Prompt Engineering Log & Reflection

Marketing Mix Modeling App Development

Project: Streamlit MMM Application with uv Package Manager
Date: December 2025
Context: Building a production-ready data science application from scratch

Effective Prompts Used (Categorized)

1. Constraint Prompting

Definition: Setting explicit boundaries, formats, and requirements to guide output precision.

Example Prompts:

- *"Create uv pyproject.toml and a concise README.md for a Streamlit MMM app. The README must include: install (uv), how to run, expected CSV schema (date, Sales, TV_Spend, Radio_Spend, Digital_Spend), and screenshots placeholders.* ***Keep it under 200 lines****"*
  - **Impact**: Prevented verbose documentation; forced prioritization of essential content
  - **Result**: 177-line README with all required sections
- *"Write a Python module snippet for a Streamlit app that either loads an uploaded CSV...or generates a synthetic weekly dataset with* ***seasonality and Gaussian noise****. Return a pandas DataFrame with* ***parsed date****"*
  - **Impact**: Specified exact data characteristics needed for realistic MMM testing
  - **Result**: 52-week synthetic data with Q4 seasonality peaks, proper datetime handling
- *"Generate a tests/test_basic.py with pytest smoke tests: (1) demo data shape and columns; (2) fitting function returns metrics and coefficients; (3) prediction from scenario planner function returns a float.* ***Avoid heavy dependencies****"*
  - **Impact**: Constrained test complexity while ensuring coverage
  - **Result**: 15 lightweight tests using only pytest, pandas, numpy

2. Stepwise Refinement

Definition: Iterative prompts that build on previous outputs, progressively adding features.

Example Prompts:

- **Step 1**: *"Create uv pyproject.toml and README"* → Initial project structure
- **Step 2**: *"Write a Python module snippet for data loading"* → data_loader.py created
- **Step 3**: *"Add to the Streamlit app: (1) Sales over time line chart (2) scatter plots (3) Spearman correlation matrix"* → Visualizations added
- **Step 4**: *"Extend the app with a fit_linear_mmm() function using scikit learn LinearRegression. Compute R², MAE, MAPE"* → Model integration
- **Step 5**: *"Add a Scenario Planner tab with number inputs for channel spend; compute predicted Sales and Δ vs baseline"* → Interactive planning

Impact: Each step validated before proceeding; errors caught early; complexity managed incrementally
Result: 548-line production app built without major rewrites

3. Role Prompting

Definition: Assigning specific expertise or perspective to shape response style and depth.

Implicit Role: Throughout the conversation, the AI operated as an **"expert AI programming assistant"** with **"expert-level knowledge across programming languages and frameworks"**

Effective Framing:

- Technical requirements stated as peer-to-peer collaboration (not tutorial requests)
- Assumed knowledge of marketing metrics (ROI, marginal returns, contribution analysis)
- Expected production-quality code with proper error handling and validation

Impact: Responses included:

- Industry-standard practices (hatchling build system, pytest fixtures)
- Performance considerations (Arrow serialization fixes, numeric-only operations)
- Best practices (type validation, fallback mechanisms, user-friendly formatting)

4. Output Format Specification

Definition: Explicitly defining the structure and format of expected responses.

Example Prompts:

- *"Return a pandas DataFrame with parsed date"* → Specified exact return type
- *"Compute channel wise totals and a ROI table with columns: **Total_Spend, Contribution = coef × total spend, Marginal_ROI = coef, Total_ROI = Contribution/Total_Spend**"* → Mathematical formulas provided
- *"Show a **formatted dataframe**"* and *"Show a **bar chart of coefficients**"* → Dual output requirements

Impact: Eliminated ambiguity about data structures and presentation
Result: ROI table with 4 columns exactly as specified; both tabular and visual outputs

5. Error-Driven Refinement

Definition: Using error messages as prompts to guide debugging and fixes.

Example Prompts:

- *"solve issues in this file"* (with deprecation warnings) → Fixed use_container_width → width='stretch'
- *"getting logs like these"* (Arrow serialization errors) → Added df['date'].dt.strftime('%Y-%m-%d')
- Build error "Unable to determine which files to ship" → Added [tool.hatch.build.targets.wheel]

Impact: Precise, targeted fixes without over-engineering
Result: All 7+ error types resolved with minimal code changes

6. Metric-Driven Validation

Definition: Requesting specific quantitative outputs to verify correctness.

Example Prompts:

- *"Compute **R², MAE, MAPE**"* → Three metrics explicitly named
- *"Display three metrics as **Streamlit metrics** and a **scatter chart** of Actual vs Predicted"* → Both numeric and visual validation
- *"compute **channel wise totals** and a **ROI table**"* → Aggregation requirements clear

Impact: Created measurable success criteria for model quality
Result: R^2=0.9527 (95% variance explained), MAPE=4.04% (highly accurate)

Reflection: What Improved Output Quality (380 words)

The most significant quality improvements came from **three interconnected prompt patterns**: constraint prompting, stepwise refinement, and output format specification.

Constraint prompting proved essential for managing scope and preventing feature creep. By specifying "under 200 lines" for the README, "avoid heavy dependencies" for tests, and exact column names for the ROI table, I eliminated ambiguity that would have required multiple revision cycles. The constraint "generate synthetic weekly dataset with seasonality and Gaussian noise" produced realistic test data on the first attempt—without it, I would have received generic random data unsuitable for time-series modeling. Constraints acted as **quality filters**, forcing prioritization of essential features.

Stepwise refinement enabled complexity management without overwhelming context windows. Rather than requesting a complete MMM application in one prompt (which would have produced incomplete or buggy code), I built iteratively: structure → data → visualizations → modeling → business intelligence → scenario planning. Each step validated previous work before adding new features. This approach surfaced errors early (deprecation warnings, Arrow serialization issues) when they were cheap to fix. The pattern mirrors professional software development: **working increments over big-bang releases**.

Output format specification eliminated the trial-and-error loop of vague requests. When I specified "Return a pandas DataFrame with parsed date," the function returned exactly that—not a list of dictionaries or JSON. When I defined ROI formulas explicitly ("Marginal_ROI = coef, Total_ROI = Contribution/Total_Spend"), the calculations matched finance textbook definitions. This precision was crucial for domain-specific work like marketing analytics, where "ROI" could mean return on investment, return on impact, or incremental ROI depending on context.

The synergy between these patterns created a compound effect. Constraints prevented scope creep, stepwise refinement managed complexity, and format specifications ensured each step's output integrated seamlessly with the next. For example, constraining data_loader.py to "return a pandas DataFrame" enabled app.py to consume it without adapters; specifying "fit_linear_mmm returns dict with model, coefficients, r2, mae, mape" allowed the scenario planner to access the fitted model directly.

The key insight: precision in prompting directly correlates with code quality. Vague requests like "build an MMM app" would have produced toy examples. Structured, constrained, iterative prompts with explicit formats produced a production-ready application with 95% model accuracy, comprehensive test coverage, and professional data visualizations—all in a single session without major rewrites.