**Absolutely!** You're right on both counts. Let's improve the prompts and eliminate unnecessary AI generation for fields you can auto-populate.

## **Fields You Should Auto-Populate (Don't Ask AI)**

### **Always Auto-Populate These:**

* **Category**: Always "Functional" → Just set it programmatically
* **Test Case ID**: Always blank for auto-assignment → Don't include in prompt/response
* **Mapping Correlation**: You have this from field\_metadata → Auto-populate from backend\_xpath
* **Manual/Automation**: Rule-based logic → "Manual" for Business Validation, "Automation" for Field Validation

### **Simplified AI Generation (5 columns only):**

1. **Type of Validation** (Field/Business + Positive/Negative)
2. **Test Objective** (What the test does)
3. **Request/Response Field** ("Request" or "Response")
4. **Test Steps** (How to execute)
5. **Expected Result** (What should happen)

## **Improved Prompts by Scenario**

### **1. Initial Test Case Generation Prompt**

====CONTEXT

You are generating test cases for API field validation.

FIELD INFORMATION:

- Field Name: {field\_name}

- Field Type: {field\_type}

- Description: {description}

- Backend Path: {backend\_xpath}

- Business Rules: {business\_rules}

JAVA CODE CONTEXT:

{java\_context}

CONVERSATION CONTEXT:

{conversation\_context}

====QUESTION

Generate 2-4 comprehensive test cases for this field covering different validation scenarios.

Output EXACTLY 5 tab-separated columns:

Type of Validation | Test Objective | Request/Response Field | Test Steps | Expected Result

REQUIREMENTS:

- Type of Validation: Must be exactly one of:

• "Field Validation - Positive"

• "Field Validation - Negative"

• "Business Validation - Positive"

• "Business Validation - Negative"

- Request/Response Field: "Request" or "Response"

- Test Objective: Clear, specific description of what is being tested

- Test Steps: Detailed, executable steps

- Expected Result: Specific expected outcome

FOCUS ON:

- Cover both positive and negative scenarios

- Include boundary conditions where relevant

- Consider business logic from the Java code context

- Make test steps actionable and specific

Output ONLY the 5 tab-separated columns, no explanations or headers.

### **2. Question Answering Prompt**

====CONTEXT

You are a test case expert helping a user understand testing for an API field.

FIELD INFORMATION:

{field\_metadata}

EXISTING TEST CASES:

{existing\_cases\_summary}

CONVERSATION CONTEXT:

{conversation\_context}

USER'S QUESTION:

{user\_question}

====QUESTION

Answer the user's question clearly and helpfully. Focus on:

1. Providing accurate information about the field and test cases

2. Explaining testing concepts if asked

3. Clarifying any confusion about the generated test cases

4. Suggesting improvements or alternatives if relevant

Be conversational and helpful. Provide specific details when possible.

IMPORTANT: Provide ONLY a text response. Do NOT generate test case tables.

### **3. Test Case Modification Prompt**

====CONTEXT

You are modifying an existing test case based on user feedback.

ORIGINAL TEST CASE ({tc\_id}):

- Type of Validation: {original\_validation\_type}

- Test Objective: {original\_objective}

- Request/Response Field: {original\_req\_resp}

- Test Steps: {original\_steps}

- Expected Result: {original\_expected}

FIELD INFORMATION:

{field\_metadata}

USER MODIFICATION REQUEST:

{modification\_request}

====QUESTION

Modify the original test case based on the user's feedback. Keep the same structure but update only what the user requested to change.

Output EXACTLY 5 tab-separated columns:

Type of Validation | Test Objective | Request/Response Field | Test Steps | Expected Result

REQUIREMENTS:

- Address the specific feedback provided

- Keep unchanged elements from the original test case

- Maintain consistency with the field requirements

- Make only the changes requested in the feedback

Output ONLY the modified test case row, no explanations.

### **4. Feedback-Based Generation Prompt**

====CONTEXT

You are generating additional test cases based on user feedback.

FIELD INFORMATION:

{field\_metadata}

EXISTING TEST CASES:

{existing\_cases\_summary}

CONVERSATION CONTEXT:

{conversation\_context}

USER FEEDBACK:

{user\_feedback}

FEEDBACK TYPE: {feedback\_type}

====QUESTION

Generate 1-3 additional test cases that address the user's specific feedback and complement the existing test cases.

Output EXACTLY 5 tab-separated columns:

Type of Validation | Test Objective | Request/Response Field | Test Steps | Expected Result

FOCUS ON:

- Address the specific requirements mentioned in the feedback

- Avoid duplicating existing test scenarios

- Complement the existing test coverage

- {specific\_instruction\_based\_on\_feedback\_type}

Output ONLY the test case rows, no explanations or headers.

## **Auto-Population Logic**

### **In your test case parsing, replace manual column handling with:**

def create\_complete\_test\_case(parsed\_columns, field\_metadata, default\_mapping):

"""Create complete test case with auto-populated fields"""

# Extract AI-generated fields (5 columns)

ai\_generated = {

'type\_of\_validation': parsed\_columns[0],

'test\_objective': parsed\_columns[1],

'request\_response\_field': parsed\_columns[2],

'test\_steps': parsed\_columns[3],

'expected\_result': parsed\_columns[4]

}

# Auto-populate standard fields

complete\_test\_case = {

'Category': 'Functional', # Always functional

'Test Case ID': '', # Will be auto-assigned

'Type of Validation': ai\_generated['type\_of\_validation'],

'Test Objective': ai\_generated['test\_objective'],

'Request/Response Field': ai\_generated['request\_response\_field'],

'Test Steps': ai\_generated['test\_steps'],

'Expected Result': ai\_generated['expected\_result'],

'Mapping Correlation': field\_metadata.get('backend\_xpath', default\_mapping),

'Manual/Automation': determine\_automation\_mode(ai\_generated['type\_of\_validation']),

'Field Name': field\_metadata.get('field\_name', 'Unknown'),

'Timestamp': datetime.now().strftime('%Y-%m-%d %H:%M:%S')

}

return complete\_test\_case

def determine\_automation\_mode(validation\_type):

"""Auto-determine automation mode based on validation type"""

return 'Manual' if 'Business Validation' in validation\_type else 'Automation'

## **Benefits of This Approach**

### **Improved AI Performance**

* **Focused generation**: AI only handles complex creative tasks
* **Reduced errors**: Fewer fields to get wrong
* **Better quality**: More focused prompts produce better content
* **Faster responses**: Less for AI to generate

### **More Reliable Results**

* **Consistent metadata**: Auto-populated fields are always correct
* **Standardized format**: No variation in standard fields
* **Rule-based logic**: Automation/Manual determination is deterministic
* **Cleaner parsing**: Only 5 columns to parse instead of 9

### **Enhanced Prompts**

* **More specific**: Each prompt tailored to its exact purpose
* **Better context**: Relevant information for each scenario
* **Clearer instructions**: Focused requirements
* **Examples included**: Where helpful for complex scenarios

### **Easier Maintenance**

* **Simpler parsing**: Less complex column handling
* **Centralized rules**: Business logic in one place
* **Easier updates**: Change rules without prompt modifications
* **Better testing**: Can test AI generation and auto-population separately

This approach gives you the best of both worlds: AI creativity where needed, and reliable automation for standard fields. The prompts become more focused and effective, while your system becomes more predictable and maintainable.

Next

Great points! Here are several improvements to address these specific issues:

## \*\*1. Test Case Identification Improvements\*\*

### \*\*Problem\*\*: AI doesn’t properly identify which test case to modify

### \*\*Solutions\*\*:

#### \*\*A. Enhanced Test Case Display\*\*

Instead of truncated display, show full context:

```python

Def display\_test\_cases\_with\_full\_context(cases: List[Dict], show\_details: bool = True):

“””Display test cases with complete information for easy reference”””

If not cases:

Print(“No test cases to display.”)

Return

For i, case in enumerate(cases):

Tc\_id = case.get(“Test Case ID”, “N/A”)

Status = case.get(“Status”, “pending”)

Field\_name = case.get(“Field Name”, “Unknown”)

Val\_type = case.get(“Type of Validation”, “N/A”)

Objective = case.get(“Test Objective”, “N/A”) # FULL objective, no truncation

Steps = case.get(“Test Steps”, “N/A”)

Expected = case.get(“Expected Result”, “N/A”)

# Status icon

Status\_icons = {“approved”: “✅”, “rejected”: “❌”, “pending”: “📋”}

Icon = status\_icons.get(status, “📋”)

Print(f”\n{icon} {tc\_id} [{field\_name}] – {val\_type}”)

Print(f”📋 OBJECTIVE: {objective}”) # Full objective visible

Print(f”🔧 STEPS: {steps}”)

Print(f”✅ EXPECTED: {expected}”)

Print(“-“ \* 80) # Clear separator

```

#### \*\*B. Better Test Case Context in Modification Prompts\*\*

```

====CONTEXT

You are modifying a specific test case. Here is the COMPLETE context:

CURRENT FIELD: {field\_name}

ALL EXISTING TEST CASES FOR REFERENCE:

{all\_test\_cases\_with\_full\_details}

SPECIFIC TEST CASE TO MODIFY:

ID: {target\_tc\_id}

Type: {original\_validation\_type}

Objective: {original\_objective}

Steps: {original\_steps}

Expected: {original\_expected}

Request/Response: {original\_req\_resp}

USER MODIFICATION REQUEST:

{modification\_request}

====QUESTION

The user wants to modify {target\_tc\_id} specifically.

Based on their request “{modification\_request}”, modify ONLY the {target\_tc\_id} test case.

Keep all other test cases unchanged. Focus on the specific changes requested for {target\_tc\_id}.

Output EXACTLY 5 tab-separated columns for the MODIFIED version of {target\_tc\_id}:

Type of Validation | Test Objective | Request/Response Field | Test Steps | Expected Result

```

## \*\*2. Prompt Improvements to Reduce Errors\*\*

### \*\*A. Add Explicit Validation Rules\*\*

```

VALIDATION REQUIREMENTS:

* + Type of Validation: MUST be exactly one of these 4 options (copy exactly):

“Field Validation – Positive”

“Field Validation – Negative”

“Business Validation – Positive”

“Business Validation – Negative”

* + Request/Response Field: MUST be exactly “Request” or “Response”

- Test Objective: Must be 10-80 characters, specific and actionable

- Test Steps: Must be detailed enough for manual execution

- Expected Result: Must be specific, measurable outcome

FORMATTING RULES:

- Use single TAB character between columns (not spaces)

- No quotes around text unless part of the content

- No extra newlines or formatting

- No headers or explanations in output

```

### \*\*B. Add Domain-Specific Context\*\*

```

TESTING CONTEXT:

- This is API field validation testing

- Focus on data validation, format checking, and business rules

- Consider edge cases: empty values, null, boundary conditions

- Positive tests verify valid data is accepted

- Negative tests verify invalid data is properly rejected

- Business validation tests check business logic rules

- Field validation tests check format, length, type constraints

COMMON PATTERNS TO AVOID:

- Vague test objectives like “Test field validation”

- Generic steps like “Send request and check response”

- Unclear expected results like “Should work correctly”

```

### \*\*C. Few-Shot Examples in Prompts\*\*

```

EXAMPLE OUTPUT:

Field Validation – Positive Test valid email format acceptance Request Send API request with valid email format ([user@domain.com](mailto:user@domain.com)) System accepts email and returns success response

Field Validation – Negative Test invalid email format rejection Request Send API request with invalid email format (invalid-email) System rejects email and returns validation error message

Business Validation – Positive Test email uniqueness for new registration Request Send registration request with unique email not in system System accepts registration and creates new user account

```

## \*\*3. Enhanced Field Context Prompts\*\*

### \*\*A. Better Field Metadata Integration\*\*

```

====CONTEXT

FIELD DETAILED ANALYSIS:

- Field Name: {field\_name}

- Data Type: {data\_type}

- Required: {is\_required}

- Max Length: {max\_length}

- Format Rules: {format\_constraints}

- Business Rules: {business\_rules}

- Related Fields: {dependent\_fields}

EXTRACTED BUSINESS LOGIC:

{java\_validation\_logic}

FIELD USAGE PATTERNS:

{usage\_patterns\_from\_code}

```

### \*\*B. Context-Aware Test Generation\*\*

```

SMART TEST GENERATION RULES:

- For email fields: Include format validation, domain validation, length checks

- For postal codes: Include country-specific formats, length validation

- For phone numbers: Include format validation, country codes, length checks

- For dates: Include format validation, range checks, business date rules

- For Ids: Include uniqueness, format, reference validation

- For amounts: Include decimal precision, range validation, currency handling

Based on the field type “{field\_type}” and business context, generate appropriate test scenarios.

```

## \*\*4. Error Prevention Techniques\*\*

### \*\*A. Structured Response Validation\*\*

```

RESPONSE VALIDATION CHECKLIST:

Before outputting your response, verify:

1. Exactly 5 columns separated by TAB characters

2. Type of Validation uses exact text from allowed options

3. Request/Response Field is exactly “Request” or “Response”

4. Test Objective is specific and actionable

5. Test Steps are detailed and executable

6. Expected Result is measurable and specific

7. No extra formatting, headers, or explanations included

If any validation fails, correct it before outputting.

```

### \*\*B. Anti-Pattern Instructions\*\*

```

DO NOT:

- Use generic test objectives like “Test field” or “Validate input”

- Include vague steps like “Test the field” or “Check validation”

- Give unclear expected results like “Should work” or “Error occurs”

- Add explanatory text before or after the test case data

- Use pipe (|) characters instead of tabs

- Include headers or formatting in the output

- Generate duplicate or nearly identical test cases

```

## \*\*5. Modification-Specific Improvements\*\*

### \*\*A. Clear Modification Instructions\*\*

```

MODIFICATION GUIDELINES:

- ONLY change what the user specifically requested

- If user says “make it test Canadian postal codes”, change test objective, steps, and expected result accordingly

- If user says “make it negative”, change validation type and adjust logic accordingly

- If user says “add boundary testing”, enhance the existing test with boundary conditions

- Keep the same validation type unless explicitly asked to change it

- Preserve the original intent while incorporating the requested changes

BEFORE MODIFYING:

1. Identify which specific aspect the user wants changed

2. Determine if this requires changing validation type, objective, steps, or expected result

3. Make targeted changes while preserving everything else

```

### \*\*B. Context Preservation\*\*

```

MODIFICATION CONTEXT:

Original test case was: {original\_objective}

User wants: {modification\_request}

This means: {interpreted\_change}

Your modified test case should:

- Address the user’s specific request

- Maintain consistency with the field requirements

- Keep the same validation approach unless explicitly changed

- Preserve any aspects not mentioned in the modification request

```

## \*\*6. Display Improvements for CLI\*\*

### \*\*A. Comprehensive Test Case Display\*\*

```python

Def display\_comprehensive\_test\_cases(cases: List[Dict]):

“””Display test cases with complete information and modification guidance”””

If not cases:

Print(“📝 No test cases to display.”)

Return

Print(f”\n{‘=’\*100}”)

Print(f”📋 TEST CASES SUMMARY ({len(cases)} total)”)

Print(f”{‘=’\*100}”)

For case in cases:

Tc\_id = case.get(“Test Case ID”, “N/A”)

Status = case.get(“Status”, “pending”)

Val\_type = case.get(“Type of Validation”, “N/A”)

Objective = case.get(“Test Objective”, “N/A”) # FULL objective

Steps = case.get(“Test Steps”, “N/A”)

Expected = case.get(“Expected Result”, “N/A”)

Status\_icons = {“approved”: “✅ APPROVED”, “rejected”: “❌ REJECTED”, “pending”: “📋 PENDING”}

Status\_display = status\_icons.get(status, “📋 PENDING”)

Print(f”\n🆔 {tc\_id} – {status\_display}”)

Print(f”🎯 TYPE: {val\_type}”)

Print(f”📝 OBJECTIVE: {objective}”)

Print(f”🔧 STEPS: {steps}”)

Print(f”✅ EXPECTED: {expected}”)

Print(f”💡 TO MODIFY: \”Modify {tc\_id} to [your changes]\””)

Print(“-“ \* 100)

Print(f”\n💬 MODIFICATION EXAMPLES:”)

Print(f” • \”Modify {cases[0].get(‘Test Case ID’, ‘TC\_001’)} to test Canadian postal codes\””)

Print(f” • \”Change {cases[0].get(‘Test Case ID’, ‘TC\_001’)} to be a negative test\””)

Print(f” • \”Update {cases[0].get(‘Test Case ID’, ‘TC\_001’)} to include boundary value testing\””)

```

These improvements will significantly reduce AI errors, make test case identification much clearer, and provide users with the full context they need to make informed modification requests.

Next

Excellent points! Here are additional improvements for both prompts and intent parsing:

## \*\*Additional Prompt Improvements\*\*

### \*\*1. Contextual Constraint Prompting\*\*

```

FIELD-SPECIFIC CONSTRAINTS:

{dynamic\_field\_constraints}

BUSINESS CONTEXT AWARENESS:

- If financial field: Include currency, precision, regulatory compliance tests

- If personal data field: Include privacy, format, validation tests

- If ID field: Include uniqueness, reference integrity, format tests

- If date/time field: Include timezone, format, business date rules

- If address field: Include geocoding, format validation, completeness tests

COVERAGE REQUIREMENTS:

Generate tests to ensure:

✓ Happy path scenarios (2 positive tests minimum)

✓ Error scenarios (2 negative tests minimum)

✓ Boundary conditions (where applicable)

✓ Business rule validation (based on field context)

```

### \*\*2. Dynamic Test Depth Based on Field Complexity\*\*

```

COMPLEXITY ANALYSIS for {field\_name}:

- Simple fields (basic validation): Generate 2-3 test cases

- Medium fields (business rules): Generate 3-4 test cases

- Complex fields (multiple constraints): Generate 4-5 test cases

DETECTED COMPLEXITY: {complexity\_level}

RECOMMENDED TEST COUNT: {recommended\_count}

Based on this analysis, generate {recommended\_count} comprehensive test cases.

```

### \*\*3. Anti-Hallucination Instructions\*\*

```

ACCURACY REQUIREMENTS:

- Use ONLY information provided in the field metadata and Java context

- Do NOT invent business rules not present in the context

- If uncertain about a constraint, create a general validation test instead

- Base expected results on actual field requirements, not assumptions

- If Java code shows specific validation logic, align tests with that logic

WHEN IN DOUBT:

- Default to standard field validation patterns

- Use generic but correct validation approaches

- Ask for clarification rather than guessing requirements

```

## \*\*Enhanced Intent Parser with Stop Words Cleanup\*\*

Here’s the improved intent parser that handles natural language cleanup:

```python

Class EnhancedUserIntentParser:

“””Enhanced intent parser with natural language cleanup and better flexibility”””

Def \_\_init\_\_(self):

# ... (existing spaCy setup) ...

# Define comprehensive stop words and filler phrases

Self.stop\_words = {

‘polite\_words’: [‘please’, ‘kindly’, ‘could’, ‘would’, ‘can’, ‘will’, ‘may’],

‘filler\_words’: [‘you’, ‘i’, ‘me’, ‘my’, ‘we’, ‘us’, ‘our’, ‘the’, ‘a’, ‘an’],

‘request\_phrases’: [‘can you’, ‘could you’, ‘would you’, ‘please’, ‘i want’, ‘i need’, ‘i would like’],

‘confirmation\_words’: [‘go ahead’, ‘proceed’, ‘continue’, ‘do it’, ‘yes please’, ‘sure’],

‘question\_starters’: [‘how do i’, ‘how can i’, ‘what should i’, ‘where do i’, ‘when should i’],

‘intensifiers’: [‘really’, ‘very’, ‘quite’, ‘rather’, ‘pretty’, ‘somewhat’, ‘just’, ‘simply’]

}

# Flexible command variations

Self.command\_variations = {

‘generate’: [

‘generate’, ‘create’, ‘make’, ‘build’, ‘produce’, ‘develop’,

‘give me’, ‘show me’, ‘provide’, ‘come up with’

],

‘more’: [

‘more’, ‘additional’, ‘extra’, ‘another’, ‘some more’, ‘a few more’,

‘couple more’, ‘several more’, ‘add more’, ‘need more’

],

‘modify’: [

‘modify’, ‘change’, ‘update’, ‘edit’, ‘alter’, ‘adjust’, ‘fix’,

‘improve’, ‘enhance’, ‘revise’, ‘correct’, ‘tweak’

],

‘show’: [

‘show’, ‘display’, ‘list’, ‘see’, ‘view’, ‘look at’, ‘check’,

‘review’, ‘examine’, ‘inspect’

]

}

Def clean\_user\_input(self, user\_input: str) -> str:

“””Clean user input by removing stop words and filler phrases”””

# Convert to lowercase for processing

Cleaned = user\_input.lower().strip()

# Remove common request phrases (order matters – longer phrases first)

Request\_phrases\_ordered = [

‘can you please’, ‘could you please’, ‘would you please’,

‘can you kindly’, ‘could you kindly’, ‘would you kindly’,

‘i would like you to’, ‘i need you to’, ‘i want you to’,

‘please can you’, ‘please could you’, ‘please would you’,

‘can you’, ‘could you’, ‘would you’, ‘will you’,

‘i would like’, ‘i want’, ‘i need’, ‘please’

]

For phrase in request\_phrases\_ordered:

If cleaned.startswith(phrase):

Cleaned = cleaned[len(phrase):].strip()

Break

# Remove trailing politeness

Politeness\_endings = [‘please’, ‘thanks’, ‘thank you’, ‘if possible’, ‘for me’]

For ending in politeness\_endings:

If cleaned.endswith(ending):

Cleaned = cleaned[:-len(ending)].strip()

# Clean up extra whitespace and common filler words at start/end

Words = cleaned.split()

# Remove leading filler words

While words and words[0] in [‘just’, ‘simply’, ‘maybe’, ‘perhaps’, ‘possibly’]:

Words.pop(0)

# Remove trailing filler words

While words and words[-1] in [‘now’, ‘today’, ‘here’, ‘there’]:

Words.pop()

Return ‘ ‘.join(words).strip()

Def extract\_intent\_keywords(self, cleaned\_input: str) -> Dict[str, float]:

“””Extract intent keywords with confidence scores”””

Intent\_scores = {}

Words = cleaned\_input.split()

# Check for each command variation

For intent, variations in self.command\_variations.items():

Score = 0.0

For variation in variations:

Variation\_words = variation.split()

# Exact phrase match (highest score)

If variation in cleaned\_input:

Score = max(score, 1.0)

# Partial word match

Elif any(word in words for word in variation\_words):

Matching\_words = sum(1 for word in variation\_words if word in words)

Score = max(score, matching\_words / len(variation\_words) \* 0.8)

If score > 0:

Intent\_scores[intent] = score

Return intent\_scores

Def parse\_intent(self, user\_input: str) -> tuple:

“””Enhanced intent parsing with natural language cleanup”””

If not user\_input.strip():

Return ‘unknown’, {}

Original\_input = user\_input.strip()

# Step 1: Clean the input

Cleaned\_input = self.clean\_user\_input(user\_input)

# Step 2: Extract test case Ids from original input

Tc\_ids = re.findall(self.tc\_id\_pattern, original\_input.upper())

Mod\_ids = re.findall(self.mod\_id\_pattern, original\_input.upper())

All\_ids = tc\_ids + mod\_ids

# Step 3: Try spaCy parsing on cleaned input

Doc = self.nlp(cleaned\_input.lower())

Matches = self.matcher(doc)

If matches:

# Process matches as before, but with cleaned input

Match\_id, start, end = matches[0]

Label = self.nlp.vocab.strings[match\_id]

# ... (existing match processing logic) ...

Params = {‘tc\_ids’: all\_ids, ‘original\_input’: original\_input}

# Use cleaned input for target extraction

If label == “MODIFY\_TESTCASE”:

Params[‘target’] = original\_input # Keep original for modifications

Else:

Params[‘target’] = cleaned\_input

Return self.\_map\_intent(label), params

# Step 4: Enhanced keyword fallback with cleaned input

Intent\_scores = self.extract\_intent\_keywords(cleaned\_input)

If intent\_scores:

Best\_intent = max(intent\_scores.items(), key=lambda x: x[1])

If best\_intent[1] > 0.6: # Confidence threshold

Return self.\_map\_keyword\_to\_intent(best\_intent[0]), {

‘tc\_ids’: all\_ids,

‘target’: cleaned\_input,

‘original\_input’: original\_input,

‘confidence’: best\_intent[1],

‘fallback’: ‘keyword\_enhanced’

}

# Step 5: Final fuzzy matching

Return self.\_fuzzy\_match\_fallback(cleaned\_input, all\_ids, original\_input)

Def \_map\_keyword\_to\_intent(self, keyword\_intent: str) -> str:

“””Map keyword intentions to actual intents”””

Keyword\_to\_intent = {

‘generate’: ‘generate’,

‘more’: ‘regenerate’,

‘modify’: ‘improve’,

‘show’: ‘show\_pending’

}

Return keyword\_to\_intent.get(keyword\_intent, ‘unknown’)

Def handle\_natural\_language\_variations(self, user\_input: str) -> str:

“””Handle common natural language variations”””

# Common variations mapping

Variations = {

‘give me some tests’: ‘generate test cases’,

‘make some more’: ‘generate more test cases’,

‘let me see what we have’: ‘show pending test cases’,

‘are we done’: ‘show status’,

‘what do you think’: ‘show pending test cases’,

‘looks good to me’: ‘approve all’,

‘not quite right’: ‘need modifications’,

‘make it better’: ‘improve test cases’,

‘save everything’: ‘export test cases’

}

User\_lower = user\_input.lower().strip()

For variation, standard\_form in variations.items():

If variation in user\_lower:

Return standard\_form

Return user\_input

```

## \*\*Additional Improvements\*\*

### \*\*4. Smart Context Injection\*\*

```python

Def build\_smart\_context\_prompt(field\_metadata: dict, conversation\_context: str) -> str:

“””Build context with smart field analysis”””

Field\_name = field\_metadata.get(‘field\_name’, ‘Unknown’)

# Detect field patterns

Field\_type = detect\_field\_type(field\_name)

Complexity = analyze\_field\_complexity(field\_metadata)

Context = f”””SMART FIELD ANALYSIS:

Field: {field\_name}

Detected Type: {field\_type}

Complexity Level: {complexity}

RECOMMENDED TEST STRATEGY:

{get\_test\_strategy(field\_type, complexity)}

{standard\_field\_context}

“””

Return context

Def detect\_field\_type(field\_name: str) -> str:

“””Auto-detect field type from name patterns”””

Patterns = {

‘email’: [‘email’, ‘mail’, ‘e-mail’],

‘phone’: [‘phone’, ‘telephone’, ‘mobile’, ‘cell’],

‘postal\_code’: [‘postal’, ‘zip’, ‘postcode’],

‘date’: [‘date’, ‘time’, ‘timestamp’, ‘created’, ‘updated’],

‘id’: [‘id’, ‘identifier’, ‘key’, ‘number’],

‘address’: [‘address’, ‘street’, ‘city’, ‘state’, ‘country’],

‘amount’: [‘amount’, ‘price’, ‘cost’, ‘fee’, ‘balance’, ‘total’]

}

Field\_lower = field\_name.lower()

For ftype, keywords in patterns.items():

If any(keyword in field\_lower for keyword in keywords):

Return ftype

Return ‘generic’

```

### \*\*5. Intent Confidence and Clarification\*\*

```python

Def handle\_low\_confidence\_intent(user\_input: str, confidence: float) -> str:

“””Handle cases where intent confidence is low”””

If confidence < 0.4:

Return f”””I’m not quite sure what you want to do. Your request was: “{user\_input}”

Did you want to:

• Generate test cases? (say ‘generate’ or ‘create tests’)

• Review existing cases? (say ‘show’ or ‘review’)

• Modify a test case? (say ‘modify TC\_001’ or ‘change TC\_002’)

• Export your work? (say ‘export’ or ‘save’)

Just tell me more clearly what you’d like to do!”””

Elif confidence < 0.7:

Suggested\_actions = get\_suggested\_actions(user\_input)

Return f”””I think I understand, but want to be sure. Based on “{user\_input}”, did you want to:

{suggested\_actions}

Please confirm or rephrase!”””

```

### \*\*6. Example Handling\*\*

Test the improved parser with these examples:

```

“Can you please generate more test cases for me?”

→ Cleaned: “generate more test cases”

→ Intent: regenerate

“I would like you to modify TC\_001 to test Canadian postal codes please”

→ Cleaned: “modify TC\_001 to test Canadian postal codes”

→ Intent: improve (with TC\_001 context)

“Could you kindly show me what we have so far?”

→ Cleaned: “show what we have”

→ Intent: show\_pending

“Please go ahead and export everything to Excel”

→ Cleaned: “export everything to Excel”

→ Intent: export

```

These improvements make the system much more natural to use while maintaining accuracy and reducing the need for users to learn specific command syntax.

Next

Perfect! Let's implement Prompt-Based MCP in phases. Here's the roadmap and Phase 1 implementation:

## **Phase Breakdown**

### **Phase 1: Core MCP Infrastructure (Week 1-2)**

* Tool registration system
* Basic prompt-based tool calling
* Response parsing
* 3-4 core tools to prove concept

### **Phase 2: Tool Expansion (Week 3-4)**

* Convert all existing functions to tools
* Enhanced error handling
* Tool chaining capabilities

### **Phase 3: Advanced Orchestration (Week 5-6)**

* Multi-step workflows
* Context awareness
* Learning and adaptation

## **Phase 1 Implementation**

### **1. NEW FILE:** mcp\_tools.py

Create this new file for tool definitions:

from typing import Dict, Any, List, Optional, Callable

from dataclasses import dataclass

from datetime import datetime

import json

@dataclass

class Tool:

"""Tool definition for MCP system"""

name: str

description: str

parameters: Dict[str, Any]

function: Callable

examples: List[str]

category: str = "general"

class MCPToolRegistry:

"""Registry for MCP tools"""

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

self.tools: Dict[str, Tool] = {}

self.categories: Dict[str, List[str]] = {}

def register\_tool(self, tool: Tool):

"""Register a tool in the system"""

self.tools[tool.name] = tool

if tool.category not in self.categories:

self.categories[tool.category] = []

self.categories[tool.category].append(tool.name)

print(f"[MCP] Registered tool: {tool.name}")

def get\_tool(self, name: str) -> Optional[Tool]:

"""Get tool by name"""

return self.tools.get(name)

def get\_tools\_by\_category(self, category: str) -> List[Tool]:

"""Get all tools in a category"""

tool\_names = self.categories.get(category, [])

return [self.tools[name] for name in tool\_names]

def get\_all\_tools(self) -> List[Tool]:

"""Get all registered tools"""

return list(self.tools.values())

def generate\_tool\_schema(self) -> str:

"""Generate tool schema for LLM prompt"""

schema\_parts = []

for category, tool\_names in self.categories.items():

schema\_parts.append(f"\n\*\*{category.upper()} TOOLS:\*\*")

for tool\_name in tool\_names:

tool = self.tools[tool\_name]

schema\_parts.append(f"- {tool.name}({self.\_format\_parameters(tool.parameters)})")

schema\_parts.append(f" Description: {tool.description}")

if tool.examples:

schema\_parts.append(f" Examples: {'; '.join(tool.examples)}")

return "\n".join(schema\_parts)

def \_format\_parameters(self, params: Dict[str, Any]) -> str:

"""Format parameters for schema display"""

if not params:

return ""

param\_strs = []

for name, info in params.items():

param\_type = info.get('type', 'string')

required = " [required]" if info.get('required', False) else " [optional]"

param\_strs.append(f"{name}: {param\_type}{required}")

return ", ".join(param\_strs)

# Global tool registry

tool\_registry = MCPToolRegistry()

### **2. NEW FILE:** mcp\_orchestrator.py

Create the orchestrator that handles tool selection:

import re

import json

from typing import Dict, Any, List, Optional, Tuple

from .mcp\_tools import tool\_registry, Tool

class MCPOrchestrator:

"""Orchestrates tool selection and execution based on user input"""

def \_\_init\_\_(self, client, conversation\_manager=None):

self.client = client

self.conversation\_manager = conversation\_manager

self.execution\_history = []

self.context\_state = {}

def process\_user\_request(self, user\_input: str, session\_context: Dict = None) -> Dict[str, Any]:

"""Main entry point for processing user requests via MCP"""

print(f"[MCP] Processing: {user\_input}")

try:

# Step 1: Analyze request and select tools

tool\_plan = self.\_analyze\_and\_plan(user\_input, session\_context or {})

if not tool\_plan.get('tools'):

return {

'success': False,

'response': "I couldn't understand what you want to do. Could you rephrase that?",

'action\_taken': 'analysis\_failed'

}

# Step 2: Execute tool plan

results = self.\_execute\_tool\_plan(tool\_plan, user\_input)

# Step 3: Format response

return self.\_format\_response(results, user\_input)

except Exception as e:

print(f"[MCP ERROR] {str(e)}")

return {

'success': False,

'response': f"Something went wrong: {str(e)}",

'action\_taken': 'error'

}

def \_analyze\_and\_plan(self, user\_input: str, context: Dict) -> Dict[str, Any]:

"""Analyze user input and create execution plan"""

# Build analysis prompt

prompt = self.\_build\_analysis\_prompt(user\_input, context)

# Get LLM analysis

try:

response = self.client.chat\_completion(prompt)

analysis\_text = response.choices[0].message.content.strip()

# Parse the analysis

tool\_plan = self.\_parse\_tool\_plan(analysis\_text)

print(f"[MCP] Planned tools: {[t['name'] for t in tool\_plan.get('tools', [])]}")

return tool\_plan

except Exception as e:

print(f"[MCP] Analysis failed: {e}")

return {'tools': [], 'reasoning': 'Analysis failed'}

def \_build\_analysis\_prompt(self, user\_input: str, context: Dict) -> str:

"""Build prompt for tool analysis"""

# Get available tools

tool\_schema = tool\_registry.generate\_tool\_schema()

prompt = f"""You are a tool selection AI. Analyze the user's request and determine which tools to use.

AVAILABLE TOOLS:

{tool\_schema}

CURRENT CONTEXT:

- Current field: {context.get('current\_field', 'None')}

- Session state: {context.get('session\_stats', {})}

- Recent actions: {context.get('recent\_actions', [])}

USER REQUEST: "{user\_input}"

Analyze this request and respond in this EXACT format:

ANALYSIS: [Brief explanation of what the user wants]

TOOLS: [List of tool names to use, in order]

PARAMETERS: [JSON object with parameters for each tool]

Example response:

ANALYSIS: User wants to generate test cases for current field

TOOLS: generate\_test\_cases

PARAMETERS: {{"generate\_test\_cases": {{"field\_name": "current", "requirements": "standard validation"}}}}

Your response:"""

return prompt

def \_parse\_tool\_plan(self, analysis\_text: str) -> Dict[str, Any]:

"""Parse LLM analysis into executable plan"""

plan = {'tools': [], 'reasoning': '', 'parameters': {}}

try:

# Extract analysis

analysis\_match = re.search(r'ANALYSIS:\s\*(.+?)(?=TOOLS:)', analysis\_text, re.DOTALL)

if analysis\_match:

plan['reasoning'] = analysis\_match.group(1).strip()

# Extract tools

tools\_match = re.search(r'TOOLS:\s\*(.+?)(?=PARAMETERS:)', analysis\_text, re.DOTALL)

if tools\_match:

tools\_text = tools\_match.group(1).strip()

tool\_names = [t.strip() for t in tools\_text.replace(',', '\n').split('\n') if t.strip()]

for tool\_name in tool\_names:

if tool\_registry.get\_tool(tool\_name):

plan['tools'].append({'name': tool\_name})

# Extract parameters

params\_match = re.search(r'PARAMETERS:\s\*(.+)', analysis\_text, re.DOTALL)

if params\_match:

params\_text = params\_match.group(1).strip()

try:

plan['parameters'] = json.loads(params\_text)

except:

print(f"[MCP] Failed to parse parameters: {params\_text}")

except Exception as e:

print(f"[MCP] Plan parsing error: {e}")

return plan

def \_execute\_tool\_plan(self, plan: Dict, original\_input: str) -> List[Dict]:

"""Execute the planned tools"""

results = []

for tool\_info in plan.get('tools', []):

tool\_name = tool\_info['name']

tool = tool\_registry.get\_tool(tool\_name)

if not tool:

results.append({

'tool': tool\_name,

'success': False,

'error': f"Tool {tool\_name} not found"

})

continue

# Get parameters for this tool

tool\_params = plan.get('parameters', {}).get(tool\_name, {})

# Add context if needed

if 'original\_input' not in tool\_params:

tool\_params['original\_input'] = original\_input

try:

print(f"[MCP] Executing {tool\_name} with params: {tool\_params}")

# Execute the tool

result = tool.function(\*\*tool\_params)

results.append({

'tool': tool\_name,

'success': True,

'result': result,

'params\_used': tool\_params

})

# Update context state

self.\_update\_context\_state(tool\_name, result)

except Exception as e:

print(f"[MCP] Tool {tool\_name} failed: {e}")

results.append({

'tool': tool\_name,

'success': False,

'error': str(e),

'params\_used': tool\_params

})

return results

def \_update\_context\_state(self, tool\_name: str, result: Any):

"""Update orchestrator context based on tool execution"""

# Track recent actions

if 'recent\_actions' not in self.context\_state:

self.context\_state['recent\_actions'] = []

self.context\_state['recent\_actions'].append({

'tool': tool\_name,

'timestamp': datetime.now(),

'success': True

})

# Keep only last 5 actions

if len(self.context\_state['recent\_actions']) > 5:

self.context\_state['recent\_actions'] = self.context\_state['recent\_actions'][-5:]

def \_format\_response(self, results: List[Dict], original\_input: str) -> Dict[str, Any]:

"""Format tool execution results into user response"""

if not results:

return {

'success': False,

'response': "No tools were executed.",

'action\_taken': 'no\_execution'

}

successful\_results = [r for r in results if r['success']]

failed\_results = [r for r in results if not r['success']]

if not successful\_results:

return {

'success': False,

'response': f"All tools failed. Errors: {[r['error'] for r in failed\_results]}",

'action\_taken': 'all\_tools\_failed'

}

# Get the primary result (last successful tool)

primary\_result = successful\_results[-1]['result']

# Format based on result type

if isinstance(primary\_result, dict):

response = primary\_result.get('response', str(primary\_result))

action\_taken = primary\_result.get('action\_taken', 'tool\_executed')

return {

'success': True,

'response': response,

'action\_taken': action\_taken,

'tools\_executed': [r['tool'] for r in successful\_results],

'details': primary\_result

}

else:

return {

'success': True,

'response': str(primary\_result),

'action\_taken': 'tool\_executed',

'tools\_executed': [r['tool'] for r in successful\_results]

}

### **3. MODIFY:** complete\_test\_objective\_core.py

**ADD these imports at the top:**

from .mcp\_tools import tool\_registry, Tool

from .mcp\_orchestrator import MCPOrchestrator

**ADD this method to TestObjectiveGeneratorCore class:**

def register\_as\_mcp\_tools(self):

"""Register core functions as MCP tools"""

# Tool 1: Generate Test Cases

generate\_tool = Tool(

name="generate\_test\_cases",

description="Generate test cases for a specified field",

parameters={

"field\_name": {"type": "string", "required": False, "description": "Field name (uses current field if not specified)"},

"requirements": {"type": "string", "required": False, "description": "Specific requirements or focus areas"}

},

function=self.\_mcp\_generate\_test\_cases,

examples=["generate test cases", "create tests for postal code"],

category="generation"

)

tool\_registry.register\_tool(generate\_tool)

# Tool 2: Process Feedback

process\_feedback\_tool = Tool(

name="process\_feedback",

description="Process user feedback about test cases (questions, modifications, improvements)",

parameters={

"feedback": {"type": "string", "required": True, "description": "User feedback or question"},

"field\_name": {"type": "string", "required": False, "description": "Target field name"}

},

function=self.\_mcp\_process\_feedback,

examples=["change TC\_001 to test Canadian postal codes", "what does TC\_002 validate?"],

category="feedback"

)

tool\_registry.register\_tool(process\_feedback\_tool)

# Tool 3: Get Pending Cases

get\_pending\_tool = Tool(

name="get\_pending\_cases",

description="Show pending test cases that need review",

parameters={

"field\_name": {"type": "string", "required": False, "description": "Filter by field name"}

},

function=self.\_mcp\_get\_pending\_cases,

examples=["show pending", "what do we have", "review test cases"],

category="review"

)

tool\_registry.register\_tool(get\_pending\_tool)

# Tool 4: Export Results

export\_tool = Tool(

name="export\_results",

description="Export approved test cases to Excel file",

parameters={

"format": {"type": "string", "required": False, "description": "Export format (default: excel)"}

},

function=self.\_mcp\_export\_results,

examples=["export", "save to excel", "download results"],

category="export"

)

tool\_registry.register\_tool(export\_tool)

def \_mcp\_generate\_test\_cases(self, field\_name: str = None, requirements: str = "", original\_input: str = "") -> Dict[str, Any]:

"""MCP wrapper for generate\_for\_field"""

# Use current field if no field specified

if not field\_name or field\_name == "current":

if not self.current\_field\_name or not self.field\_contexts.get(self.current\_field\_name):

return {

'success': False,

'response': "No field is currently selected. Please select a field first.",

'action\_taken': 'no\_field\_selected'

}

field\_metadata = self.field\_contexts[self.current\_field\_name]['metadata']

else:

# TODO: In Phase 2, add field lookup by name

return {

'success': False,

'response': f"Field lookup not yet implemented. Please select field first.",

'action\_taken': 'field\_lookup\_pending'

}

try:

success = self.generate\_for\_field(field\_metadata)

if success:

# Get generated cases for display

generated\_cases = self.test\_manager.get\_field\_test\_cases(self.current\_field\_name)

pending\_cases = [case for case in generated\_cases if case.get('Status') == 'pending']

response = f"✅ Generated {len(pending\_cases)} test cases for {field\_metadata.get('field\_name', 'current field')}:\n"

for case in pending\_cases[-3:]: # Show last 3

tc\_id = case.get('Test Case ID', 'N/A')

objective = case.get('Test Objective', 'N/A')

response += f"- {tc\_id}: {objective}\n"

response += f"\nSay 'show pending' to review all cases or 'approve TC\_001' to approve specific ones."

return {

'success': True,

'response': response,

'action\_taken': 'test\_cases\_generated',

'generated\_count': len(pending\_cases)

}

else:

return {

'success': False,

'response': "Failed to generate test cases. Please try again or check the field configuration.",

'action\_taken': 'generation\_failed'

}

except Exception as e:

return {

'success': False,

'response': f"Error generating test cases: {str(e)}",

'action\_taken': 'generation\_error'

}

def \_mcp\_process\_feedback(self, feedback: str, field\_name: str = None, original\_input: str = "") -> Dict[str, Any]:

"""MCP wrapper for process\_user\_feedback"""

# Use current field if no field specified

if not field\_name:

if not self.current\_field\_name:

return {

'success': False,

'response': "No field is currently selected. Please select a field first.",

'action\_taken': 'no\_field\_selected'

}

field\_metadata = self.field\_contexts.get(self.current\_field\_name, {}).get('metadata', {})

field\_metadata['field\_name'] = self.current\_field\_name

else:

# TODO: Phase 2 - field lookup

field\_metadata = {'field\_name': field\_name}

try:

result = self.process\_user\_feedback(feedback, field\_metadata)

return {

'success': not result.get('error'),

'response': result.get('response', 'Feedback processed'),

'action\_taken': result.get('action\_taken', 'feedback\_processed'),

'is\_question': result.get('is\_question', False),

'new\_test\_cases': result.get('new\_test\_cases', []),

'modifications\_created': result.get('modifications\_created', [])

}

except Exception as e:

return {

'success': False,

'response': f"Error processing feedback: {str(e)}",

'action\_taken': 'feedback\_error'

}

def \_mcp\_get\_pending\_cases(self, field\_name: str = None, original\_input: str = "") -> Dict[str, Any]:

"""MCP wrapper for getting pending cases"""

try:

if field\_name:

pending\_cases = self.test\_manager.get\_field\_test\_cases(field\_name)

pending\_cases = [case for case in pending\_cases if case.get('Status') == 'pending']

else:

pending\_cases = self.test\_manager.get\_pending\_cases()

if not pending\_cases:

return {

'success': True,

'response': "No pending test cases to review. Generate some test cases first!",

'action\_taken': 'no\_pending\_cases'

}

response = f"📋 Found {len(pending\_cases)} pending test cases:\n\n"

for case in pending\_cases:

tc\_id = case.get('Test Case ID', 'N/A')

val\_type = case.get('Type of Validation', 'N/A')

objective = case.get('Test Objective', 'N/A')

response += f"{tc\_id} - {val\_type}\n"

response += f" 📝 {objective}\n\n"

response += "💡 Say 'approve TC\_001' to approve specific cases or 'approve all' to approve everything."

return {

'success': True,

'response': response,

'action\_taken': 'pending\_cases\_displayed',

'pending\_count': len(pending\_cases)

}

except Exception as e:

return {

'success': False,

'response': f"Error getting pending cases: {str(e)}",

'action\_taken': 'get\_pending\_error'

}

def \_mcp\_export\_results(self, format: str = "excel", original\_input: str = "") -> Dict[str, Any]:

"""MCP wrapper for export functionality"""

try:

# Auto-complete current field if ready

if self.current\_field\_name and self.is\_field\_ready\_for\_completion():

completion\_result = self.complete\_current\_field()

print(f"[MCP] Auto-completed field: {completion\_result.get('completed\_field')}")

# Check for approved cases

approved\_cases = self.test\_manager.get\_approved\_cases()

if not approved\_cases:

return {

'success': False,

'response': "No approved test cases to export. Please approve some test cases first.",

'action\_taken': 'no\_approved\_cases'

}

# Export

timestamp = datetime.now().strftime("%Y%m%d\_%H%M%S")

filename = f"test\_cases\_{timestamp}.xlsx"

success = self.export\_all\_completed\_fields(filename)

if success:

return {

'success': True,

'response': f"✅ Successfully exported {len(approved\_cases)} test cases to {filename}",

'action\_taken': 'export\_completed',

'filename': filename,

'cases\_exported': len(approved\_cases)

}

else:

return {

'success': False,

'response': "Export failed. Please try again.",

'action\_taken': 'export\_failed'

}

except Exception as e:

return {

'success': False,

'response': f"Export error: {str(e)}",

'action\_taken': 'export\_error'

}

### **4. MODIFY:** complete\_agentic\_test\_generator.py

**ADD at the top with other imports:**

from .mcp\_orchestrator import MCPOrchestrator

**REPLACE the conversational\_interactive\_mode function signature:**

def conversational\_interactive\_mode(generator: TestObjectiveGeneratorCore, field\_loader: FieldMetadataLoader) -> bool:

"""Enhanced conversational mode with MCP orchestration"""

# Initialize conversation components

convo\_mgr = SimpleConversationManager(50)

session = ConversationalSession(convo\_mgr)

# NEW: Initialize MCP orchestrator

orchestrator = MCPOrchestrator(generator.client, convo\_mgr)

# NEW: Register tools

generator.register\_as\_mcp\_tools()

print("🤖 CONVERSATIONAL TEST CASE GENERATOR (MCP Enhanced)")

print("Talk to me naturally! I understand what you want to do using smart tool selection.")

print("=" \* 80)

**REPLACE the main processing loop with:**

while True:

# Show context if available

if session.current\_field:

field\_name = session.current\_field.split('/')[-1]

stats = session.session\_context

status\_text = f"Working on: {field\_name}"

if stats['generated\_count'] > 0:

status\_text += f" | Generated: {stats['generated\_count']} | Approved: {stats['approved\_count']}"

print(f"\n{status\_text}")

user\_input = input("\n💬 You: ").strip()

if not user\_input:

continue

# Check for exit

if user\_input.lower() in ['exit', 'quit', 'bye']:

break

try:

# NEW: Use MCP orchestrator for processing

context = {

'current\_field': session.current\_field,

'session\_stats': session.session\_context,

'recent\_actions': getattr(session, 'recent\_actions', [])

}

# Update generator context if field is selected

if session.current\_field and session.current\_field\_metadata:

generator.current\_field\_name = session.current\_field\_metadata.get('field\_name')

if generator.current\_field\_name not in generator.field\_contexts:

generator.field\_contexts[generator.current\_field\_name] = {

'metadata': session.current\_field\_metadata,

'generation\_timestamp': datetime.now()

}

result = orchestrator.process\_user\_request(user\_input, context)

print(f"🤖 Assistant: {result['response']}")

# Update session stats based on result

if result.get('generated\_count'):

session.update\_stats('generated\_count', result['generated\_count'])

# Add to conversation history

session.add\_interaction(user\_input, result['response'], result.get('action\_taken', 'mcp\_processed'))

except Exception as e:

print(f"🤖 Assistant: Sorry, I encountered an error: {str(e)}")

print("Please try rephrasing your request.")

session.add\_interaction(user\_input, f"Error: {str(e)}", 'error')

return handle\_exit\_flow(generator)

## **Phase 1 Testing**

After implementing Phase 1, test with these commands:

* "generate test cases" → Should trigger generate\_test\_cases tool
* "show me what we have" → Should trigger get\_pending\_cases tool
* "export everything" → Should trigger export\_results tool
* "change TC\_001 to test different data" → Should trigger process\_feedback tool

## **What's Next: Phase 2**

After Phase 1 is working:

1. Add field selection tools (search\_fields, select\_field)
2. Add session management tools (get\_status, switch\_field)
3. Add approval/rejection tools
4. Enhanced error handling and tool chaining
5. Better context management between tools

Would you like me to continue with the complete Phase 1 implementation details, or shall we start testing this first batch?

**Absolutely!** You're 100% right. In Phase 1, I focused on the MCP infrastructure but missed a critical piece - **field selection tools**.

Here's the issue and quick fix:

## **Current Problem**

The generate\_test\_cases tool assumes you already have a current field selected, but there's no MCP tool to select fields yet. This means:

* "generate test cases" will fail with "No field is currently selected"
* Users have no way to select/switch fields through the MCP system

## **Quick Phase 1 Fix**

### **ADD these two tools to the** register\_as\_mcp\_tools() **method:**

# Add after the existing 4 tools:

# Tool 5: Select Field

select\_field\_tool = Tool(

name="select\_field",

description="Select a field to work on by name",

parameters={

"field\_name": {"type": "string", "required": True, "description": "Name of the field to select"}

},

function=self.\_mcp\_select\_field,

examples=["select PostalCode", "work on email field", "use customer name"],

category="field\_management"

)

tool\_registry.register\_tool(select\_field\_tool)

# Tool 6: Show Available Fields

list\_fields\_tool = Tool(

name="list\_available\_fields",

description="List all available fields from the mapping file",

parameters={},

function=self.\_mcp\_list\_fields,

examples=["list fields", "show available fields", "what fields do we have"],

category="field\_management"

)

tool\_registry.register\_tool(list\_fields\_tool)

### **ADD these tool functions to the TestObjectiveGeneratorCore class:**

def \_mcp\_select\_field(self, field\_name: str, original\_input: str = "") -> Dict[str, Any]:

"""MCP wrapper for field selection"""

try:

# This is a simplified version - you'll need to integrate with your field\_loader

# For now, create a basic field metadata structure

field\_metadata = {

'field\_name': field\_name,

'backend\_xpath': f'/{field\_name.lower()}', # Simple default

'description': f'{field\_name} field validation'

}

# Set as current field

self.current\_field\_name = field\_name

self.field\_contexts[field\_name] = {

'metadata': field\_metadata,

'selection\_timestamp': datetime.now(),

'attempts': 1

}

return {

'success': True,

'response': f"✅ Selected field: {field\_name}. You can now generate test cases with 'generate test cases'.",

'action\_taken': 'field\_selected',

'selected\_field': field\_name

}

except Exception as e:

return {

'success': False,

'response': f"Failed to select field {field\_name}: {str(e)}",

'action\_taken': 'field\_selection\_failed'

}

def \_mcp\_list\_fields(self, original\_input: str = "") -> Dict[str, Any]:

"""MCP wrapper for listing available fields"""

try:

# Simplified field list for Phase 1 testing

# In Phase 2, this will integrate with your field\_loader

sample\_fields = [

"PostalCode", "EmailAddress", "PhoneNumber", "CustomerName",

"AccountNumber", "TransactionAmount", "DateOfBirth"

]

response = "📋 Available fields for testing:\n\n"

for i, field in enumerate(sample\_fields, 1):

response += f"{i}. {field}\n"

response += f"\n💡 Say 'select PostalCode' to work on a specific field."

return {

'success': True,

'response': response,

'action\_taken': 'fields\_listed',

'available\_fields': sample\_fields

}

except Exception as e:

return {

'success': False,

'response': f"Error listing fields: {str(e)}",

'action\_taken': 'list\_fields\_error'

}

## **Updated Testing Flow**

Now you can test the complete workflow:

# 1. Start by seeing what's available

"what fields do we have?"

"list available fields"

# 2. Select a field to work on

"select PostalCode"

"work on EmailAddress"

# 3. Generate test cases (now this will work!)

"generate test cases"

"create some tests"

# 4. Review and export

"show pending"

"export results"

## **Why This Quick Fix Works**

1. **Provides field context** - Users can select fields before generating
2. **Shows available options** - Users know what fields they can work with
3. **Maintains MCP architecture** - Everything goes through tool selection
4. **Enables full workflow testing** - Complete end-to-end functionality

In **Phase 2**, we'll properly integrate with your FieldMetadataLoader to get real field data from mapping files instead of these hardcoded samples.

**Try this updated version** - it should give you the complete testing experience for the MCP workflow!