools & Technologies

---

1. UNDERSTANDING BUGS - WHAT CAN GO WRONG

Types of Bugs in Temple Software

1. Critical Bugs (Software Unusable)

Examples:

- ❌ Software crashes on startup
- ❌ Database connection fails
- ❌ Login not working
- ❌ Payment gateway integration broken
- ❌ Cannot save any data

Impact: Complete failure

Priority: MUST fix before deployment

2. Major Bugs (Core Features Broken)

Examples:

- ❌ Donation entry not saving
- ❌ Receipt printing fails
- ❌ Accounting entries wrong
- ❌ Reports showing incorrect data
- ❌ Bank reconciliation not working

Impact: Core functionality broken

Priority: MUST fix before deployment

3. Moderate Bugs (Features Work but Incorrectly)

Examples:

- ❌ Wrong date format displayed
- ❌ SMS not sending sometimes
- ❌ Search not finding all results
- ❌ Filter not working properly
- ❌ Export missing some columns

Impact: Reduced functionality

Priority: Should fix before deployment

4. Minor Bugs (Cosmetic/UI Issues)

Examples:

❌ Button slightly misaligned
❌ Text color hard to read
❌ Modal doesn't center properly
❌ Tooltip shows late
❌ Icon missing in one place

Impact: User experience affected
Priority: Can fix after deployment

5. Performance Issues (Works but Slow)

Examples:

❌ Report takes 30 seconds to load
❌ Search is slow with 1000+ devotees
❌ Page loads slowly
❌ Database queries slow
❌ Memory usage keeps growing

Impact: User frustration
Priority: Must optimize

6. Security Issues (Data at Risk)

Examples:

❌ SQL injection possible
❌ Passwords stored in plain text
❌ No input validation
❌ XSS attacks possible
❌ Session hijacking possible

Impact: DATA BREACH!
Priority: CRITICAL - Must fix immediately

---

2. PREVENTION STRATEGY - 10 LAYERS OF DEFENSE

Layer 1: Code Standards & Reviews

Layer 2: Automated Linting

Layer 3: Type Checking

Layer 4: Unit Testing (90%+ coverage)

Layer 5: Integration Testing

Layer 6: Manual QA Testing

Layer 7: User Acceptance Testing (UAT)

Layer 8: Staging Environment Testing

Layer 9: Gradual Rollout

Layer 10: Monitoring & Alerts

Think of it like temple security:

- Multiple gates (layers)
- Each gate checks differently
- If one fails, others catch it
- Continuous monitoring

---

3. DEVELOPMENT PHASE - WRITE BUG-FREE CODE

3.1 Use AI Assistant (Cursor with Claude)

Why: AI catches bugs while you write

Setup:

```
bash
```

```
# 1. Install Cursor IDE
# Download from: https://cursor.sh

# 2. Use Claude Sonnet 4.5 (best model)
# Settings → Models → Select Claude Sonnet 4.5

# 3. Enable Copilot++ mode
# Settings → Copilot++ → Enable
```

How to Use:

Example 1: Ask AI to Review Code

python

```python
# You write:
def calculate_donation_tax_benefit(amount):
    return amount * 0.5

# Ask Cursor/Claude:
# "Review this function for bugs and edge cases"

# AI will respond:
# "Issues found:
# 1. No input validation (what if amount is negative?)
# 2. No type checking (what if amount is string?)
# 3. No null check
# 4. Magic number 0.5 should be constant
# 5. No error handling"

# AI suggests:
def calculate_donation_tax_benefit(amount: float) -> float:
    """Calculate 80G tax benefit (50% of donation)"""

    # Input validation
    if not isinstance(amount, (int, float)):
        raise TypeError("Amount must be a number")

    if amount < 0:
        raise ValueError("Amount cannot be negative")

    if amount == 0:
        return 0.0

    # Use constant instead of magic number
    TAX_BENEFIT_RATE = 0.5

    return amount * TAX_BENEFIT_RATE
```

Example 2: Generate Test Cases

python

```
# Ask Cursor/Claude:
# "Generate test cases for the above function"

# AI generates:
def test_calculate_donation_tax_benefit():
    # Normal case
    assert calculate_donation_tax_benefit(1000) == 500

    # Edge case: Zero
    assert calculate_donation_tax_benefit(0) == 0

    # Edge case: Large number
    assert calculate_donation_tax_benefit(1000000) == 500000

    # Edge case: Decimal
    assert calculate_donation_tax_benefit(1000.50) == 500.25

    # Error case: Negative
    with pytest.raises(ValueError):
        calculate_donation_tax_benefit(-100)

    # Error case: String
    with pytest.raises(TypeError):
        calculate_donation_tax_benefit("1000")

    # Error case: None
    with pytest.raises(TypeError):
        calculate_donation_tax_benefit(None)
```

3.2 Code Standards (Prevent Common Bugs)

Install Linting Tools:

```
bash

# For Python (Backend)
pip install pylint black mypy

# For JavaScript/React (Frontend)
npm install -D eslint prettier typescript
```

Configure Auto-Formatting:

```
json
```

```
// .vscode/settings.json
{
  "editor.formatOnSave": true,
  "python.linting.enabled": true,
  "python.linting.pylintEnabled": true,
  "python.formatting.provider": "black",
  "[javascript]": {
    "editor.defaultFormatter": "esbenp.prettier-vscode"
  }
}
```

Result: Code is automatically checked and formatted as you type!

3.3 Type Checking (Catch Bugs Early)

Python Type Hints:

python

```python
# ❌ Without types (bugs hide!)
def create_donation(devotee_id, amount, category):
    # What if devotee_id is string instead of int?
    # What if amount is negative?
    # What if category is None?
    pass

# ✅ With types (bugs caught immediately!)
from typing import Optional
from decimal import Decimal

def create_donation(
    devotee_id: int,
    amount: Decimal,
    category: str,
    notes: Optional[str] = None
) -> dict:
    """
    Create a new donation record.

    Args:
        devotee_id: Database ID of devotee (positive integer)
        amount: Donation amount in INR (must be positive)
        category: Donation category (non-empty string)
        notes: Optional notes

    Returns:
        dict: Created donation with ID and receipt number

    Raises:
        ValueError: If amount is negative or zero
        TypeError: If types are incorrect
    """
    # Input validation
    if not isinstance(devotee_id, int) or devotee_id <= 0:
        raise ValueError("devotee_id must be positive integer")

    if amount <= 0:
        raise ValueError("amount must be positive")

    if not category or not category.strip():
        raise ValueError("category cannot be empty")

    # Your code here...
    pass
```

Run Type Checker:

```bash
mypy your_file.py
```

3.4 Input Validation (Never Trust User Input!)

Every user input must be validated:

```python
```

```python
from pydantic import BaseModel, validator, constr
from decimal import Decimal
from datetime import date

class DonationInput(BaseModel):
    """Donation input validation"""

    devotee_id: int
    amount: Decimal
    category: constr(min_length=1, max_length=100)
    payment_mode: str
    donation_date: date
    pan_number: Optional[constr(regex=r'^[A-Z]{5}[0-9]{4}[A-Z]{1}$')] = None

    @validator('devotee_id')
    def validate_devotee_id(cls, v):
        if v <= 0:
            raise ValueError('devotee_id must be positive')
        return v

    @validator('amount')
    def validate_amount(cls, v):
        if v <= 0:
            raise ValueError('amount must be positive')
        if v > Decimal('10000000'):  # 1 crore max
            raise ValueError('amount too large')
        return v

    @validator('payment_mode')
    def validate_payment_mode(cls, v):
        allowed = ['Cash', 'UPI', 'Card', 'Cheque', 'NEFT', 'Online']
        if v not in allowed:
            raise ValueError(f'payment_mode must be one of {allowed}')
        return v

    @validator('donation_date')
    def validate_date(cls, v):
        if v > date.today():
            raise ValueError('donation_date cannot be in future')
        if v < date(2020, 1, 1):
            raise ValueError('donation_date too old')
        return v

# Usage:
try:
    donation = DonationInput(
```

```
        devotee_id=123,
        amount=Decimal("1000.00"),
        category="General",
        payment_mode="UPI",
        donation_date=date.today()
    )
except ValueError as e:
    # Handle validation error
    print(f"Invalid input: {e}")
```

Result: Invalid data is rejected BEFORE it reaches database!

3.5 Error Handling (Graceful Failures)

Every function should handle errors:

python

```python
import logging
from typing import Optional

logger = logging.getLogger(__name__)

def save_donation(donation_data: dict) -> Optional[dict]:
    """
    Save donation with comprehensive error handling.

    Returns:
        dict: Saved donation with ID, or None if failed
    """
    try:
        # Validate input
        validated = DonationInput(**donation_data)

        # Save to database
        db_session = get_database_session()

        try:
            donation = Donation(**validated.dict())
            db_session.add(donation)
            db_session.commit()

            # Generate receipt
            receipt = generate_receipt(donation.id)

            # Send SMS (non-critical, don't fail if this fails)
            try:
                send_sms(validated.devotee_id, receipt.number)
            except Exception as e:
                logger.warning(f"SMS failed but continuing: {e}")

            logger.info(f"Donation saved successfully: {donation.id}")
            return donation.to_dict()

        except Exception as db_error:
            db_session.rollback()
            logger.error(f"Database error: {db_error}")
            raise

    except ValueError as e:
        logger.error(f"Validation error: {e}")
        return None

    except Exception as e:
```

```
        logger.error(f"Unexpected error saving donation: {e}")
        return None
```

Benefits:

- Database rollback on error (data stays consistent)
- Errors logged (you can debug later)
- Non-critical failures don't break everything
- User gets meaningful error message

3.6 Database Transactions (Prevent Data Corruption)

Always use transactions for critical operations:

```
python
```

```
from sqlalchemy.orm import Session

def process_hundi_opening(
    hundi_id: int,
    cash_count: dict,
    total_amount: Decimal,
    counted_by: list[int]
) -> bool:
    """
    Process hundi opening with transaction safety.
    All-or-nothing operation.
    """
    session: Session = get_database_session()

    try:
        # Start transaction
        session.begin()

        # Step 1: Update hundi status
        hundi = session.query(Hundi).get(hundi_id)
        hundi.status = 'Opened'
        hundi.opened_at = datetime.now()

        # Step 2: Record cash count
        count_record = HundiCount(
            hundi_id=hundi_id,
            denominations=cash_count,
            total_amount=total_amount,
            counted_by=counted_by
        )
        session.add(count_record)

        # Step 3: Create accounting entry
        voucher = create_receipt_voucher(
            amount=total_amount,
            source='Hundi',
            source_id=hundi_id
        )
        session.add(voucher)

        # Step 4: Update cash balance
        cash_account = session.query(Account).filter_by(
            code='CASH_COUNTER_1'
        ).first()
        cash_account.balance += total_amount
```

```
        # All steps succeeded - commit!
        session.commit()

        logger.info(f"Hundi {hundi_id} processed successfully")
        return True

    except Exception as e:
        # ANY error - rollback EVERYTHING
        session.rollback()
        logger.error(f"Hundi processing failed: {e}")
        return False
```

Result: Either ALL steps complete, or NONE complete. No partial data!

---

4. TESTING STRATEGY - 6 TYPES OF TESTING

4.1 Unit Testing (Test Individual Functions)

What: Test each function in isolation

Tools: pytest (Python), Jest (JavaScript)

Example:

python

```python
# test_donation_service.py
import pytest
from decimal import Decimal
from services.donation_service import calculate_tax_benefit

def test_calculate_tax_benefit_normal():
    """Test normal case"""
    result = calculate_tax_benefit(Decimal("1000"))
    assert result == Decimal("500")

def test_calculate_tax_benefit_zero():
    """Test edge case: zero"""
    result = calculate_tax_benefit(Decimal("0"))
    assert result == Decimal("0")

def test_calculate_tax_benefit_large():
    """Test edge case: large number"""
    result = calculate_tax_benefit(Decimal("1000000"))
    assert result == Decimal("500000")

def test_calculate_tax_benefit_negative():
    """Test error case: negative"""
    with pytest.raises(ValueError):
        calculate_tax_benefit(Decimal("-100"))

def test_calculate_tax_benefit_invalid_type():
    """Test error case: wrong type"""
    with pytest.raises(TypeError):
        calculate_tax_benefit("1000")
```

Run Tests:

```
bash

pytest test_donation_service.py -v
```

Target: 90%+ code coverage

4.2 Integration Testing (Test Multiple Components Together)

What: Test how components work together

Example:

```
python
```

```python
# test_donation_flow.py
import pytest
from app import create_app
from models import db, Devotee, Donation

@pytest.fixture
def client():
    """Create test client"""
    app = create_app('testing')
    with app.test_client() as client:
        with app.app_context():
            db.create_all()
            yield client
            db.drop_all()

def test_complete_donation_flow(client):
    """Test entire donation flow"""

    # Step 1: Create devotee
    response = client.post('/api/devotees', json={
        'name': 'Test Devotee',
        'phone': '9876543210',
        'email': 'test@example.com'
    })
    assert response.status_code == 201
    devotee_id = response.json['id']

    # Step 2: Create donation
    response = client.post('/api/donations', json={
        'devotee_id': devotee_id,
        'amount': 1000,
        'category': 'General',
        'payment_mode': 'Cash'
    })
    assert response.status_code == 201
    donation_id = response.json['id']

    # Step 3: Verify accounting entry created
    response = client.get(f'/api/accounting/vouchers')
    vouchers = response.json['data']
    assert len(vouchers) == 1
    assert vouchers[0]['amount'] == 1000

    # Step 4: Verify 80G certificate generated
    response = client.get(
        f'/api/donations/{donation_id}/certificate'
```

```
)
assert response.status_code == 200
assert '80G' in response.text

# Step 5: Verify reports updated
response = client.get('/api/reports/daily-summary')
summary = response.json
assert summary['total_donations'] == 1000
```

4.3 API Testing (Test All Endpoints)

What: Test every API endpoint

Tool: Postman, pytest

Example:

```
python
```

```python
# test_api_endpoints.py
import pytest

def test_get_donations_list(client):
    """Test GET /api/donations"""
    response = client.get('/api/donations')
    assert response.status_code == 200
    assert 'data' in response.json
    assert 'total' in response.json

def test_get_donation_by_id(client):
    """Test GET /api/donations/{id}"""
    response = client.get('/api/donations/1')
    assert response.status_code == 200
    assert response.json['id'] == 1

def test_get_nonexistent_donation(client):
    """Test GET /api/donations/{invalid_id}"""
    response = client.get('/api/donations/99999')
    assert response.status_code == 404

def test_create_donation_missing_field(client):
    """Test POST with missing required field"""
    response = client.post('/api/donations', json={
        'amount': 1000
        # Missing devotee_id!
    })
    assert response.status_code == 400
    assert 'devotee_id' in response.json['error']

def test_create_donation_invalid_amount(client):
    """Test POST with invalid amount"""
    response = client.post('/api/donations', json={
        'devotee_id': 1,
        'amount': -1000  # Negative!
    })
    assert response.status_code == 400
```

4.4 Database Testing (Test Data Operations)

What: Test database queries and transactions

Example:

python

```python
# test_database.py
import pytest
from models import db, Donation, Account
from decimal import Decimal

def test_donation_creation(db_session):
    """Test creating donation in database"""
    donation = Donation(
        devotee_id=1,
        amount=Decimal("1000.00"),
        category="General",
        payment_mode="Cash"
    )
    db_session.add(donation)
    db_session.commit()

    # Verify saved
    saved = db_session.query(Donation).get(donation.id)
    assert saved is not None
    assert saved.amount == Decimal("1000.00")

def test_transaction_rollback(db_session):
    """Test transaction rollback on error"""
    try:
        # Start transaction
        db_session.begin()

        # Add donation
        donation = Donation(
            devotee_id=1,
            amount=Decimal("1000")
        )
        db_session.add(donation)

        # Simulate error
        raise Exception("Simulated error")

    except Exception:
        db_session.rollback()

    # Verify nothing was saved
    count = db_session.query(Donation).count()
    assert count == 0

def test_accounting_integration(db_session):
    """Test donation creates accounting entry"""
```

```
# Create donation
donation = create_donation(
    devotee_id=1,
    amount=Decimal("1000")
)

# Verify accounting entry
vouchers = db_session.query(Voucher).filter_by(
    source='Donation',
    source_id=donation.id
).all()

assert len(vouchers) == 1
assert vouchers[0].amount == Decimal("1000")
```

4.5 UI Testing (Test User Interface)

What: Test frontend components

Tool: Jest, React Testing Library, Cypress

Example:

javascript

```jsx
// DonationForm.test.jsx
import { render, screen, fireEvent } from '@testing-library/react';
import DonationForm from './DonationForm';

test('renders donation form', () => {
  render(<DonationForm />);

  // Check all fields present
  expect(screen.getByLabelText('Devotee')).toBeInTheDocument();
  expect(screen.getByLabelText('Amount')).toBeInTheDocument();
  expect(screen.getByLabelText('Category')).toBeInTheDocument();
  expect(screen.getByRole('button', { name: 'Save' })).toBeInTheDocument();
});

test('validates amount field', () => {
  render(<DonationForm />);

  const amountInput = screen.getByLabelText('Amount');
  const submitButton = screen.getByRole('button', { name: 'Save' });

  // Enter negative amount
  fireEvent.change(amountInput, { target: { value: '-100' } });
  fireEvent.click(submitButton);

  // Check error message
  expect(screen.getByText('Amount must be positive')).toBeInTheDocument();
});

test('submits form successfully', async () => {
  const mockSubmit = jest.fn();
  render(<DonationForm onSubmit={mockSubmit} />);

  // Fill form
  fireEvent.change(screen.getByLabelText('Amount'), {
    target: { value: '1000' }
  });
  fireEvent.change(screen.getByLabelText('Category'), {
    target: { value: 'General' }
  });

  // Submit
  fireEvent.click(screen.getByRole('button', { name: 'Save' }));

  // Verify callback called
  await waitFor(() => {
    expect(mockSubmit).toHaveBeenCalledWith({
```

```
    amount: 1000,
    category: 'General'
  });
 });
});
```

4.6 End-to-End Testing (Test Complete User Flows)

What: Test entire user journeys

Tool: Cypress, Selenium

Example:

javascript

```
// cypress/e2e/donation_flow.cy.js
describe('Complete Donation Flow', () => {
  it('should create donation and generate receipt', () => {
    // Login
    cy.visit('/login');
    cy.get('[name="username"]').type('admin');
    cy.get('[name="password"]').type('password123');
    cy.get('button[type="submit"]').click();

    // Navigate to donations
    cy.url().should('include', '/dashboard');
    cy.contains('Donations').click();
    cy.contains('New Donation').click();

    // Fill donation form
    cy.get('[name="devotee"]').type('Test Devotee');
    cy.get('[name="amount"]').type('1000');
    cy.get('[name="category"]').select('General');
    cy.get('[name="payment_mode"]').select('Cash');

    // Submit
    cy.contains('Save').click();

    // Verify success message
    cy.contains('Donation saved successfully').should('be.visible');

    // Verify receipt generated
    cy.contains('Print Receipt').should('be.visible');
    cy.contains('Print Receipt').click();

    // Verify receipt content
    cy.contains('DONATION RECEIPT').should('be.visible');
    cy.contains('₹1,000.00').should('be.visible');
    cy.contains('80G Certificate').should('be.visible');
  });
});
```

Run E2E Tests:

```
bash

npx cypress open
```

---

5. AUTOMATED TESTING - LET ROBOTS TEST

5.1 Continuous Integration (CI) Setup

What: Automatically run tests on every code change

Tool: GitHub Actions (free!)

Setup:

```yaml
```

```yaml
# .github/workflows/tests.yml
name: Test Suite

on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main, develop ]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2

    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: '3.11'

    - name: Install dependencies
      run: |
        pip install -r requirements.txt
        pip install pytest pytest-cov

    - name: Run linting
      run: |
        pylint app/

    - name: Run type checking
      run: |
        mypy app/

    - name: Run unit tests
      run: |
        pytest tests/ -v --cov=app --cov-report=html

    - name: Check coverage
      run: |
        coverage report --fail-under=90

    - name: Run integration tests
      run: |
        pytest tests/integration/ -v
```

```
    - name: Upload coverage
      uses: codecov/codecov-action@v2
```

Result:

- Tests run automatically on every commit
- Pull requests can't merge if tests fail
- Coverage report generated
- Team notified of failures

5.2 Pre-commit Hooks (Prevent Bad Code)

What: Run checks before code is committed

Setup:

```bash
# Install pre-commit
pip install pre-commit

# Create config file
```

```yaml
```

```yaml
# .pre-commit-config.yaml
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.4.0
    hooks:
      - id: trailing-whitespace
      - id: end-of-file-fixer
      - id: check-yaml
      - id: check-json
      - id: check-added-large-files
      - id: check-merge-conflict

  - repo: https://github.com/psf/black
    rev: 23.3.0
    hooks:
      - id: black
        language_version: python3.11

  - repo: https://github.com/PyCQA/pylint
    rev: v3.0.0
    hooks:
      - id: pylint
        args: [--fail-under=8.0]

  - repo: local
    hooks:
      - id: pytest
        name: pytest
        entry: pytest
        language: system
        pass_filenames: false
        always_run: true
```

```bash
# Install hooks
pre-commit install
```

Result: Can't commit code that fails tests!

---

6. MANUAL TESTING - HUMAN QA PROCESS

6.1 Test Cases Document

Create detailed test cases for every feature:

markdown

TEST CASE: TC-DON-001

Feature: Donation Entry

Priority: Critical

Preconditions:

- User logged in
- At least one devotee exists
- At least one donation category exists

Test Steps:

1. Navigate to Donations → New Donation
2. Select devotee from dropdown
3. Enter amount: 1000
4. Select category: General
5. Select payment mode: Cash
6. Click Save button

Expected Result:

- Success message displayed
- Donation saved with unique ID
- Receipt generated with unique number
- 80G certificate auto-generated
- Accounting entry created (Debit: Cash, Credit: Income)
- SMS sent to devotee (if phone number exists)
- Donation appears in donations list

Actual Result:

[To be filled during testing]

Pass/Fail:

[To be filled during testing]

Notes:

[Any observations]

Tested By:

[Tester name]

Date:

[Testing date]

Create 100+ test cases covering:

- All features
- All user roles

- All edge cases
- All error scenarios

6.2 Exploratory Testing

What: Tester freely explores software looking for bugs

Process:

```
Day 1: Explore Donation Module (4 hours)
- Try creating donations in every way possible
- Try entering invalid data
- Try very large amounts
- Try special characters
- Try rapid clicking
- Try browser back button
- Try different browsers
- Document every bug found

Day 2: Explore Booking Module (4 hours)
[Similar exploration]

Day 3: Explore Accounting Module (4 hours)
[Similar exploration]

...continue for all modules
```

Goal: Find bugs that test cases miss!

6.3 Regression Testing (After Every Change)

What: Retest everything after fixing bugs or adding features

Process:

```
1. Developer fixes Bug #123
2. QA retests Bug #123 (verify fixed)
3. QA runs FULL test suite (verify nothing else broke)
4. If anything broke → report new bugs
5. Repeat until clean
```

Tools:

- TestRail (test management)
- Jira (bug tracking)

- Spreadsheet (simple option)

---

7. USER ACCEPTANCE TESTING - TEMPLE STAFF TESTS

7.1 Beta Testing with Real Temple

Find 1-2 temples to test before full launch:

Week 1: Setup
- Install software at temple
- Train 2-3 staff members
- Observe them using software
- Note confusions/difficulties

Week 2-4: Daily Use
- Temple uses for ALL operations
- Daily check-in calls
- Log all issues
- Quick fixes for critical bugs

Week 5: Feedback Session
- Detailed feedback meeting
- Feature requests
- Pain points
- What works well

Week 6: Refinement
- Fix reported bugs
- Improve UI based on feedback
- Add small requested features
- Prepare for next temple

7.2 UAT Test Scenarios

Create real-world scenarios:

SCENARIO 1: FESTIVAL DAY RUSH

Situation: Kartik Purnima, 500+ devotees expected

Tasks:

1. Create 50 donations in 30 minutes
2. Book 20 sevas for next month
3. Open and count 3 hundis
4. Generate daily report

Success Criteria:

- No crashes
- All data saved correctly
- Receipts printed quickly
- No queue formation

SCENARIO 2: MONTH-END CLOSING

Situation: Last day of month, need reports for board meeting

Tasks:

1. Close all pending vouchers
2. Reconcile all bank accounts
3. Generate income statement
4. Generate balance sheet
5. Generate donation summary
6. Export all reports to PDF

Success Criteria:

- All reports accurate
- Numbers match manually calculated totals
- Reports generated in <5 minutes
- Professional formatting

SCENARIO 3: AUDIT PREPARATION

Situation: Annual audit in 2 weeks

Tasks:

1. Generate all required reports
2. Export voucher register
3. Export 80G certificate register
4. Generate TDS reports
5. Check all backup documents attached

Success Criteria:

- All reports available
- CA finds all needed information
- No data missing
- No calculation errors

---

8. DEPLOYMENT STRATEGY - SAFE ROLLOUT

8.1 Staged Deployment

Don't deploy to all temples at once!

STAGE 1: DEVELOPMENT
- Developer machine only
- Frequent changes
- Lots of bugs OK

STAGE 2: STAGING (Exact copy of production)
- Test environment
- No real temple data
- Extensive testing
- Must pass ALL tests

STAGE 3: PILOT (1-2 temples)
- Real temples
- Real operations
- Close monitoring
- Quick rollback if issues

STAGE 4: LIMITED ROLLOUT (5-10 temples)
- More temples
- Different sizes/types
- Monitor for patterns
- Refine based on feedback

STAGE 5: GENERAL AVAILABILITY
- All temples
- Confident system is stable
- Support team ready

Timeline: 2-3 months from Stage 1 to Stage 5

8.2 Database Migrations (Safe Data Updates)

Never manually edit production database!

Use migration tool:

```bash
# Install Alembic (Python)
pip install alembic

# Initialize
alembic init migrations

# Create migration
alembic revision --autogenerate -m "Add devotee phone field"

# Review generated migration
# migrations/versions/abc123_add_devotee_phone.py

# Apply migration
alembic upgrade head

# Rollback if needed
alembic downgrade -1
```

Result: Database changes are tracked, reversible, safe!

8.3 Feature Flags (Turn Features On/Off)

Deploy new features hidden, enable gradually:

```python
```

```python
# config.py
FEATURE_FLAGS = {
    'online_payments': False,  # Not ready yet
    'sms_notifications': True,  # Stable
    'whatsapp_receipts': False,  # Testing
    'ai_donation_insights': False  # Future
}

# In code:
from config import FEATURE_FLAGS

def create_donation(data):
    # ... save donation ...

    # Send SMS only if enabled
    if FEATURE_FLAGS['sms_notifications']:
        send_sms(devotee.phone, receipt_url)

    # Send WhatsApp if enabled (gradual rollout)
    if FEATURE_FLAGS['whatsapp_receipts']:
        send_whatsapp(devotee.phone, receipt_url)
```

Benefits:

- Deploy code without enabling feature
- Enable for some temples only
- Quick disable if bug found
- A/B testing possible

8.4 Rollback Plan

Always have plan to undo deployment:

ROLLBACK CHECKLIST:

Before Deployment:
☐ Database backup taken
☐ Previous version files saved
☐ Configuration files backed up
☐ Rollback script tested

If Deployment Fails:
☐ Stop application
☐ Restore previous version
☐ Restore database (if schema changed)
☐ Restart application
☐ Verify working
☐ Notify team

If Bugs Found Post-Deployment:
☐ Assess severity
☐ If critical → immediate rollback
☐ If moderate → hotfix or rollback
☐ If minor → fix in next release

Rollback Script:

```
#!/bin/bash
# Stop application
sudo systemctl stop mandir-app

# Restore previous version
cp -r /backups/version-1.2/* /var/www/mandir-app/

# Restore database
pg_restore -d temple_db /backups/db-2024-11-20.dump

# Restart
sudo systemctl start mandir-app

# Verify
curl http://localhost:8000/health
```

---

9. MONITORING & ALERTS - CATCH ISSUES EARLY

9.1 Application Monitoring

Install monitoring tools:

```python
# requirements.txt
sentry-sdk[fastapi]==1.40.0
prometheus-client==0.19.0

# app.py
import sentry_sdk
from sentry_sdk.integrations.fastapi import FastAPIIntegration

# Initialize Sentry (error tracking)
sentry_sdk.init(
    dsn="your-sentry-dsn",
    integrations=[FastAPIIntegration()],
    traces_sample_rate=1.0,
    environment="production"
)

# Now ALL errors are automatically logged to Sentry!
```

Set up alerts:

```
Alert Rule 1: ANY error in production
→ Slack notification immediately
→ Email to dev team

Alert Rule 2: >10 errors in 5 minutes
→ Slack notification immediately
→ SMS to on-call engineer
→ Auto-create incident

Alert Rule 3: Response time >2 seconds
→ Slack notification
→ Performance investigation needed
```

9.2 Database Monitoring

```python
```

```python
# Add query logging
import logging

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('app.log'),
        logging.StreamHandler()
    ]
)

# Log slow queries
from sqlalchemy import event
from sqlalchemy.engine import Engine
import time

@event.listens_for(Engine, "before_cursor_execute")
def before_cursor_execute(conn, cursor, statement, parameters, context, executemany):
    conn.info.setdefault('query_start_time', []).append(time.time())

@event.listens_for(Engine, "after_cursor_execute")
def after_cursor_execute(conn, cursor, statement, parameters, context, executemany):
    total = time.time() - conn.info['query_start_time'].pop(-1)

    # Alert on slow queries (>1 second)
    if total > 1.0:
        logging.warning(f"Slow query ({total:.2f}s): {statement}")
```

9.3 Health Checks

Create health check endpoint:

```python
```

```python
from fastapi import APIRouter, HTTPException
from sqlalchemy import text

router = APIRouter()

@router.get("/health")
async def health_check():
    """
    Check if application is healthy.
    Used by monitoring tools and load balancers.
    """
    checks = {
        "api": "unknown",
        "database": "unknown",
        "redis": "unknown",
        "disk_space": "unknown"
    }

    # Check API
    checks["api"] = "healthy"

    # Check database
    try:
        db = get_database_session()
        db.execute(text("SELECT 1"))
        checks["database"] = "healthy"
    except Exception as e:
        checks["database"] = f"unhealthy: {str(e)}"

    # Check Redis
    try:
        redis_client.ping()
        checks["redis"] = "healthy"
    except Exception as e:
        checks["redis"] = f"unhealthy: {str(e)}"

    # Check disk space
    import shutil
    stat = shutil.disk_usage("/")
    free_gb = stat.free / (1024**3)
    if free_gb < 5:  # Less than 5GB free
        checks["disk_space"] = f"warning: only {free_gb:.1f}GB free"
    else:
        checks["disk_space"] = "healthy"

    # Overall status
```

```python
    all_healthy = all(
        v == "healthy" for v in checks.values()
    )

    if not all_healthy:
        raise HTTPException(status_code=503, detail=checks)

    return {"status": "healthy", "checks": checks}
```

Monitor health endpoint:

```bash
# Check every 1 minute
*/1 * * * * curl http://localhost:8000/health || echo "ALERT: Health check failed!"
```

9.4 User Activity Monitoring

Track what users do:

```python
```

```
from models import ActivityLog

def log_activity(user_id: int, action: str, details: dict):
    """Log user activity for security and debugging"""
    log = ActivityLog(
        user_id=user_id,
        action=action,
        details=details,
        ip_address=request.remote_addr,
        user_agent=request.headers.get('User-Agent'),
        timestamp=datetime.now()
    )
    db.session.add(log)
    db.session.commit()

# Usage:
@router.post("/donations")
async def create_donation(data: DonationInput, current_user: User):
    donation = save_donation(data)

    # Log activity
    log_activity(
        user_id=current_user.id,
        action="donation_created",
        details={
            "donation_id": donation.id,
            "amount": float(donation.amount),
            "category": donation.category
        }
    )

    return donation
```

Review activity logs:

- Daily: Check for unusual patterns
- Weekly: Review most active users
- Monthly: Analyze feature usage
- On bug report: Check what user did before bug

---

10. MAINTENANCE & UPDATES - KEEP IT STABLE

10.1 Regular Updates Schedule

WEEKLY:
- Review error logs
- Check system health metrics
- Apply security patches (if any)
- Review user feedback

MONTHLY:
- Bug fixes release
- Minor feature updates
- Performance optimizations
- Database maintenance

QUARTERLY:
- Major feature release
- Comprehensive testing
- User training on new features
- Documentation updates

ANNUALLY:
- Major version upgrade
- Architecture review
- Security audit
- Performance benchmark

10.2 Database Maintenance

python

```python
# Automated weekly maintenance script
# run_maintenance.py

import logging
from sqlalchemy import text

def vacuum_database():
    """Reclaim storage and optimize queries"""
    logging.info("Starting database vacuum...")
    db.execute(text("VACUUM ANALYZE"))
    logging.info("Database vacuum completed")

def backup_database():
    """Create backup"""
    import subprocess
    from datetime import datetime

    backup_file = f"/backups/db_{datetime.now().strftime('%Y%m%d')}.dump"

    subprocess.run([
        "pg_dump",
        "-h", "localhost",
        "-U", "postgres",
        "-F", "c",  # Custom format
        "-f", backup_file,
        "temple_db"
    ])

    logging.info(f"Backup created: {backup_file}")

def cleanup_old_logs():
    """Delete logs older than 90 days"""
    cutoff_date = datetime.now() - timedelta(days=90)

    db.execute(
        text("DELETE FROM activity_logs WHERE timestamp < :cutoff"),
        {"cutoff": cutoff_date}
    )

    logging.info("Old logs cleaned up")

def check_data_integrity():
    """Verify data consistency"""
    # Example: Check if accounting balanced
    result = db.execute(text("""
        SELECT
```

```
        SUM(CASE WHEN debit_credit = 'Debit' THEN amount ELSE 0 END) as total_debit,
        SUM(CASE WHEN debit_credit = 'Credit' THEN amount ELSE 0 END) as total_credit
    FROM voucher_details
"""")).fetchone()

    if result.total_debit != result.total_credit:
        logging.error("CRITICAL: Accounts not balanced!")
        send_alert("Accounting imbalance detected!")
    else:
        logging.info("Data integrity check passed")

if __name__ == "__main__":
    vacuum_database()
    backup_database()
    cleanup_old_logs()
    check_data_integrity()
```

Schedule with cron:

```
bash

# Run every Sunday at 2 AM
0 2 * * 0 /usr/bin/python3 /app/run_maintenance.py
```

10.3 Performance Optimization

Monitor and optimize slow queries:

```
python
```

```
# Find slow queries from logs
SELECT
    query,
    COUNT(*) as frequency,
    AVG(duration) as avg_duration,
    MAX(duration) as max_duration
FROM query_logs
WHERE duration > 1.0  -- Queries taking >1 second
GROUP BY query
ORDER BY avg_duration DESC
LIMIT 10;

# Add indexes for slow queries
CREATE INDEX idx_donations_date ON donations(donation_date);
CREATE INDEX idx_donations_devotee ON donations(devotee_id);
CREATE INDEX idx_vouchers_date ON vouchers(voucher_date);

# Analyze query performance
EXPLAIN ANALYZE
SELECT * FROM donations WHERE donation_date >= '2024-01-01';
```

10.4 Security Updates

Regular security checks:

```
bash

# Check for vulnerable dependencies
pip install safety
safety check

# Update dependencies
pip list --outdated
pip install --upgrade package-name

# Security audit
pip install bandit
bandit -r app/

# Check for SQL injection vulnerabilities
pip install sqlmap
```

---

11. COMPLETE TESTING CHECKLIST

Pre-Deployment Checklist

☐ DEVELOPMENT COMPLETE
- ☐ All features implemented
- ☐ Code reviewed by 2+ people
- ☐ No hardcoded credentials
- ☐ Environment variables configured
- ☐ Documentation updated

☐ UNIT TESTS
- ☐ 90%+ code coverage
- ☐ All tests passing
- ☐ Edge cases covered
- ☐ Error cases covered

☐ INTEGRATION TESTS
- ☐ All API endpoints tested
- ☐ All database operations tested
- ☐ All module integrations tested
- ☐ All tests passing

☐ SECURITY TESTS
- ☐ SQL injection tested
- ☐ XSS tested
- ☐ CSRF protection enabled
- ☐ Authentication tested
- ☐ Authorization tested
- ☐ Input validation tested
- ☐ Dependency vulnerabilities checked

☐ PERFORMANCE TESTS
- ☐ Load testing done (100+ concurrent users)
- ☐ Stress testing done
- ☐ All queries <1 second
- ☐ API response times <500ms
- ☐ Memory leaks checked

☐ USER INTERFACE TESTS
- ☐ All pages tested in Chrome
- ☐ All pages tested in Firefox
- ☐ All pages tested in Safari
- ☐ Mobile responsive tested
- ☐ Tablet responsive tested
- ☐ All forms validated

- ☐ All error messages clear

☐ MANUAL TESTING
- ☐ Happy path scenarios tested
- ☐ Error scenarios tested
- ☐ Edge cases tested
- ☐ Exploratory testing done (8+ hours)
- ☐ All found bugs fixed

☐ USER ACCEPTANCE TESTING
- ☐ Tested by temple staff (5+ people)
- ☐ Real-world scenarios tested
- ☐ Feedback incorporated
- ☐ Training materials created
- ☐ User manual created

☐ DEPLOYMENT PREPARATION
- ☐ Database backed up
- ☐ Rollback plan documented
- ☐ Monitoring configured
- ☐ Alerts configured
- ☐ Health checks working
- ☐ Staging environment tested

☐ DOCUMENTATION
- ☐ API documentation complete
- ☐ User manual complete
- ☐ Admin guide complete
- ☐ Troubleshooting guide complete
- ☐ Training videos created

☐ SUPPORT READINESS
- ☐ Support team trained
- ☐ Support tickets system ready
- ☐ Escalation process defined
- ☐ FAQ document created
- ☐ Support hours defined

☐ FINAL CHECKS
- ☐ All team members approved
- ☐ Stakeholders approved
- ☐ Deployment date/time set
- ☐ Communication plan ready
- ☐ Celebration planned! 🎉

12. TOOLS & TECHNOLOGIES

Testing Tools

Python (Backend)

```bash
# Testing frameworks
pytest==7.4.3              # Unit & integration testing
pytest-cov==4.1.0          # Code coverage
pytest-asyncio==0.21.1     # Async testing

# Code quality
pylint==3.0.2              # Linting
black==23.11.0             # Code formatting
mypy==1.7.0                # Type checking
bandit==1.7.5              # Security checking

# Performance
locust==2.17.0             # Load testing
py-spy==0.3.14             # Profiling
```

JavaScript/React (Frontend)

```bash
# Testing frameworks
jest                       # Unit testing
@testing-library/react     # React component testing
@testing-library/user-event  # User interaction testing
cypress                    # E2E testing

# Code quality
eslint                     # Linting
prettier                   # Code formatting
typescript                 # Type checking
```

Monitoring Tools

```bash
```

```
# Error tracking
sentry-sdk              # Error monitoring & tracking

# Performance monitoring
prometheus-client       # Metrics collection
grafana                 # Metrics visualization

# Logging
python-json-logger      # Structured logging
logstash                # Log aggregation

# Uptime monitoring
uptimerobot.com         # Free uptime monitoring
statuspage.io           # Status page for customers
```

Development Tools

```
bash

# Version control
git                     # Source control
github                  # Code hosting

# CI/CD
github-actions          # Automated testing & deployment

# Code review
github-pr-review        # Pull request reviews

# Project management
jira / github-projects  # Task tracking
confluence              # Documentation
```

SUMMARY: 10-STEP STABILITY CHECKLIST

For Every Release:

STEP 1: CODE QUALITY ✅

- [ ] Use AI assistant (Cursor/Claude) for code review
- [ ] Run linting (pylint/eslint)
- [ ] Run type checking (mypy/typescript)
- [ ] Code coverage >90%

STEP 2: UNIT TESTING ✅

- [ ] Test every function
- [ ] Test edge cases
- [ ] Test error cases
- [ ] All tests pass

STEP 3: INTEGRATION TESTING ✅

- [ ] Test API endpoints
- [ ] Test database operations
- [ ] Test module interactions
- [ ] All tests pass

STEP 4: SECURITY TESTING ✅

- [ ] Check SQL injection
- [ ] Check XSS
- [ ] Check authentication
- [ ] Check authorization
- [ ] No vulnerabilities found

STEP 5: MANUAL TESTING ✅

- [ ] Test all features manually
- [ ] Test on different browsers
- [ ] Test on mobile
- [ ] Exploratory testing (4+ hours)
- [ ] All bugs fixed

STEP 6: USER ACCEPTANCE TESTING ✅

- [ ] Temple staff testing (1-2 weeks)
- [ ] Real-world scenarios
- [ ] Feedback incorporated
- [ ] Approval received

STEP 7: STAGING DEPLOYMENT ✅

- [ ] Deploy to staging
- [ ] Run all tests on staging
- [ ] Performance testing
- [ ] Load testing
- [ ] All checks pass

STEP 8: PRODUCTION DEPLOYMENT ✅

- [ ] Database backup
- [ ] Deploy to production
- [ ] Verify health checks

- [ ] Smoke testing
- [ ] Monitor for 24 hours

STEP 9: MONITORING SETUP ✅

- [ ] Error tracking enabled (Sentry)
- [ ] Performance monitoring enabled
- [ ] Health checks running
- [ ] Alerts configured
- [ ] Logs being collected

STEP 10: SUPPORT READY ✅

- [ ] Support team trained
- [ ] Documentation complete
- [ ] Rollback plan ready
- [ ] On-call schedule set
- [ ] Communication plan ready

FINAL THOUGHTS

Software stability is NOT achieved by luck.

It's achieved by:

1. ✅ Writing good code (with AI help!)
2. ✅ Testing thoroughly (automated + manual)
3. ✅ Deploying carefully (staged rollout)
4. ✅ Monitoring constantly (catch issues early)
5. ✅ Maintaining regularly (keep it healthy)

For temple software specifically:

Critical Success Factors:

- 🎯 **Financial Accuracy** - Accounting must be 100% accurate
- 🎯 **Data Integrity** - Never lose data, never corrupt data
- 🎯 **Reliability** - Work during festivals when traffic is 10x
- 🎯 **Security** - Protect devotee data and donations
- 🎯 **Ease of Use** - Temple staff should find it simple

Follow this guide, and you'll have:

- ✅ Software that WORKS

- ✅ Software that's FAST
- ✅ Software that's SECURE
- ✅ Software that SCALES
- ✅ Software that LASTS

Most importantly:

- ✅ Temples will TRUST your software
- ✅ Devotees will have GREAT experience
- ✅ You'll sleep well at night! 😊

Questions to Ask Yourself Before Deployment:

1. **Would I trust my temple's money with this?**
   - If not, don't deploy.
2. **Would my grandmother find this easy to use?**
   - If not, simplify.
3. **Can this handle 10x current load?**
   - If not, optimize.
4. **If it breaks at 2 AM, can I fix it quickly?**
   - If not, improve monitoring.
5. **Would I be proud showing this to other temples?**
   - If not, polish more.

Only deploy when you can answer YES to all 5!

END OF STABILITY GUIDE

May your software be as stable as a temple's foundation! 🏛️