Tutorial Script 4: Using Claude to Plan and Implement a New Feature

Duration: 12-15 minutes (Split into Part A and Part B) **Objective:** Demonstrate end-to-end feature implementation using Claude Code - from learning about requirements to planning, implementation, and testing

Feature Example: Adding Multiple Choice Questions to the Flashcard App

---

PART A: Learning and Planning (7 minutes)

INTRODUCTION (1 minute)

[SCREEN: Claude Code interface with minimal-flashcards open]

Professor:

"Welcome back! In our previous tutorials, we learned how to use Claude Code and how to understand a new codebase. Now we're going to put it all together and implement a real feature.

Today's task: Adding multiple-choice question support to our flashcard application.

Currently, our app only supports basic question-and-answer cards. We want to add the ability to create cards with four options where the user selects the correct answer.

This is a realistic feature request you'd encounter in a real development job. We'll use Claude to:

1. Understand what needs to change
2. Plan the implementation
3. Write the code
4. Test it

Let's get started!"

PART 1: Learn About the Feature Requirements (2 minutes)

[SCREEN: Claude Code interface]

Professor:

"Step one: Before we write ANY code, we need to understand what we're building. Let me ask Claude to help us think through the requirements.

Question 1: Understanding the current state"

[TYPE in Claude Code]:

```
I want to add multiple-choice questions to this flashcard app. Currently,
we only support basic Q&A cards.

First, help me understand what currently exists:
1. Show me the Card model in backend/app/models/card.py
2. What card types are currently defined?
3. How are cards currently displayed in the study session?
```

[SCREEN: Shows Claude reading files and explaining]

Professor:

"Perfect! Claude shows us:

1. **Card model** has fields for `type`, `prompt`, `answer`, `options`, and `cloze_data`
2. **CardType enum** currently only has `BASIC = 'basic'` (in models/enums.py)
3. **Study session** displays cards with a flip animation showing the answer

Wait - Claude mentioned that the Card model ALREADY has an `options` field and `cloze_data`. Let me ask about that:"

[TYPE in Claude Code]:

```
I see the Card model already has an 'options' field. Is multiple choice
already implemented? Check the CardType enum in
backend/app/models/enums.py
```

[SCREEN: Shows Claude checking the enum]

Professor:

"Ah! Claude confirms: The Card model has the STRUCTURE for multiple choice (the `options` field exists), but:

- The CardType enum only has `BASIC`
- There's no UI for creating or displaying multiple choice
- The study logic doesn't handle multiple choice

This is actually a common scenario: someone started implementing a feature but didn't finish it. The database schema is ready, but the business logic and UI aren't.

Now let me define our requirements clearly:"

[TYPE in Claude Code]:

```
Here are the requirements for multiple choice questions:

1. A multiple choice card should have:
   - A question (prompt)
   - Four answer options (stored in the options field)
   - The correct answer (stored in the answer field as the text of the
correct option)
   - Optional explanation shown after answering

2. When creating a card:
   - User should be able to select "Multiple Choice" as the card type
   - Provide the question
   - Enter four options
   - Select which option is correct
   - Optionally add an explanation

3. When studying:
   - Display the question with four clickable options (not a flip card)
   - After selecting, show whether they were correct
   - Display the explanation if available
   - For the SRS algorithm, quality should be auto-graded: correct = 5,
incorrect = 2

Does this make sense given the current architecture?
```

[SCREEN: Shows Claude analyzing and responding]

Professor:

"Excellent! Claude confirms this approach makes sense and identifies potential considerations:

- The `options` field already exists in the database
- We need to update the CardType enum
- We need to modify the frontend form
- We need to update the study session UI
- We need to modify the answer grading logic

Now we can plan the implementation!"

PART 2: Ask Claude to Outline the Implementation (4 minutes)

[SCREEN: Claude Code interface]

Professor:

"Now that we understand WHAT we're building, let's ask Claude to help us plan HOW to build it. This is crucial - planning before coding saves time and prevents mistakes.

Planning Question:"

[TYPE in Claude Code]:

```
Help me plan the implementation for adding multiple choice cards.
```

```
Break down all the changes needed across:
1. Backend models and enums
2. Backend API/services
3. Frontend types
4. Frontend components
5. Database migrations (if any)

For each area, list the specific files that need to be modified and what
changes are needed.
```

[SCREEN: Shows Claude providing detailed implementation plan]

Professor:

"Look at this comprehensive plan! Let me walk through what Claude identified:

BACKEND CHANGES:

1. **Models & Enums** (backend/app/models/enums.py)
   - Add `MULTIPLE_CHOICE = 'multiple_choice'` to CardType enum
2. **Schemas** (backend/app/schemas/card.py)
   - Ensure CardCreate and CardUpdate schemas handle options field
   - Add validation for multiple choice cards
3. **Services** (backend/app/services/study.py)
   - Modify answer checking logic to handle multiple choice
   - Auto-grade: correct answer = quality 5, incorrect = quality 2
4. **Database Migration**
   - Claude notes: The Card model already has `options` field, so no migration needed!

FRONTEND CHANGES:

1. **Types** (frontend/src/types/api.ts or similar)
   - Ensure CardType type includes 'multiple_choice'
2. **Card Creation Form** (frontend/src/components/... or pages/...)
   - Add card type selector
   - Conditional form: show options inputs for multiple choice
   - Validation: ensure 4 options provided and one marked correct
3. **Study Session** (frontend/src/pages/StudySessionPage.tsx)
   - Different rendering for multiple choice cards
   - Show 4 buttons instead of flip card
   - Handle answer selection

    - Show result with explanation

This is exactly what we need! Notice how Claude organized it by domain (backend, frontend) and prioritized (models first, then services, then UI).

Let me ask Claude for the specific order of implementation:"

[TYPE in Claude Code]:

```
What order should I implement these changes in? Give me a step-by-step
sequence that minimizes dependencies and allows for testing at each step.
```

[SCREEN: Shows Claude's implementation sequence]

Professor:

"Perfect! Claude recommends:

STEP 1: Backend Foundation

1. Add MULTIPLE_CHOICE to CardType enum
2. Update schemas to validate options field
3. Test: Create a multiple choice card via API (Swagger UI)

STEP 2: Backend Study Logic 4. Modify study service to handle multiple choice grading 5. Test: Submit answer via API and verify grading

STEP 3: Frontend Types 6. Update TypeScript types for CardType

STEP 4: Frontend Card Creation 7. Add card type selector to form 8. Add options input fields (conditional rendering) 9. Test: Create a multiple choice card via UI

STEP 5: Frontend Study Display 10. Update StudySessionPage to render multiple choice differently 11. Handle option selection 12. Display results 13. Test: Study a deck with multiple choice cards

This is brilliant! We can implement and test incrementally. Each step is independently testable before moving to the next.

Now, let me ask one more planning question:"

[TYPE in Claude Code]:

```
Are there any edge cases or potential issues I should be aware of when
implementing this?
```

[SCREEN: Shows Claude identifying edge cases]

Professor:

"Claude identifies several important considerations:

1. **What if user submits an answer to a basic card with the new grading logic?**
   - Need to check card type before auto-grading
2. **What if options array doesn't have exactly 4 items?**
   - Need validation in backend schema
3. **What if the 'answer' field doesn't match any of the options?**
   - Validation: answer must be one of the options
4. **Existing basic cards shouldn't be affected**
   - Make sure we don't break existing functionality

These are things we might have missed! This is why planning with Claude is so valuable.

Now we're ready to implement. We have:

- Clear requirements
- Comprehensive plan
- Implementation sequence
- Edge cases identified

Let's write some code!"

PART B: Implementation (7 minutes)

PART 3: Use the Plan to Generate Implementation Code (5 minutes)

[SCREEN: Claude Code interface]

Professor:

"Now comes the implementation. We have our plan, so we can be very specific with Claude. We'll go step by step, following the sequence Claude recommended.

STEP 1: Update the CardType Enum

Let me write a prompt that references our plan:"

[TYPE in Claude Code]:

```
Following our implementation plan, let's start with Step 1: Add
MULTIPLE_CHOICE to the CardType enum.

Please:
1. Read backend/app/models/enums.py
2. Add MULTIPLE_CHOICE = "multiple_choice" to the CardType enum
3. Explain the change you made
```

[SCREEN: Shows Claude reading the file and making the change]

Professor:

"Perfect! Claude:

1. Read the current enums.py file
2. Added the new card type
3. Explained that it used 'multiple_choice' to match database enum naming conventions

Let me verify this change:"

[SCREEN: Open backend/app/models/enums.py in editor]

Professor:

"Yes, I can see the change. The CardType enum now has:

```python
class CardType(str, Enum):
    BASIC = 'basic'
    MULTIPLE_CHOICE = 'multiple_choice'
```

Great! Now let's continue:

STEP 2: Update Schemas for Validation"

[TYPE in Claude Code]:

```
Next step: Update backend/app/schemas/card.py to add validation for
multiple choice cards.

Requirements:
- If card type is MULTIPLE_CHOICE, the options field must be a list of
exactly 4 strings
- The answer field must be one of the options
- Add a Pydantic validator to check this

Show me the changes needed.
```

[SCREEN: Shows Claude reading schemas/card.py and proposing changes]

Professor:

"Excellent! Claude shows us how to add a Pydantic field validator. Look at this code:

```python
from pydantic import field_validator

class CardCreate(CardBase):
    @field_validator('options')
    def validate_multiple_choice_options(cls, v, info):
        if info.data.get('type') == CardType.MULTIPLE_CHOICE:
            if not v or len(v) != 4:
                raise ValueError('Multiple choice cards must have exactly 
4 options')
            if not all(isinstance(opt, str) and opt.strip() for opt in v):
                raise ValueError('All options must be non-empty strings')
        return v

    @field_validator('answer')
    def validate_multiple_choice_answer(cls, v, info):
        if info.data.get('type') == CardType.MULTIPLE_CHOICE:
            options = info.data.get('options', [])
            if v not in options:
                raise ValueError('Answer must be one of the provided 
options')
        return v
```

This ensures data integrity at the API level. Any invalid multiple choice card will be rejected before it reaches the database.

Let me apply this change:"

[SCREEN: Shows change being applied]

Professor:

"Good! Now let's test what we have so far:

TESTING THE BACKEND"

[TYPE in Claude Code]:

```
Let's test our backend changes. Help me:
1. Start the backend server (remind me of the command)
2. Give me a curl command or JSON to create a test multiple choice card 
via the API
```

[SCREEN: Shows Claude providing commands]

Professor:

"Claude gives us the commands. Let me run them:"

[SCREEN: Terminal]

```
cd backend
uvicorn app.main:app --reload --port 8000
```

[SCREEN: Another terminal, using the API]

Professor:

"Now let me open the Swagger UI at localhost:8000/docs and try creating a multiple choice card:"

[SCREEN: Browser showing Swagger UI]

Professor:

"I'll POST to `/api/v1/decks/{deck_id}/cards` with this payload:

```
{
  \"type\": \"multiple_choice\",
  \"prompt\": \"What is the capital of France?\",
  \"answer\": \"Paris\",
  \"options\": [\"London\", \"Berlin\", \"Paris\", \"Madrid\"],
  \"explanation\": \"Paris has been the capital of France since 987 AD.\"
}
```

[SCREEN: Shows successful response]

Perfect! It worked. Now let me try an INVALID one to test our validation:

```
{
  \"type\": \"multiple_choice\",
  \"prompt\": \"What is 2+2?\",
  \"answer\": \"Five\",
  \"options\": [\"Three\", \"Four\", \"Six\"]
}
```

[SCREEN: Shows 422 validation error]

Excellent! Our validation caught two errors:

1. Only 3 options (needs 4)
2. Answer 'Five' not in options

Our backend is solid. Let's move to the study logic:

STEP 3: Update Study Service for Grading"

[TYPE in Claude Code]:

```
Now let's update backend/app/services/study.py to auto-grade multiple
choice questions.

When a user answers a multiple choice card:
- If answer is correct: quality = 5
- If answer is incorrect: quality = 2

Update the record_answer function to handle this. Show me the specific
changes needed.
```

[SCREEN: Shows Claude analyzing study.py and proposing changes]

Professor:

"Claude shows us where to add the auto-grading logic. Look at the change in the `record_answer` function:

```
async def record_answer(
    db: Session,
    session: QuizSession,
    card: Card,
    user: User,
    answer_in: StudyAnswerCreate,
) -> tuple[QuizResponse, Optional[str]]:
    is_correct: bool | None = None
    quality = answer_in.quality

    # Auto-grade multiple choice
    if card.type == CardType.MULTIPLE_CHOICE and answer_in.user_answer:
        is_correct = (answer_in.user_answer == card.answer)
        quality = 5 if is_correct else 2

    # ... rest of function
```

This automatically grades multiple choice while leaving basic cards alone. Let me apply this:"

[SCREEN: Shows change being applied]

Professor:

"Perfect! Backend is complete. Now let's move to the frontend."

PART 4: Frontend Implementation (2 minutes)

[SCREEN: Claude Code interface]

Professor:

"For the frontend, we need to update the card creation form and the study session display. Let me ask Claude for help:

STEP 4: Update Card Creation Form"

[TYPE in Claude Code]:

```
Now let's update the frontend card creation form. I need to:
1. Add a card type selector (dropdown or radio buttons)
2. Conditionally show options inputs when multiple_choice is selected
3. Show which option is the correct answer

Find the card creation component and show me the changes needed. The
component is likely in frontend/src/pages/DeckDetailPage.tsx or a similar
location.
```

[SCREEN: Shows Claude finding and analyzing the component]

Professor:

"Claude found the form in DeckDetailPage.tsx. It shows us we need to add:

1. A select dropdown for card type
2. Conditional rendering for the options inputs
3. State management for the four options
4. Marking which option is correct

Rather than show you all the JSX code line by line (which would take too long), let me have Claude implement it:"

[TYPE in Claude Code]:

```
Please implement the card creation form changes for multiple choice
support. Add:
- Card type selector (basic or multiple_choice)
- Four input fields for options (shown only when multiple_choice selected)
- Radio buttons or similar to mark which option is correct
- Update the submit handler to properly format the data

Make sure the code follows React best practices and TypeScript typing.
```

[SCREEN: Shows Claude implementing the changes]

Professor:

"Claude implements the full form with proper state management, TypeScript types, and conditional rendering. The key changes:

```
const [cardType, setCardType] = useState<'basic' | 'multiple_choice'>
('basic');
const [options, setOptions] = useState<string[]>(['', '', '', '']);
const [correctIndex, setCorrectIndex] = useState(0);

// In the form:
{cardType === 'multiple_choice' && (
  <div>
    {options.map((option, index) => (
      <div key={index}>
        <input
          value={option}
          onChange={(e) => {
            const newOptions = [...options];
            newOptions[index] = e.target.value;
            setOptions(newOptions);
          }}
        />
        <input
          type=\"radio\"
          checked={correctIndex === index}
          onChange={() => setCorrectIndex(index)}
        />
      </div>
    ))}
  </div>
)}
```

Let me test this in the browser:"

[SCREEN: Browser showing the updated form]

Professor:

"Perfect! I can now:

1. Select 'Multiple Choice' from the dropdown
2. See four option input fields appear
3. Enter options and mark one as correct
4. Create the card

FINAL STEP: Update Study Session Display"

[TYPE in Claude Code]:

```
Last step: Update frontend/src/pages/StudySessionPage.tsx to display
multiple choice cards differently.

Instead of a flip card, show:
- The question
- Four clickable option buttons
- After clicking, show if correct/incorrect
- Display the explanation if available
- Then move to next card

Implement this with proper TypeScript and React patterns.
```

[SCREEN: Shows Claude implementing the study display]

Professor:

"Claude implements conditional rendering based on card type:

```
{currentCard.type === 'multiple_choice' ? (
  <div>
    <h2>{currentCard.prompt}</h2>
    <div>
      {currentCard.options?.map((option, index) => (
        <button
          key={index}
          onClick={() => handleMultipleChoiceAnswer(option)}
          disabled={answered}
        >
          {option}
        </button>
      ))}
    </div>
    {answered && (
      <div>
        {isCorrect ? '✓ Correct!' : '✗ Incorrect'}
        {currentCard.explanation && <p>{currentCard.explanation}</p>}
      </div>
    )}
  </div>
) : (
  // Original flip card for basic cards
  <FlipCard .../>
)}
```

Let me test the complete flow in the browser:"

[SCREEN: Browser showing study session with multiple choice card]

Professor:

"Excellent! Everything works:

1. Multiple choice card displays with four options
2. Clicking an option shows if correct
3. Explanation appears
4. Next card button appears
5. Basic cards still work with the flip animation

We've successfully implemented the feature!"

CLOSING (1 minute)

[SCREEN: Split screen showing before/after comparison]

Professor:

"Let's recap what we accomplished in this tutorial:

What We Built:

- Added multiple choice card type to the enum
- Implemented validation for multiple choice cards
- Auto-grading logic in the study service
- Card creation form with conditional UI
- Study session display for multiple choice
- All while maintaining backward compatibility with basic cards

How We Did It:

1. **Learned the Requirements** - Understood what we were building and why
2. **Asked Claude for a Plan** - Got a comprehensive implementation roadmap
3. **Implemented Step by Step** - Following the plan, we built incrementally
4. **Tested as We Went** - Verified each piece before moving forward

Key Takeaways:

- Planning before coding saves time
- Claude can help identify files, plan changes, and implement code
- Break large features into small, testable steps
- Always verify and test Claude's code
- Use specific prompts that reference your plan

This is the power of AI-assisted development: You stay in control of WHAT to build and HOW to design it, while Claude accelerates the implementation.

In our final tutorial, we'll cover tips and tricks for advanced Claude Code usage, common pitfalls to avoid, and how to debug when things go wrong.

See you next time!"

INSTRUCTOR NOTES:

Timing Breakdown:

- **Part A: Learning and Planning (7 min)**
  - Introduction: 1 min
  - Part 1 (Requirements): 2 min
  - Part 2 (Planning): 4 min
- **Part B: Implementation (7 min)**
  - Part 3 (Backend Implementation): 5 min
  - Part 4 (Frontend Implementation): 2 min
- Closing: 1 min
- **Total: ~15 minutes (Can be split into two 7-minute videos)**

Key Points to Emphasize:

1. Planning is MORE important than coding
2. Break features into small, testable increments
3. Claude understands context - reference previous discussions
4. Test backend before moving to frontend
5. Verify all changes manually

Preparation Needed:

- Have backend and frontend running
- Swagger UI bookmarked
- Sample test data ready
- Before/after screenshots prepared
- Testing checklist ready

Live Demo Critical Points:

- Must show actual API testing (Swagger UI or curl)
- Must show validation working (rejected bad data)
- Must show frontend form actually working
- Must show study session with multiple choice card

Common Issues During Implementation:

- TypeScript errors - acknowledge and show how to fix
- Validation errors - this is good! Shows validation working
- UI not updating - explain React state management
- If time is short - show plan in Part A, then show completed code in Part B

Suggested Split Point for Two Videos:

- **Video 4A: Learning and Planning** (Parts 1-2, ~7 min)
- **Video 4B: Implementation** (Parts 3-4, ~7 min)

Extension Ideas (if time permits):

- Add unit tests for the new validation
- Show how to write a test for the grading logic
- Demonstrate debugging with Claude when something breaks