Validation Prompt Analysis - Scope & Reusability

Document Purpose: Analyze whether the Claude Code validation prompt is universal or phase-specific, and how to adapt it for different contexts.

Date: 2025-11-09
Prompt File: `CLAUDE_CODE_VALIDATION_PROMPT.md`

🎯 Quick Answer

The validation prompt is 40% universal, 60% context-specific.

Universal Components (Reusable Across Phases/Sprints):

- ✅ Systematic validation methodology
- ✅ Error categorization approach
- ✅ Fix prioritization framework
- ✅ Testing validation patterns
- ✅ Documentation standards
- ✅ Common TypeScript fix patterns

Context-Specific Components (Sprint 1 / Current Platform State):

- 🔵 Specific file references (plan.ts, material.ts, auth.ts)
- 🔵 Sprint 1 security features focus
- 🔵 Current database schema expectations
- 🔵 Known issue list (schema mismatches)
- 🔵 Test credentials and data
- 🔵 Current tech stack configuration

Verdict: The prompt is a **hybrid template** that needs customization for each phase/sprint.

📊 Component Breakdown

Universal Components (40%)

These sections work across all phases and sprints:

1. Systematic Validation Process ✅


markdown

```
Step 1: Analyze Current State
Step 2: Prisma Schema Validation
Step 3: Fix TypeScript Errors - Systematic Approach
Step 4: Implement Fixes
Step 5: Test Compilation
Step 6: Test Backend Startup
Step 7: Test API Endpoints
Step 8: Test Sprint Features
Step 9: Test Frontend Connection
Step 10: Document Changes
```

Why Universal:

- This process works for any validation scenario
- Order of operations is logically sound
- Can be applied to any sprint's features
- Framework-agnostic (TypeScript, Prisma, Express)

Reusability: 100% - Use this exact process every time

---

2. Error Categorization Framework ✅


markdown

```
Priority Order:
- Critical Path Files (Must Fix) - Blocks server startup
- Non-Critical Path Files (Can Disable) - Future features
```

Why Universal:

- Prioritization logic applies to any codebase
- Critical vs non-critical distinction always valid
- Helps make pragmatic decisions under pressure

Reusability: 100% - Always categorize by impact

---

3. Common TypeScript Fix Patterns ✅


markdown

```
Error: Type '{}' is missing properties → Use Partial<T>
Error: Type 'null' not assignable → Use undefined
Error: Property 'X' does not exist → Check schema vs code
Error: Argument type mismatch → Check type definitions
```

Why Universal:

- TypeScript errors follow patterns
- Same fix patterns apply across different contexts
- Reference guide for any TypeScript issue

Reusability: 100% - Keep as reference guide

---

4. Testing Validation Pattern ✅


markdown

```
1. Test compilation (tsc --noEmit)
2. Test server startup (npm run dev)
3. Test health endpoint (curl /health)
4. Test feature endpoints (curl with auth)
5. Test frontend connection
```

Why Universal:

- Testing pyramid applies to all features
- Bottom-up validation catches issues early
- Same pattern regardless of feature set

Reusability: 100% - Standard testing approach

---

5. Documentation Standards ✅


markdown

```
Report Template:
- Summary of changes
- Files modified
- Test results
- Remaining issues
- Recommendations
```

Why Universal:

- Documentation format applies to any work
- Stakeholders always need same information
- Future you needs to understand what happened

Reusability: 100% - Use for all validation work

Context-Specific Components (60%)

These sections are tied to current platform state and Sprint 1:

1. Specific File References 🔵

markdown

```
Files to check:
- backend/src/services/auth.ts
- backend/src/services/CustomerService.ts
- backend/src/services/plan.ts
- backend/src/services/material.ts
```

Why Context-Specific:

- These files exist in current codebase
- Sprint 6-7 will have different files (PlanManagementService.ts)
- Sprint 8-9 will have different files (MaterialInventoryService.ts)

How to Adapt: Update file list for each sprint:

Sprint 2-3 (Customer UI):

markdown

```
Files to check:
- frontend/src/components/customers/CustomerList.tsx
- frontend/src/components/customers/CustomerForm.tsx
- frontend/src/contexts/CustomerContext.tsx
- frontend/src/api/customerApi.ts
```

Sprint 6-7 (Plan Management):


markdown

```
Files to check:
- backend/src/services/plan/PlanService.ts (refactored)
- backend/src/services/plan/PlanTemplateService.ts (new)
- backend/src/controllers/PlanController.ts (updated)
- frontend/src/components/plans/PlanManager.tsx (new)
```

---

2. Sprint-Specific Features 🔵


markdown

```
Test Sprint 1 Security Features:
- JWT_SECRET validation
- Seed security (no hardcoded passwords)
- Security headers (8 HTTP headers)
```

Why Context-Specific:

- These tests are Sprint 1 deliverables
- Each sprint has different features to validate

How to Adapt:

Sprint 2-3 (Customer UI):


markdown

```
Test Sprint 2-3 Customer Features:
- Customer list with search/filter
- Customer create form validation
- Customer edit/update functionality
- Contact management CRUD
- Pricing tier management
- External ID mapping
```

Sprint 6-7 (Plan Management):


markdown

```
Test Sprint 6-7 Plan Features:
- Plan template import
- Plan option combinations
- Plan pricing calculations
- Plan template versioning
- Plan BOM (Bill of Materials) generation
```

Sprint 11-14 (Learning System):


markdown

```
Test Sprint 11-14 Learning Features:
- Variance capture (predicted vs actual)
- Pattern detection accuracy
- Confidence scoring
- Auto-apply workflow
- Learning dashboard visualization
```

---

3. Known Issues List 🔵


markdown

Known Issues:
- Plan service has schema mismatches (18 errors)
- Material service has schema mismatches (25+ errors)
- Expected: customerId field → Doesn't exist
- Expected: MaterialPricing model → Doesn't exist

Why Context-Specific:

- These are current platform-specific issues
- Different sprints have different issues

How to Adapt:

Sprint 2 (After Customer API complete):


markdown

Known Issues:
- None - Customer API fully functional
- Frontend needs state management implementation
- API client needs error handling improvements

Sprint 6 (Starting Plan Management):


markdown

Known Issues:
- Plan schema needs refactoring (customerId, planNumber fields)
- Plan service expects different relations
- Need to design plan template versioning strategy
- BOM calculation logic not implemented

Sprint 11 (Starting Learning System):


markdown

```
Known Issues:
- Variance capture schema needs implementation
- Pattern detection algorithms not defined
- Confidence scoring methodology needs design
- Historical data insufficient for ML training
```

---

4. Database Schema Expectations 🔵


markdown

```
Check Prisma schema:
- Does Plan model have customerId?
- Does Material have pricing relation?
- Does MaterialPricing model exist?
```

Why Context-Specific:

- Schema evolves across sprints
- Different features require different models

How to Adapt:

Sprint 6-7 (Plan Management Implementation):


markdown

```
Check Prisma schema:
- Plan model fields: code, name, sqft, bedrooms, bathrooms
- PlanElevation model with relation to Plan
- PlanOption model with conditional logic fields
- PlanTemplateItem for BOM
- Relations: Plan → PlanElevations, Plan → PlanOptions
```

Sprint 11-14 (Learning System Implementation):


markdown

```
Check Prisma schema:
- TakeoffLineItem: quantityPredicted, quantityActual, variance fields
- VariancePattern model with confidence scoring
- VarianceReview model for approval workflow
- Relations: VariancePattern → TakeoffLineItems
- Enum: PatternLevel (PLAN_SPECIFIC, CROSS_PLAN, etc.)
```

---

5. Tech Stack Configuration 🔵


markdown

```
Platform Details:
- React 19, Node.js, Express, Prisma, PostgreSQL, TypeScript
- Backend port: 3001
- Frontend port: 5173
- Database port: 5433
```

Why Context-Specific:

- Tech stack can change
- Ports might change
- Different environments (dev, staging, prod)

How to Adapt:

If using different stack:


markdown

```
Platform Details:
- Next.js 14, tRPC, Prisma, PostgreSQL, TypeScript
- Backend: /api routes (same server as frontend)
- Frontend/Backend port: 3000 (unified)
- Database port: 5432 (standard PostgreSQL)
```

If deploying to production:


markdown

```
Platform Details:
- Environment: Production (Vercel + Railway)
- Backend: https://api.mindflow.app
- Frontend: https://app.mindflow.app
- Database: Railway PostgreSQL (managed)
```

---

6. Test Credentials 🔵


markdown

```
Test Users:
- admin@mindflow.com / DevPassword123!
- estimator@mindflow.com / DevPassword123!
```

Why Context-Specific:

- Test data changes over time
- Production has different credentials
- Different environments need different data

How to Adapt:

Staging Environment:


markdown

```
Test Users:
- staging-admin@mindflow.com / StagingPass123!
- staging-estimator@mindflow.com / StagingPass123!
- staging-richmond@mindflow.com / RichmondTest!
```

Production Environment:


markdown

Test Users:
- (No test users in production)
- Use real customer credentials
- Validate with Richmond American pilot users

🔄 How to Adapt the Prompt for Each Sprint

Template Adaptation Process

Step 1: Copy the Universal Components (40%) These sections remain unchanged:

- ✅ Systematic validation process
- ✅ Error categorization framework
- ✅ Common TypeScript fix patterns
- ✅ Testing validation pattern
- ✅ Documentation standards

Step 2: Update Context-Specific Components (60%)

A. Update File References


markdown

```
# Sprint 1 (Security Foundation)
Files to check:
- backend/src/services/auth.ts
- backend/src/middleware/securityHeaders.ts
- backend/test-jwt-validation.js

# Sprint 2-3 (Customer UI)
Files to check:
- frontend/src/components/customers/CustomerList.tsx
- frontend/src/components/customers/CustomerForm.tsx
- frontend/src/hooks/useCustomers.ts

# Sprint 6-7 (Plan Management)
Files to check:
- backend/src/services/plan/PlanService.ts (refactored)
- backend/src/services/plan/PlanTemplateService.ts
- frontend/src/components/plans/PlanManager.tsx
```

B. Update Sprint Objectives

markdown

```
# Sprint 1
Sprint 1 Goal: Security Foundation & Critical Fixes
Sprint 1 Features:
1. JWT_SECRET validation
2. Remove hardcoded credentials
3. Security headers

# Sprint 6-7
Sprint 6-7 Goal: Plan Management System
Sprint 6-7 Features:
1. Plan template CRUD
2. Plan option combinations
3. BOM generation
4. Plan pricing calculations
```

C. Update Schema Expectations


markdown

```
# Sprint 1 (Current)
Check Prisma schema:
- User, Customer, Material models exist
- Relations are correct
- Auth fields present

# Sprint 6-7 (Plan Management)
Check Prisma schema:
- Plan model: customerId, planNumber, code, name, sqft
- PlanElevation: relation to Plan
- PlanOption: conditionalLogic field
- PlanTemplateItem: BOM structure
```

D. Update Test Scenarios


markdown

```
# Sprint 1
Test Sprint 1 Security Features:
- JWT_SECRET validation
- Seed security
- Security headers

# Sprint 6-7
Test Sprint 6-7 Plan Features:
- Plan CRUD operations
- Plan template import
- BOM generation accuracy
- Pricing calculation correctness
```

E. Update Known Issues


markdown

```
# Sprint 1 (Current)
Known Issues:
- Plan service schema mismatch (disable for now)
- Material service schema mismatch (disable for now)

# Sprint 6-7 (Starting Plan Work)
Known Issues:
- Plan schema needs refactoring
- Plan template versioning not designed
- Option combination logic not implemented
```

Step 3: Save Adapted Prompt

```
CLAUDE_CODE_VALIDATION_PROMPT_SPRINT_06.md
CLAUDE_CODE_VALIDATION_PROMPT_SPRINT_11.md
etc.
```

📋 Sprint-Specific Prompt Examples

Sprint 2-3: Customer UI Validation Prompt

Context-Specific Sections to Update:

markdown

Current Sprint Context

Sprint: Sprint 2-3 - Customer Management UI
Focus: Build frontend components for customer CRUD operations
Dependencies: Backend customer API (Sprint 1 ✓)

Files to Validate

Frontend Components
- frontend/src/components/customers/CustomerList.tsx
- frontend/src/components/customers/CustomerForm.tsx
- frontend/src/components/customers/CustomerDetail.tsx
- frontend/src/components/customers/ContactManager.tsx
- frontend/src/components/customers/PricingTierManager.tsx

Frontend Hooks & Context
- frontend/src/hooks/useCustomers.ts
- frontend/src/contexts/CustomerContext.tsx
- frontend/src/api/customerApi.ts

Backend (Should Already Work)
- backend/src/routes/customer.ts
- backend/src/controllers/CustomerController.ts
- backend/src/services/CustomerService.ts

Sprint 2-3 Test Scenarios

Frontend Tests
1. **Customer List Display**
   - Fetches customers from API
   - Displays in table/card format
   - Search filter works
   - Pagination works

2. **Customer Creation**
   - Form validation works
   - Successful creation shows success message
   - New customer appears in list
   - Error handling displays properly

3. **Customer Editing**

- Loads existing customer data
- Updates save correctly
- Optimistic updates work
- Error states handled

4. **Contact Management**
- Add contact modal works
- Edit contact works
- Delete contact shows confirmation
- Primary contact toggle works

5. **State Management**
- Customer context provides data to all components
- Loading states display correctly
- Error states display correctly
- Cache invalidation works on mutations

Expected Outcomes Sprint 2-3

- ✅ Customer list page loads and displays all customers
- ✅ Search and filter work on customer list
- ✅ Create customer form validates and submits
- ✅ Edit customer form pre-fills and updates
- ✅ Delete customer shows confirmation and removes from list
- ✅ Contact CRUD operations work within customer detail
- ✅ Loading states show during API calls
- ✅ Error messages display for failed operations
- ✅ Navigation between pages works

Sprint 6-7: Plan Management Validation Prompt

Context-Specific Sections to Update:

markdown

Current Sprint Context

Sprint: Sprint 6-7 - Plan Management System
Focus: Implement plan template CRUD and BOM generation
Dependencies: Database schema refactored (Sprint 5 ✓)

Files to Validate

Backend Services (Refactored)
- backend/src/services/plan/PlanService.ts (NEW)
- backend/src/services/plan/PlanTemplateService.ts (NEW)
- backend/src/services/plan/BOMService.ts (NEW)
- backend/src/controllers/PlanController.ts (UPDATED)

Frontend Components (New)
- frontend/src/components/plans/PlanList.tsx
- frontend/src/components/plans/PlanForm.tsx
- frontend/src/components/plans/PlanDetail.tsx
- frontend/src/components/plans/TemplateManager.tsx
- frontend/src/components/plans/BOMViewer.tsx

Database Schema (Must Match Code)
- prisma/schema.prisma:
  - Plan model: customerId, planNumber, code, name, sqft, bedrooms, bathrooms
  - PlanElevation model with relation to Plan
  - PlanOption model with conditionalLogic fields
  - PlanTemplateItem model for BOM

Sprint 6-7 Test Scenarios

Plan CRUD Operations
1. **Create Plan**
   - Validates required fields (code, name, customerId)
   - Associates with customer
   - Creates with initial template
   - Returns complete plan object with relations

2. **List Plans**
   - Returns all plans with customer info
   - Filters by customer
   - Searches by code or name

- Paginates correctly

3. **Update Plan**
   - Updates plan details
   - Preserves relations
   - Updates timestamps
   - Validates changes

4. **Delete Plan**
   - Checks for dependencies (jobs, takeoffs)
   - Cascade deletes elevations/options if allowed
   - Returns appropriate error if has dependencies

Plan Template Operations
1. **BOM Generation**
   - Calculates materials from template
   - Applies quantities correctly
   - Includes all material categories
   - Respects conditional logic

2. **Template Import**
   - Imports from Excel/CSV
   - Validates material references
   - Creates template items
   - Reports errors clearly

Frontend Integration
1. **Plan Manager UI**
   - Lists all plans
   - Create/Edit forms work
   - Detail view shows BOM
   - Template editor functional

Schema Validation Sprint 6-7

Check that Prisma schema includes:
```typescript
model Plan {
  id            String          @id @default(uuid())
  customerId    String          // ← NEW: Required for plan service
  customer      Customer        @relation(fields: [customerId], references: [id])
```

```
  planNumber       String          @unique // ← NEW: Business identifier
  code             String          @unique
  name             String
  description      String?         // ← NEW: Required by plan service
  sqft             Int?
  bedrooms         Int?
  bathrooms        Decimal?

  // Relations
  elevations       PlanElevation[]
  options          PlanOption[]
  templateItems    PlanTemplateItem[]

  // ... other fields
}

model PlanElevation {
  id               String          @id @default(uuid())
  planId           String
  plan             Plan            @relation(fields: [planId], references: [id])
  elevation        String          // A, B, C, D
  // ... other fields
}

model PlanOption {
  id               String          @id @default(uuid())
  planId           String
  plan             Plan            @relation(fields: [planId], references: [id])
  name             String
  category         String
  conditionalLogic Json?           // NEW: For option dependencies
  // ... other fields
}
```

Expected Outcomes Sprint 6-7

- ✅ Plan CRUD operations fully functional
- ✅ Plans associated with customers correctly
- ✅ BOM generation works with template
- ✅ Option combinations calculate correctly

- ✅ Template import from Excel works
- ✅ Frontend plan manager displays all plans
- ✅ Create/Edit forms validate properly
- ✅ Delete confirms and cascades correctly
- ✅ Performance acceptable with 40+ plans (Richmond)

Sprint 11-14: Learning System Validation Prompt

Context-Specific Sections to Update:


markdown

Current Sprint Context

Sprint: Sprint 11-14 - Learning System (Variance Capture)
Focus: Implement variance tracking and pattern detection
Dependencies: Takeoff system operational (Sprint 9-10 ✓)

Files to Validate

Backend Services (New)
- backend/src/services/learning/VarianceCaptureService.ts (NEW)
- backend/src/services/learning/PatternDetectionService.ts (NEW)
- backend/src/services/learning/ConfidenceScoreService.ts (NEW)
- backend/src/controllers/LearningController.ts (NEW)

Frontend Components (New)
- frontend/src/components/learning/VarianceDashboard.tsx
- frontend/src/components/learning/PatternReviewModal.tsx
- frontend/src/components/learning/LearningInsights.tsx
- frontend/src/components/takeoff/VarianceInputField.tsx (UPDATED)

Database Schema (Must Include Learning Fields)
- prisma/schema.prisma:
  - TakeoffLineItem: quantityPredicted, quantityActual, variance fields
  - VariancePattern model with confidence scoring
  - VarianceReview model for approval workflow
  - PatternLevel enum (PLAN_SPECIFIC, CROSS_PLAN, etc.)

Sprint 11-14 Test Scenarios

Variance Capture
1. **Record Actual Quantities**
   - Captures actual vs predicted quantities
   - Calculates variance automatically
   - Stores variance reason (dropdown + free text)
   - Associates with correct takeoff line item

2. **Variance Reasons**
   - Categorizes by type (MATERIAL, LABOR, WASTE, etc.)
   - Allows free-form notes
   - Timestamps capture moment
   - Links to job/plan context

Pattern Detection

1. **Detect Patterns**
   - Runs nightly analysis
   - Identifies systematic variances (>5 occurrences)
   - Calculates average variance and std deviation
   - Assigns confidence score (0.0-1.0)

2. **Pattern Classification**
   - PLAN_SPECIFIC: Variance in single plan
   - CROSS_PLAN: Variance across multiple plans
   - COMMUNITY: Variance in specific community
   - BUILDER: Variance across builder's projects
   - REGIONAL: Variance in geographic region

3. **Confidence Scoring**
   - Based on: sample size, variance consistency, data quality
   - Low confidence (0.0-0.5): Flag for review
   - Medium confidence (0.5-0.8): Review recommended
   - High confidence (0.8-1.0): Auto-apply candidate

Pattern Review Workflow

1. **Review Interface**
   - Shows pattern details (avg variance, occurrences, etc.)
   - Displays supporting data (which jobs, when, etc.)
   - Provides APPROVE/REJECT/MODIFY options
   - Captures reviewer reasoning

2. **Auto-Apply Logic**
   - Only patterns with confidence > 0.9 auto-apply
   - Only after human approval
   - Updates plan templates automatically
   - Versions template changes
   - Logs all auto-applied changes

Frontend Dashboards

1. **Variance Dashboard**
   - Shows variance trends over time
   - Highlights patterns detected
   - Displays patterns pending review
   - Shows auto-applied patterns

2. **Learning Insights**
   - Material waste trends
   - Estimation accuracy by plan
   - Cost variance over time
   - Learning system effectiveness metrics

Schema Validation Sprint 11-14

Check that Prisma schema includes:
```typescript
model TakeoffLineItem {
  // ... existing fields ...

  // Learning-First Fields (NEW)
  quantityPredicted     Float?            // From template
  quantityActual        Float?            // From job completion
  variance              Float?            // Actual - Predicted
  variancePercent       Float?            // (Variance / Predicted) * 100
  varianceReason        String?           // Why it varied
  varianceCategory      VarianceCategory?  // MATERIAL, LABOR, WASTE, etc.
  predictionConfidence  Float?            // 0.0 - 1.0
  predictionVersion     String?           // Which model version
  actualCapturedAt      DateTime?         // When actual was recorded

  variancePatterns      VariancePattern[]
}

model VariancePattern {
  id                  String          @id @default(uuid())
  patternType         PatternType     // SYSTEMATIC, SEASONAL, etc.
  patternLevel        PatternLevel    // PLAN_SPECIFIC, CROSS_PLAN, etc.
  patternDescription  String

  // Statistical Data
  occurrenceCount     Int             // Times observed
  averageVariance     Float           // Average variance
  stdDeviation        Float           // Standard deviation
  confidenceScore     Float           // 0.0 - 1.0

  // Context
```

```
  materialCategory      String?
  planId                String?
  customerId            String?
  communityId           String?

  // Automation Status
  status                PatternStatus      // DETECTED, UNDER_REVIEW, APPROVED, etc.
  detectedAt            DateTime           @default(now())
  reviewedAt            DateTime?
  reviewedBy            String?            // User ID
  appliedAt             DateTime?

  // Relations
  takeoffLineItems      TakeoffLineItem[]
  reviews               VarianceReview[]
}

model VarianceReview {
  id                    String             @id @default(uuid())
  variancePatternId     String
  variancePattern       VariancePattern    @relation(fields: [variancePatternId], references: [id])
  reviewedBy            String             // User ID
  reviewedAt            DateTime           @default(now())
  decision              ReviewDecision     // APPROVE, REJECT, MODIFY
  decisionReason        String
  modifiedConfidence    Float?
  modifiedVariance      Float?
}

enum PatternLevel {
  PLAN_SPECIFIC
  CROSS_PLAN
  COMMUNITY
  BUILDER
  REGIONAL
}

enum PatternStatus {
  DETECTED
  UNDER_REVIEW
  APPROVED
```

```
  APPLIED
  REJECTED
}
```

Expected Outcomes Sprint 11-14

- ✅ Variance capture works on all takeoff line items
- ✅ Variance reasons categorize correctly
- ✅ Pattern detection runs nightly and finds patterns
- ✅ Confidence scoring assigns appropriate scores
- ✅ Review workflow functional (approve/reject/modify)
- ✅ Auto-apply only for high-confidence approved patterns
- ✅ Template versioning tracks all changes
- ✅ Dashboard displays variance trends
- ✅ Learning insights show system effectiveness
- ✅ Historical data accumulates for ML training
- ✅ Performance acceptable with 1000+ variance records

🎯 Creating a Universal Validation Template

Here's how to create a reusable template that adapts to any sprint:

Universal Validation Template Structure

markdown

Claude Code: [PROJECT_NAME] Validation Prompt

Sprint: [SPRINT_NUMBER] - [SPRINT_NAME]
Phase: [PHASE_NUMBER] - [PHASE_NAME]
Date: [DATE]
Branch: [GIT_BRANCH]

🎯 Mission

[DESCRIBE SPRINT OBJECTIVES AND WHAT NEEDS VALIDATION]

📊 Current Platform Status

✅ What Works
[LIST WORKING FEATURES FROM PREVIOUS SPRINTS]

❌ What's Broken
[LIST KNOWN ISSUES TO FIX THIS SPRINT]

🎯 Goal State
[DESCRIBE DESIRED END STATE AFTER VALIDATION]

🔍 Step-by-Step Validation Process

Step 1: Analyze Current State
[KEEP UNIVERSAL - Same for all sprints]

Step 2: [TECHNOLOGY] Validation
[ADAPT - Check sprint-specific technology (Prisma, React, etc.)]

Step 3: Fix Errors - Systematic Approach
[KEEP UNIVERSAL - Error prioritization always same]

Step 4: Implement Fixes
[ADAPT - Sprint-specific fixes]

Step 5: Test Compilation
[KEEP UNIVERSAL - Always test compilation]

Step 6: Test [FEATURE] Startup
[ADAPT - Test sprint's main feature]

Step 7: Test [FEATURE] Endpoints
[ADAPT - Test sprint's API endpoints]

Step 8: Test Sprint [N] Features
[ADAPT - Test this sprint's specific features]

Step 9: Test Integration
[ADAPT - Test integration with previous sprints]

Step 10: Document Changes
[KEEP UNIVERSAL - Documentation format same]

🛠️ Specific File Checklist

[FEATURE CATEGORY] Files

Must be error-free:
- [ ] [PATH_TO_FILE_1]
- [ ] [PATH_TO_FILE_2]

Can be disabled if broken:
- [ ] [PATH_TO_OPTIONAL_FILE]

🔧 Common Fixes Reference

[KEEP UNIVERSAL - TypeScript patterns don't change]

✅ Success Criteria

Critical (Must Pass)
[ADAPT - Sprint-specific success criteria]

Important (Should Pass)
[ADAPT - Sprint-specific nice-to-haves]

🎓 Context for Claude Code

Project: [PROJECT_NAME]
Tech Stack: [LIST_STACK]
Current Sprint: [SPRINT_DETAILS]
Goal: [SPRINT_GOAL]

[ADAPT - Update with sprint-specific context]

🚀 Start Validation Now

Begin with Step 1: Analyze Current State

📊 Reusability Matrix

Component	Universal?	Adaptation Required	Effort to Adapt
Validation Process (10 steps)	✅ Yes	None	0 min
Error Categorization	✅ Yes	None	0 min
TypeScript Fix Patterns	✅ Yes	None	0 min
Testing Pattern	✅ Yes	None	0 min
Documentation Template	✅ Yes	None	0 min
File References	❌ No	Update file paths	5 min
Sprint Objectives	❌ No	Update sprint goals	5 min
Test Scenarios	❌ No	Update feature tests	10 min
Schema Expectations	❌ No	Update schema checks	10 min
Known Issues	❌ No	Update issue list	5 min
Tech Stack Config	🔵 Maybe	Update if changed	2 min
Test Credentials	🔵 Maybe	Update if changed	2 min

Total Adaptation Time per Sprint: ~30-40 minutes

💡 Best Practices for Prompt Maintenance

1. Version Control Your Prompts


bash

```
# Store prompts with sprint numbers
docs/validation-prompts/
├── VALIDATION_PROMPT_SPRINT_01.md
├── VALIDATION_PROMPT_SPRINT_02.md
├── VALIDATION_PROMPT_SPRINT_06.md
├── VALIDATION_PROMPT_SPRINT_11.md
└── VALIDATION_PROMPT_TEMPLATE.md  # Universal template
```

2. Maintain a Changelog


markdown

```
# Validation Prompt Changelog

## Sprint 6-7 (Plan Management)
- Added: Plan schema validation section
- Updated: File references (PlanService.ts)
- Changed: Test scenarios (plan CRUD)
- Removed: Material service references (different sprint)

## Sprint 11-14 (Learning System)
- Added: Learning schema validation
- Added: Variance capture test scenarios
- Added: Pattern detection checks
- Changed: Success criteria (learning metrics)
```

3. Create Sprint-Specific Quick References


markdown

```
# Sprint 6-7 Validation Quick Reference

**Files to Check:**
- backend/src/services/plan/*.ts
- frontend/src/components/plans/*.tsx

**Schema Must Have:**
- Plan.customerId, Plan.planNumber
- PlanElevation, PlanOption, PlanTemplateItem

**Tests Must Pass:**
- Plan CRUD operations
- BOM generation
- Template import
```

4. Document Adaptations

markdown

```
# How This Prompt Was Adapted from Sprint 1

**What Changed:**
1. File references: auth.ts → PlanService.ts
2. Test scenarios: Security tests → Plan tests
3. Schema checks: Auth fields → Plan fields
4. Known issues: Material disabled → Plan refactored

**What Stayed Same:**
1. 10-step validation process
2. Error categorization framework
3. TypeScript fix patterns
4. Testing validation pattern
5. Documentation standards
```

🎯 Conclusion

The Validation Prompt Is:

✅ 40% Universal (Reuse As-Is)

- Validation methodology
- Error categorization
- Fix prioritization
- Testing patterns
- Documentation standards

🔵 60% Context-Specific (Adapt Per Sprint)

- File references
- Sprint objectives
- Test scenarios
- Schema expectations
- Known issues

Recommended Approach:

1. **Keep Master Template**
   - Store universal components
   - Update only when methodology improves
2. **Create Sprint-Specific Copies**
   - Copy template for each sprint
   - Adapt 60% context-specific sections
   - Version control each sprint's prompt
3. **Maintain Adaptation Guide**
   - Document what to change
   - Provide sprint-specific examples
   - Track adaptation time (~30-40 min)
4. **Build Prompt Library**
   - Keep successful prompts
   - Reference for future sprints
   - Learn what works best

Time Investment:

Initial Creation: ~4 hours (done!)
Per-Sprint Adaptation: ~30-40 minutes
Time Savings: Multiple hours of debugging
ROI: Very high (prevents issues, documents decisions)

📚 Related Documents

- `CLAUDE_CODE_VALIDATION_PROMPT.md` - Original Sprint 1 prompt
- `ISSUE_RESOLUTION_SUMMARY.md` - What the prompt helped fix
- `STRATEGIC_ANALYSIS_AND_RECOMMENDATIONS.md` - Sprint planning context

Document Version: 1.0
Last Updated: 2025-11-09
Reusability Assessment: Complete
Recommendation: Create sprint-specific copies, maintain universal template

---

Bottom Line: The validation prompt is a **hybrid template**. Keep the universal components (40%) as your standard methodology, and adapt the context-specific sections (60%) for each sprint. This approach gives you a proven validation process while remaining flexible for different features and phases.