Group 2 (Generic AI) Testing Methodology & Implementation

Strategic Analysis: Internal vs External Approach

Recommended Approach: Hybrid Implementation

After analyzing the requirements, I recommend a **hybrid approach** that provides both internal API integration and external prompt-based testing, with the internal method as primary for data quality and the external as backup for broader accessibility.

Approach 1: Internal API Integration (Primary Method)

Advantages:

- ✅ **Complete data capture**: Full logging of interactions, timing, and metadata
- ✅ **Standardized environment**: Same interface for all participants
- ✅ **Real-time benchmarking**: Cognitive metrics calculated during session
- ✅ **Quality control**: Consistent prompt delivery and response handling
- ✅ **Research validity**: Controlled variables and reliable data collection

Implementation:

- Direct API calls to Claude/ChatGPT from our platform
- Same UI as MENTOR group but with generic AI backend
- Standardized prompts that mimic typical AI assistant behavior
- Full interaction logging for benchmarking analysis

Approach 2: External Prompt-Based Testing (Backup Method)

Advantages:

- ✅ **Real-world validity**: Authentic interaction with external platforms
- ✅ **Scale flexibility**: Can accommodate more participants
- ✅ **Cost efficiency**: No API costs for research team
- ✅ **Platform diversity**: Can test different AI models easily

Challenges:

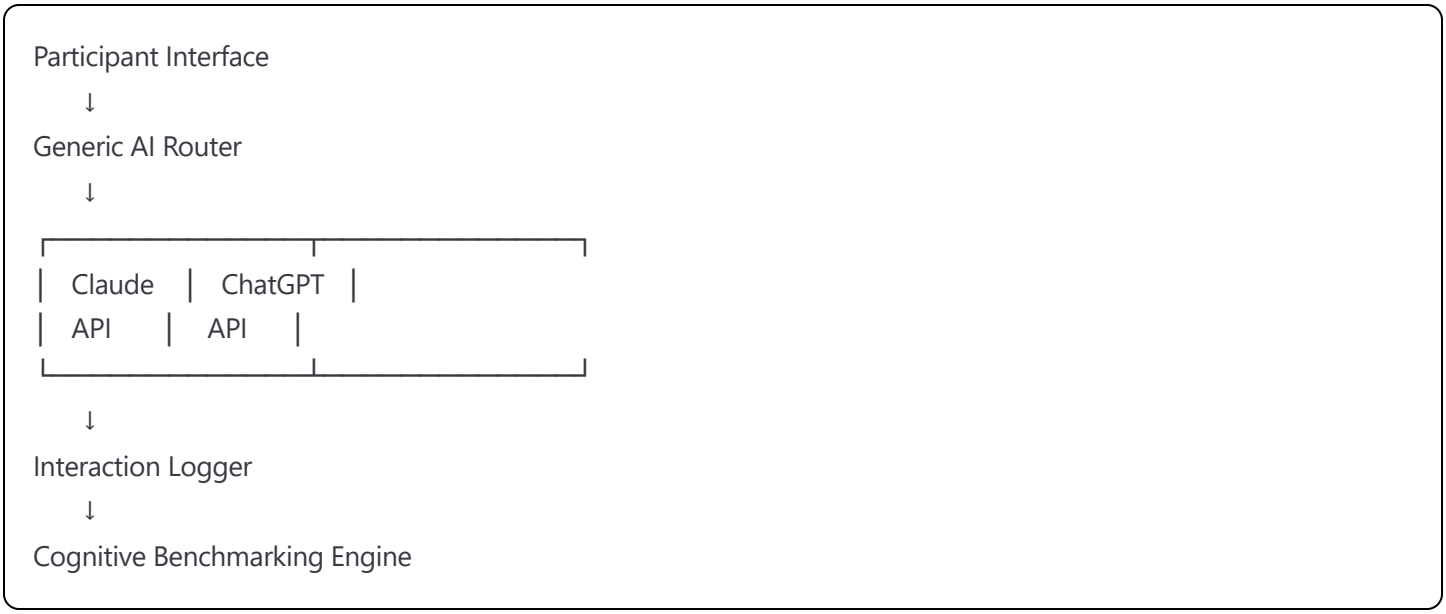
- ⚠️ **Data collection dependency**: Relies on participant compliance

- ⚠️ **Environment variation**: Different interfaces may affect behavior
- ⚠️ **Limited real-time feedback**: No immediate cognitive metrics

---

Internal API Integration Implementation

System Architecture



Core Components

1. Generic AI Router

File: `src/ai_integrations/generic_ai_router.py`

python

```
"""
Generic AI Router - Handles calls to external AI services
Designed to mimic standard AI assistant behavior without cognitive scaffolding
"""

import asyncio
import logging
from typing import Dict, List, Any, Optional
from datetime import datetime
import json
import aiohttp
from anthropic import AsyncAnthropic
import openai

logger = logging.getLogger(__name__)

class GenericAIRouter:
    """Routes interactions to generic AI services"""

    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.anthropic_client = AsyncAnthropic(api_key=config['anthropic']['api_key'])
        self.openai_client = openai.AsyncOpenAI(api_key=config['openai']['api_key'])
        self.current_model = config.get('default_model', 'claude')

        # Generic assistant prompts (non-scaffolding)
        self.system_prompts = {
            'claude': self._get_claude_system_prompt(),
            'chatgpt': self._get_chatgpt_system_prompt()
        }

    async def process_interaction(self, user_input: str, session_context: Dict[str, Any],
                                  phase: str) -> Dict[str, Any]:
        """Process user interaction through generic AI"""

        start_time = datetime.now()

        try:
            # Route to appropriate AI service
            if self.current_model == 'claude':
                response = await self._call_claude(user_input, session_context, phase)
            elif self.current_model == 'chatgpt':
                response = await self._call_chatgpt(user_input, session_context, phase)
```

```
            else:
                raise ValueError(f"Unsupported model: {self.current_model}")

            processing_time = (datetime.now() - start_time).total_seconds()

            return {
                'ai_response': response,
                'model_used': self.current_model,
                'processing_time': processing_time,
                'timestamp': datetime.now().isoformat(),
                'metadata': {
                    'phase': phase,
                    'approach': 'direct_assistance',
                    'scaffolding_level': 'none'
                }
            }

        except Exception as e:
            logger.error(f"Generic AI processing error: {e}")
            return {
                'ai_response': "I apologize, but I'm having difficulty processing your request right now. Please try rephrasing y
                'model_used': self.current_model,
                'processing_time': (datetime.now() - start_time).total_seconds(),
                'error': str(e)
            }

    async def _call_claude(self, user_input: str, session_context: Dict[str, Any],
                           phase: str) -> str:
        """Call Claude API with generic assistant behavior"""

        # Build conversation history
        messages = self._build_conversation_history(session_context)
        messages.append({
            "role": "user",
            "content": user_input
        })

        response = await self.anthropic_client.messages.create(
            model="claude-3-sonnet-20240229",
            max_tokens=1000,
            system=self.system_prompts['claude'],
            messages=messages
        )
```

```
    return response.content[0].text

async def _call_chatgpt(self, user_input: str, session_context: Dict[str, Any],
                        phase: str) -> str:
    """Call ChatGPT API with generic assistant behavior"""

    # Build conversation history
    messages = [{"role": "system", "content": self.system_prompts['chatgpt']}]
    messages.extend(self._build_conversation_history(session_context))
    messages.append({
        "role": "user",
        "content": user_input
    })

    response = await self.openai_client.chat.completions.create(
        model="gpt-4-turbo-preview",
        max_tokens=1000,
        messages=messages
    )

    return response.choices[0].message.content

def _get_claude_system_prompt(self) -> str:
    """Get system prompt for Claude to behave as generic assistant"""
    return """
    You are a helpful AI assistant designed to provide direct, informative responses to questions about architectural de

    Your behavior should be:
    - Provide clear, comprehensive answers to questions
    - Offer specific suggestions and solutions when asked
    - Share relevant examples and case studies
    - Give detailed explanations of architectural concepts
    - Provide step-by-step guidance when requested
    - Be supportive and encouraging
    - Focus on being helpful and informative

    You should NOT:
    - Ask excessive follow-up questions instead of answering
    - Withhold information to make the user think more
    - Use Socratic questioning techniques
    - Force the user to discover answers on their own
    - Provide minimal guidance that requires extensive user reflection

    The user is working on architectural design projects and you should provide the direct assistance they're seeking.
```

```
        """

    def _get_chatgpt_system_prompt(self) -> str:
        """Get system prompt for ChatGPT to behave as generic assistant"""
        return """
        You are ChatGPT, a helpful AI assistant created by OpenAI. You provide direct, informative responses to questions .

        Your approach:
        - Give comprehensive, detailed answers to user questions
        - Provide specific suggestions and actionable advice
        - Share relevant examples, precedents, and case studies
        - Explain architectural concepts clearly and thoroughly
        - Offer step-by-step guidance and practical solutions
        - Be encouraging and supportive of the user's design process

        You are designed to be maximally helpful by providing the information and guidance the user is seeking directly, r

        The user is working on architectural design projects and you should assist them efficiently and comprehensively.
        """

    def _build_conversation_history(self, session_context: Dict[str, Any]) -> List[Dict[str, str]]:
        """Build conversation history from session context"""

        history = []
        interactions = session_context.get('interactions', [])

        # Include last 10 interactions for context
        for interaction in interactions[-10:]:
            if interaction.get('user_input'):
                history.append({
                    "role": "user",
                    "content": interaction['user_input']
                })
            if interaction.get('ai_response'):
                history.append({
                    "role": "assistant",
                    "content": interaction['ai_response']
                })

        return history

class ModelRotator:
    """Handles rotation between different AI models for comparative testing"""
```

```python
    def __init__(self, models: List[str]):
        self.models = models
        self.current_index = 0
        self.assignments = {}  # participant_id -> model

    def assign_model(self, participant_id: str) -> str:
        """Assign a model to a participant"""
        if participant_id not in self.assignments:
            self.assignments[participant_id] = self.models[self.current_index % len(self.models)]
            self.current_index += 1

        return self.assignments[participant_id]

    def get_assignment(self, participant_id: str) -> Optional[str]:
        """Get current model assignment for participant"""
        return self.assignments.get(participant_id)
```

2. Generic AI Session Manager

File: `src/data_processing/generic_ai_session_manager.py`

```python
```

```
"""
Session Manager for Generic AI Control Group
Handles logging and benchmarking for non-MENTOR interactions
"""

import asyncio
import uuid
from datetime import datetime
from typing import Dict, List, Any
import logging
from dataclasses import dataclass

from .session_manager import SessionManager
from ..benchmarking.cognitive_metrics import CognitiveMetricsEngine

logger = logging.getLogger(__name__)

@dataclass
class GenericAIInteraction:
    session_id: str
    timestamp: datetime
    user_input: str
    ai_response: str
    model_used: str
    processing_time: float
    phase: str
    interaction_type: str

class GenericAISessionManager(SessionManager):
    """Extended session manager for generic AI control group"""

    def __init__(self):
        super().__init__()
        self.cognitive_engine = CognitiveMetricsEngine()

    async def log_generic_ai_interaction(self, interaction: GenericAIInteraction) -> str:
        """Log interaction from generic AI and calculate cognitive metrics"""

        # Convert to standard interaction format for cognitive analysis
        interaction_data = {
            'session_id': interaction.session_id,
            'user_input': interaction.user_input,
            'ai_response': interaction.ai_response,
```

```
        'type': interaction.interaction_type,
        'phase': interaction.phase,
        'metadata': {
            'model_used': interaction.model_used,
            'processing_time': interaction.processing_time,
            'approach_type': 'direct_assistance',
            'scaffolding_present': False
        }
    }

    # Calculate cognitive metrics (adapted for non-scaffolded responses)
    metrics_result = await self.cognitive_engine.process_interaction(
        session_id=interaction.session_id,
        interaction_data=interaction_data
    )

    # Enhanced logging for comparative analysis
    enhanced_interaction_data = {
        **interaction_data,
        'cognitive_metrics': metrics_result['metrics'].__dict__,
        'generic_ai_specific': {
            'direct_answer_provided': await self._analyze_directness(interaction.ai_response),
            'information_density': await self._calculate_information_density(interaction.ai_response),
            'suggestion_count': await self._count_suggestions(interaction.ai_response),
            'question_ratio': await self._calculate_question_ratio(interaction.ai_response)
        }
    }

    # Save to database with additional metadata
    interaction_id = await self.save_interaction(interaction.session_id, enhanced_interaction_data)

    return interaction_id

async def _analyze_directness(self, ai_response: str) -> float:
    """Analyze how directly the AI answered the question"""

    # Look for direct answer patterns
    direct_indicators = [
        'you should', 'i recommend', 'the best approach is',
        'here\'s how', 'the solution is', 'you can',
        'i suggest', 'the answer is', 'use this'
    ]

    response_lower = ai_response.lower()
```

```
    direct_count = sum(1 for indicator in direct_indicators if indicator in response_lower)

    # Normalize by response length
    directness_score = min(1.0, direct_count / max(1, len(ai_response.split()) / 50))

    return directness_score

async def _calculate_information_density(self, ai_response: str) -> float:
    """Calculate information density of the response"""

    # Count specific information elements
    info_indicators = [
        'example', 'case study', 'precedent', 'research shows',
        'studies indicate', 'according to', 'data suggests',
        'guidelines', 'standards', 'regulations', 'best practices'
    ]

    response_lower = ai_response.lower()
    info_count = sum(1 for indicator in info_indicators if indicator in response_lower)

    # Factor in numbers, measurements, specific references
    numeric_refs = len([word for word in ai_response.split() if any(char.isdigit() for char in word)])

    total_words = len(ai_response.split())
    density = (info_count + numeric_refs * 0.5) / max(total_words / 100, 1)

    return min(1.0, density)

async def _count_suggestions(self, ai_response: str) -> int:
    """Count number of discrete suggestions in response"""

    # Count numbered lists, bullet points, and suggestion phrases
    suggestion_patterns = [
        r'\d+\.', r'•', r'-\s+', r'first', r'second', r'third',
        r'another option', r'alternatively', r'you could also'
    ]

    import re
    suggestion_count = 0

    for pattern in suggestion_patterns:
        matches = re.findall(pattern, ai_response, re.IGNORECASE)
        suggestion_count += len(matches)
```

```
        return suggestion_count

    async def _calculate_question_ratio(self, ai_response: str) -> float:
        """Calculate ratio of questions to statements in response"""

        sentences = [s.strip() for s in ai_response.split('.') if s.strip()]
        total_sentences = len(sentences)

        if total_sentences == 0:
            return 0.0

        question_count = sum(1 for sentence in sentences if '?' in sentence)

        return question_count / total_sentences

    async def generate_comparative_report(self, session_id: str) -> Dict[str, Any]:
        """Generate report comparing generic AI interaction patterns to MENTOR patterns"""

        session_data = await self.get_session_data(session_id)
        interactions = session_data['interactions']

        if not interactions:
            return {"error": "No interactions found"}

        # Analyze interaction patterns
        total_interactions = len(interactions)
        total_directness = sum(i.get('generic_ai_specific', {}).get('direct_answer_provided', 0)
                               for i in interactions)
        avg_directness = total_directness / total_interactions

        total_info_density = sum(i.get('generic_ai_specific', {}).get('information_density', 0)
                                 for i in interactions)
        avg_info_density = total_info_density / total_interactions

        total_suggestions = sum(i.get('generic_ai_specific', {}).get('suggestion_count', 0)
                                for i in interactions)

        avg_question_ratio = sum(i.get('generic_ai_specific', {}).get('question_ratio', 0)
                                 for i in interactions) / total_interactions

        # Extract cognitive metrics progression
        cognitive_progression = []
        for interaction in interactions:
            if 'cognitive_metrics' in interaction:
```

```
            cognitive_progression.append(interaction['cognitive_metrics'])

    return {
        'session_id': session_id,
        'group_type': 'generic_ai',
        'interaction_analysis': {
            'total_interactions': total_interactions,
            'average_directness': avg_directness,
            'average_information_density': avg_info_density,
            'total_suggestions_provided': total_suggestions,
            'average_question_ratio': avg_question_ratio
        },
        'cognitive_metrics_summary': {
            'final_metrics': cognitive_progression[-1] if cognitive_progression else {},
            'progression': cognitive_progression
        },
        'comparative_indicators': {
            'information_provision_vs_questioning': avg_directness / max(avg_question_ratio, 0.1),
            'assistance_density': avg_info_density * total_suggestions / total_interactions,
            'scaffolding_level': 'minimal'  # By design for generic AI
        }
    }
```

3. Modified API Endpoints

File: `src/api/generic_ai_routes.py`

```
python
```

```
"""
API Routes specifically for Generic AI Control Group
"""

from fastapi import APIRouter, HTTPException
from typing import Dict, Any
import logging

from ..ai_integrations.generic_ai_router import GenericAIRouter, ModelRotator
from ..data_processing.generic_ai_session_manager import GenericAISessionManager, GenericAIInteraction
from ..utils.config import settings

logger = logging.getLogger(__name__)
router = APIRouter(prefix="/generic-ai", tags=["generic-ai"])

# Initialize components
ai_router = GenericAIRouter(settings.ai_models)
model_rotator = ModelRotator(['claude', 'chatgpt'])
session_manager = GenericAISessionManager()

@router.post("/sessions/{session_id}/interact")
async def process_generic_ai_interaction(session_id: str, interaction_data: Dict[str, Any]):
    """Process interaction through generic AI services"""

    try:
        # Get participant's assigned model
        participant_id = interaction_data.get('participant_id')
        if not participant_id:
            raise ValueError("participant_id required")

        assigned_model = model_rotator.assign_model(participant_id)
        ai_router.current_model = assigned_model

        # Get session context
        session_context = await session_manager.get_session_data(session_id)

        # Process through generic AI
        ai_result = await ai_router.process_interaction(
            user_input=interaction_data['user_input'],
            session_context=session_context,
            phase=interaction_data.get('phase', 'ideation')
        )
```

```
        # Create interaction record
        interaction = GenericAIInteraction(
            session_id=session_id,
            timestamp=datetime.now(),
            user_input=interaction_data['user_input'],
            ai_response=ai_result['ai_response'],
            model_used=ai_result['model_used'],
            processing_time=ai_result['processing_time'],
            phase=interaction_data.get('phase', 'ideation'),
            interaction_type=interaction_data.get('type', 'question')
        )

        # Log and analyze
        interaction_id = await session_manager.log_generic_ai_interaction(interaction)

        return {
            'interaction_id': interaction_id,
            'ai_response': ai_result['ai_response'],
            'model_used': ai_result['model_used'],
            'processing_time': ai_result['processing_time'],
            'metadata': ai_result.get('metadata', {})
        }

    except Exception as e:
        logger.error(f"Generic AI interaction error: {e}")
        raise HTTPException(status_code=500, detail=str(e))

@router.get("/sessions/{session_id}/comparative-analysis")
async def get_comparative_analysis(session_id: str):
    """Get comparative analysis for generic AI session"""

    try:
        report = await session_manager.generate_comparative_report(session_id)
        return report
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@router.post("/model-assignment")
async def assign_model_to_participant(assignment_data: Dict[str, str]):
    """Manually assign specific model to participant"""

    participant_id = assignment_data['participant_id']
    model = assignment_data['model']
```

```
    if model not in ['claude', 'chatgpt']:
        raise HTTPException(status_code=400, detail="Invalid model")

    model_rotator.assignments[participant_id] = model

    return {"participant_id": participant_id, "assigned_model": model}
```

External Prompt-Based Testing Implementation

Comprehensive Testing Protocol

1. Master Prompt Template

File: `external_testing/master_prompt_template.md`

```
markdown

# ARCHITECTURAL DESIGN RESEARCH STUDY - PARTICIPANT INSTRUCTIONS

## Study Information
- **Study ID**: MENTOR-BTEST-2024
- **Participant ID**: [TO BE PROVIDED]
- **Group Assignment**: Generic AI Control Group
- **Session Duration**: 3 hours (across multiple sessions)

## Overview
You are participating in a research study on AI-assisted architectural design. You will work through a series of design ch

## CRITICAL: Data Collection Requirements
**You MUST copy and save ALL interactions with the AI for research analysis.**

### Required Data Collection Format:
For EVERY interaction, copy this template into a document and fill it out:
```

INTERACTION LOG #[NUMBER] Timestamp: [DATE/TIME] Phase: [Ideation/Visualization/Materialization] User Input: "[YOUR EXACT QUESTION/INPUT]" AI Response: "[AI'S COMPLETE RESPONSE]" Response Time: [APPROXIMATE SECONDS] Notes: [YOUR OBSERVATIONS]

Phase 1: Ideation Phase (20 minutes)

Pre-Phase Setup Prompt for AI:
Copy and paste this EXACT prompt to your AI assistant before beginning:

"""
You are assisting an architecture student with a community center design project. The project involves converting an abandoned warehouse (150m x 80m x 12m height) into a community center for a diverse urban neighborhood of 15,000 residents.

Please provide helpful, comprehensive responses to questions about:
- Architectural design principles
- Community center programming
- Adaptive reuse strategies
- Cultural sensitivity in design
- Sustainability considerations
- Spatial planning and organization

Be thorough in your responses, provide specific examples when possible, and offer practical design guidance. The student is working through their design process systematically.
"""

Task 1: Architectural Concept Development (15 minutes)

Your Mission:
Develop a comprehensive concept for transforming the warehouse into a community center.

Required Interactions (Ask these questions to the AI):

1. **Community Analysis Question:**
   "I need to design a community center for a diverse urban neighborhood of 15,000 residents in a converted warehouse. What are the key community needs I should consider?"

2. **Adaptive Reuse Question:**
   "What are the main opportunities and challenges when converting an industrial warehouse (150m x 80m x 12m height) into a community center?"

3. **Programming Question:**
   "What spaces and functions should I include in this community center? Please suggest a preliminary space program."

4. **Cultural Sensitivity Question:**

"How can I ensure my community center design is culturally sensitive and inclusive for a diverse neighborhood?"

5. **Design Concept Synthesis:**
"Based on our discussion, help me synthesize these considerations into a coherent design concept. What should be my main design principles?"

Task 2: Spatial Program Development (5 minutes)

Follow-up Questions:

6. **Spatial Relationships:**
"How should I organize the spatial relationships between different functions in the community center?"

7. **Circulation Planning:**
"What circulation patterns would work best for this community center?"

Phase 2: Visualization Phase (15 minutes)

AI Setup Prompt for Visualization Phase:

You are now helping the student develop their community center concept visually. Focus on:

- 2D planning and layout strategies
- Spatial proportions and relationships
- Environmental design (lighting, ventilation)
- Site integration and context

Provide specific guidance on spatial design and proportions.

Task 3: 2D Design Development (10 minutes)

Required Interactions:

8. **Layout Planning:**
   "I'm developing the floor plan for my community center. The warehouse is 150m x 80m. How should I approach the layout to accommodate [list your main program elements from Phase 1]?"

9. **Spatial Proportions:**
   "What are appropriate spatial proportions for the main gathering space, library area, and multi-purpose rooms in a community center?"

10. **Natural Lighting Strategy:**
   "The existing warehouse has industrial windows along the north and south walls. How can I optimize natural lighting throughout the community center?"

11. **Circulation Design:**
   "How should I design the main circulation spine to connect different areas efficiently while creating opportunities for social interaction?"

Task 4: Environmental Integration (5 minutes)

12. **Ventilation Strategy:**
   "What ventilation strategies work best for large community spaces in converted warehouses?"

13. **Acoustic Considerations:**
   "How do I manage acoustics in a large, open warehouse conversion with diverse program uses?"

Phase 3: Materialization Phase (20 minutes)

AI Setup Prompt for Materialization:

You are now helping the student develop technical and material aspects of their community center design. Focus on:

- Material selection for adaptive reuse
- Structural considerations
- Building systems integration
- Implementation strategies

Provide practical, technical guidance for realizing the design.

Task 5: Material and Structural Systems (15 minutes)

14. **Material Selection:**
    "What materials are most appropriate for the interior build-out of this warehouse conversion? How can I balance durability, maintenance, and budget?"

15. **Structural Integration:**
    "The existing warehouse has a steel frame structure. How can I work with this system to create my spaces?"

16. **Building Systems:**
    "How should I integrate HVAC, electrical, and plumbing systems into the existing warehouse structure?"

17. **Accessibility Compliance:**
    "What accessibility requirements must I meet for a community center, and how do I integrate these into the warehouse conversion design?"

Task 6: Implementation Strategy (5 minutes)

18. **Construction Phasing:**
    "How should I phase the construction to allow parts of the community center to open while other areas are still under construction?"

19. **Community Engagement:**
    "How can the community be involved in the construction and finishing process?"

Post-Testing Requirements

Final Documentation Tasks:

1. **Complete Interaction Log**: Ensure all 24+ interactions are properly logged with timestamps

2. **Reflection Journal**: Write a 500-word reflection on:
   - How the AI influenced your design process
   - What types of responses were most/least helpful
   - How your thinking evolved through the phases
   - Any limitations you noticed in the AI's guidance

3. **Design Summary**: Create a 1-page summary of your final design concept including:
   - Main design principles
   - Key spatial relationships
   - Material and technical strategies
   - Community engagement approach

4. **AI Interaction Analysis**: Complete this self-assessment:
   - Did you rely more on AI suggestions or your own creative thinking?
   - How often did you ask follow-up questions vs. accept first responses?
   - What patterns did you notice in how you used the AI?

File Submission Requirements:
Submit these files to [RESEARCH EMAIL]:
- `interactions_log_[PARTICIPANT_ID].txt` - Complete interaction log
- `reflection_[PARTICIPANT_ID].txt` - Reflection journal
- `design_summary_[PARTICIPANT_ID].pdf` - Design summary
- `self_assessment_[PARTICIPANT_ID].txt` - AI interaction analysis

Technical Requirements

Recommended AI Platforms:
- **Primary**: Claude (claude.ai)
- **Secondary**: ChatGPT (chat.openai.com)
- Use the SAME platform for all phases

Browser/Setup Requirements:
- Use consistent browser and settings
- Clear AI conversation history before starting (fresh session)
- Ensure stable internet connection
- Have document ready for copying interactions

Time Management:
- Phase 1: 60 minutes maximum
- Break: 15-30 minutes recommended
- Phase 2: 60 minutes maximum
- Break: 15-30 minutes recommended
- Phase 3: 60 minutes maximum
- Documentation: 30 minutes

Research Ethics and Consent

By participating, you confirm:
- You understand this is research comparing AI interaction approaches
- You consent to analysis of your interaction patterns
- You will provide honest responses and complete documentation
- You understand your data will be anonymized for research publication

Support Contact
Research Team: [EMAIL]

Technical Issues: [EMAIL]
Questions: [EMAIL]

Study ID: MENTOR-BTEST-2024 | Version 1.0 | Date: [DATE]

2. Data Collection Automation Script

File: **external_testing/data_collection_parser.py**

python

```
"""
Parser for external AI interaction logs
Converts participant submissions into analyzable format
"""

import re
import json
import pandas as pd
from datetime import datetime
from typing import Dict, List, Any
import logging

logger = logging.getLogger(__name__)

class ExternalLogParser:
    """Parses and analyzes external AI interaction logs"""

    def __init__(self):
        self.interaction_pattern = re.compile(
            r'---\s*INTERACTION LOG #(\d+)\s*\n'
            r'Timestamp: (.*?)\n'
            r'Phase: (.*?)\n'
            r'User Input: "(.*?)"\n'
            r'AI Response: "(.*?)"\n'
            r'Response Time: (.*?)\n'
            r'Notes: (.*?)\n'
            r'---',
            re.DOTALL
        )

    def parse_interaction_log(self, log_content: str, participant_id: str) -> Dict[str, Any]:
        """Parse complete interaction log from participant"""

        interactions = []
        matches = self.interaction_pattern.findall(log_content)

        for match in matches:
            interaction_num, timestamp, phase, user_input, ai_response, response_time, notes = match

            interaction = {
                'interaction_number': int(interaction_num),
                'timestamp': self._parse_timestamp(timestamp),
                'phase': phase.strip(),
```

```
            'user_input': user_input.strip(),
            'ai_response': ai_response.strip(),
            'response_time': self._parse_response_time(response_time),
            'notes': notes.strip(),
            'participant_id': participant_id,
            'word_count_input': len(user_input.split()),
            'word_count_response': len(ai_response.split()),
            'character_count_response': len(ai_response)
        }

        # Add analysis metrics
        interaction.update(self._analyze_interaction(interaction))

        interactions.append(interaction)

    return {
        'participant_id': participant_id,
        'total_interactions': len(interactions),
        'interactions': interactions,
        'phase_distribution': self._calculate_phase_distribution(interactions),
        'summary_metrics': self._calculate_summary_metrics(interactions)
    }

def _parse_timestamp(self, timestamp_str: str) -> str:
    """Parse and standardize timestamp"""
    # Handle various timestamp formats
    timestamp_str = timestamp_str.strip()

    try:
        # Try to parse and reformat
        if '/' in timestamp_str:
            dt = datetime.strptime(timestamp_str, '%m/%d/%Y %H:%M:%S')
        elif '-' in timestamp_str:
            dt = datetime.strptime(timestamp_str, '%Y-%m-%d %H:%M:%S')
        else:
            return timestamp_str  # Return as-is if can't parse

        return dt.isoformat()
    except:
        return timestamp_str

def _parse_response_time(self, response_time_str: str) -> float:
    """Parse response time to seconds"""
    response_time_str = response_time_str.strip().lower()
```

```
        try:
            # Extract number from string
            numbers = re.findall(r'\d+', response_time_str)
            if numbers:
                time_value = float(numbers[0])

                if 'min' in response_time_str:
                    return time_value * 60
                else:
                    return time_value
        except:
            pass

        return 0.0

    def _analyze_interaction(self, interaction: Dict[str, Any]) -> Dict[str, Any]:
        """Analyze individual interaction for research metrics"""

        user_input = interaction['user_input'].lower()
        ai_response = interaction['ai_response'].lower()

        analysis = {
            # Question type analysis
            'question_type': self._classify_question_type(user_input),
            'question_complexity': self._assess_question_complexity(user_input),

            # Response analysis
            'response_directness': self._assess_response_directness(ai_response),
            'suggestions_provided': self._count_suggestions(ai_response),
            'examples_provided': self._count_examples(ai_response),
            'follow_up_questions': self._count_follow_up_questions(ai_response),

            # Information density
            'technical_terms_count': self._count_technical_terms(ai_response),
            'specific_numbers': self._count_specific_numbers(ai_response),
            'references_provided': self._count_references(ai_response),

            # Cognitive indicators
            'reasoning_language': self._detect_reasoning_language(user_input),
            'metacognitive_language': self._detect_metacognitive_language(user_input)
        }

        return analysis
```

```
def _classify_question_type(self, user_input: str) -> str:
    """Classify the type of question asked"""

    if any(phrase in user_input for phrase in ['what is', 'define', 'explain']):
        return 'definitional'
    elif any(phrase in user_input for phrase in ['how should', 'how can', 'how do']):
        return 'procedural'
    elif any(phrase in user_input for phrase in ['why', 'what makes', 'what causes']):
        return 'causal'
    elif any(phrase in user_input for phrase in ['compare', 'difference', 'better']):
        return 'comparative'
    elif any(phrase in user_input for phrase in ['what if', 'suppose', 'consider']):
        return 'hypothetical'
    else:
        return 'other'

def _assess_question_complexity(self, user_input: str) -> str:
    """Assess complexity level of question"""

    complexity_indicators = {
        'high': ['integrate', 'synthesize', 'evaluate', 'analyze', 'optimize', 'balance'],
        'medium': ['consider', 'address', 'ensure', 'manage', 'coordinate'],
        'low': ['what', 'how', 'where', 'when', 'list', 'describe']
    }

    for level, indicators in complexity_indicators.items():
        if any(indicator in user_input for indicator in indicators):
            return level

    return 'low'

def _assess_response_directness(self, ai_response: str) -> float:
    """Assess how directly the AI answered"""

    direct_indicators = [
        'you should', 'i recommend', 'the best approach',
        'use this', 'do this', 'here\'s how', 'follow these steps'
    ]

    indirect_indicators = [
        'consider', 'you might', 'it depends', 'think about',
        'what do you think', 'have you considered'
    ]
```

```
    direct_count = sum(1 for indicator in direct_indicators if indicator in ai_response)
    indirect_count = sum(1 for indicator in indirect_indicators if indicator in ai_response)

    total_indicators = direct_count + indirect_count
    if total_indicators == 0:
        return 0.5

    return direct_count / total_indicators

def _count_suggestions(self, ai_response: str) -> int:
    """Count discrete suggestions in AI response"""

    suggestion_patterns = [
        r'\d+\.', r'•', r'-\s+[A-Z]', r'first,?', r'second,?', r'third,?',
        r'another', r'also consider', r'alternatively'
    ]

    count = 0
    for pattern in suggestion_patterns:
        matches = re.findall(pattern, ai_response, re.IGNORECASE)
        count += len(matches)

    return count

def _count_examples(self, ai_response: str) -> int:
    """Count examples provided in response"""

    example_indicators = [
        'for example', 'such as', 'like', 'including',
        'case study', 'precedent', 'consider the'
    ]

    return sum(1 for indicator in example_indicators if indicator in ai_response)

def _count_follow_up_questions(self, ai_response: str) -> int:
    """Count follow-up questions AI asked"""

    sentences = ai_response.split('.')
    return sum(1 for sentence in sentences if '?' in sentence)

def _count_technical_terms(self, ai_response: str) -> int:
    """Count architectural/technical terms used"""
```

```
        technical_terms = [
            'circulation', 'adjacency', 'proportion', 'scale', 'hvac',
            'structural', 'spatial', 'program', 'zoning', 'accessibility',
            'sustainability', 'thermal', 'acoustic', 'daylighting'
        ]

        return sum(1 for term in technical_terms if term in ai_response)

    def _count_specific_numbers(self, ai_response: str) -> int:
        """Count specific measurements/numbers provided"""

        number_patterns = [
            r'\d+\s*(?:m|ft|feet|meters|mm|cm|inches)',  # measurements
            r'\d+\s*(?:sf|square feet|m²|square meters)',  # area
            r'\d+\s*(?:%|percent)',  # percentages
            r'\$\d+',  # costs
            r'\d+\s*(?:people|persons|occupants)'  # capacity
        ]

        count = 0
        for pattern in number_patterns:
            matches = re.findall(pattern, ai_response, re.IGNORECASE)
            count += len(matches)

        return count

    def _count_references(self, ai_response: str) -> int:
        """Count references to standards, codes, precedents"""

        reference_indicators = [
            'code', 'standard', 'regulation', 'guideline',
            'ada', 'ibc', 'leed', 'research shows', 'studies'
        ]

        return sum(1 for indicator in reference_indicators if indicator in ai_response)

    def _detect_reasoning_language(self, user_input: str) -> bool:
        """Detect reasoning/analytical language in user input"""

        reasoning_indicators = [
            'because', 'therefore', 'since', 'given that',
            'considering', 'due to', 'as a result'
        ]
```

```
        return any(indicator in user_input for indicator in reasoning_indicators)

    def _detect_metacognitive_language(self, user_input: str) -> bool:
        """Detect metacognitive awareness in user input"""

        metacognitive_indicators = [
            'i think', 'i believe', 'i\'m considering',
            'my approach', 'i realize', 'i understand'
        ]

        return any(indicator in user_input for indicator in metacognitive_indicators)

    def _calculate_phase_distribution(self, interactions: List[Dict]) -> Dict[str, int]:
        """Calculate distribution of interactions across phases"""

        phase_counts = {}
        for interaction in interactions:
            phase = interaction['phase']
            phase_counts[phase] = phase_counts.get(phase, 0) + 1

        return phase_counts

    def _calculate_summary_metrics(self, interactions: List[Dict]) -> Dict[str, Any]:
        """Calculate summary metrics for the session"""

        if not interactions:
            return {}

        total_interactions = len(interactions)

        return {
            'avg_user_input_length': sum(i['word_count_input'] for i in interactions) / total_interactions,
            'avg_ai_response_length': sum(i['word_count_response'] for i in interactions) / total_interactions,
            'avg_response_directness': sum(i['response_directness'] for i in interactions) / total_interactions,
            'total_suggestions_received': sum(i['suggestions_provided'] for i in interactions),
            'total_examples_received': sum(i['examples_provided'] for i in interactions),
            'question_complexity_distribution': self._get_complexity_distribution(interactions),
            'reasoning_language_usage': sum(1 for i in interactions if i['reasoning_language']) / total_interactions,
            'metacognitive_language_usage': sum(1 for i in interactions if i['metacognitive_language']) / total_interactions
        }

    def _get_complexity_distribution(self, interactions: List[Dict]) -> Dict[str, int]:
        """Get distribution of question complexity levels"""
```

```
        complexity_counts = {}
        for interaction in interactions:
            complexity = interaction['question_complexity']
            complexity_counts[complexity] = complexity_counts.get(complexity, 0) + 1

        return complexity_counts

    def export_for_analysis(self, parsed_data: Dict[str, Any], output_format: str = 'json') -> str:
        """Export parsed data for statistical analysis"""

        if output_format == 'json':
            return json.dumps(parsed_data, indent=2)

        elif output_format == 'csv':
            # Convert to flat CSV format
            interactions_df = pd.DataFrame(parsed_data['interactions'])
            return interactions_df.to_csv(index=False)

        elif output_format == 'analysis_ready':
            # Format for direct input to cognitive benchmarking system
            analysis_format = {
                'session_id': f"external_{parsed_data['participant_id']}",
                'group_assignment': 'generic_ai_external',
                'interactions': []
            }

            for interaction in parsed_data['interactions']:
                formatted_interaction = {
                    'timestamp': interaction['timestamp'],
                    'user_input': interaction['user_input'],
                    'ai_response': interaction['ai_response'],
                    'phase': interaction['phase'],
                    'metadata': {
                        'external_source': True,
                        'response_time': interaction['response_time'],
                        'question_type': interaction['question_type'],
                        'response_directness': interaction['response_directness'],
                        'suggestions_count': interaction['suggestions_provided']
                    }
                }
                analysis_format['interactions'].append(formatted_interaction)

            return json.dumps(analysis_format, indent=2)
```

```
        else:
            raise ValueError(f"Unsupported output format: {output_format}")

# Usage example
if __name__ == "__main__":
    parser = ExternalLogParser()

    # Example usage
    with open('participant_001_log.txt', 'r') as f:
        log_content = f.read()

    parsed_data = parser.parse_interaction_log(log_content, 'PART_001')

    # Export for analysis
    json_output = parser.export_for_analysis(parsed_data, 'json')
    csv_output = parser.export_for_analysis(parsed_data, 'csv')
    analysis_ready = parser.export_for_analysis(parsed_data, 'analysis_ready')
```

Comparative Analysis Framework

Metrics Comparison Between Internal and External Groups

Metric Category	Internal API Group	External Prompt Group
Data Quality	Perfect capture	Dependent on participant compliance
Timing Accuracy	Millisecond precision	Participant-reported estimates
Context Preservation	Full session context	Limited to logged interactions
Standardization	Identical conditions	Variable external factors
Real-time Analysis	Live cognitive metrics	Post-hoc analysis only
Cost	API usage costs	Participant time costs
Scalability	Limited by API costs	Limited by participant compliance
Ecological Validity	Controlled environment	Real-world usage patterns

Recommended Implementation Strategy

1. **Primary Method**: Internal API integration for core research participants (n=20-30)
2. **Secondary Method**: External prompt-based testing for additional validation (n=10-15)
3. **Cross-validation**: Compare subset of participants who complete both methods
4. **Quality control**: Implement automated parsing and validation for external submissions

This hybrid approach maximizes both research validity and practical scalability while ensuring robust data collection for meaningful cognitive benchmarking comparisons.