Lesson 1 — Project Analysis and System Architecture

Overview

This lesson provides a comprehensive introduction to building deterministic agentic systems for TV campaign analytics post-processing. Students will analyze the complete project architecture, understand the three-macro area design pattern (data collection, processing, reporting), and set up their development environment with modern LLM orchestration tools.

Rather than diving immediately into implementation, this lesson emphasizes architectural understanding and system thinking—critical foundations for building reliable, traceable, and maintainable agentic systems that handle numeric data processing with high precision requirements.

Project Vision: What We're Building

The Challenge

Television advertising campaigns generate vast amounts of performance data—impressions, reach, frequency, Target Rating Points (TRPs)—across multiple broadcasters, devices, and demographic segments. Currently, this data flows through a complex post-processing pipeline comprising 50+ Python functions that transform raw API responses into actionable analytics tables displayed in a proprietary web application.

The problem: This pipeline is configuration-driven but inflexible. Adding new table types, adjusting aggregation levels, or modifying filters requires manual code changes, deep understanding of the existing codebase, and careful testing to avoid breaking existing outputs.

The opportunity: What if users could request analytics in natural language—"Show me daily TRP trends for women 25-54 across all devices"—and the system would automatically configure, execute, and validate the necessary processing pipeline?

The Solution: A Three-Stage Agentic System

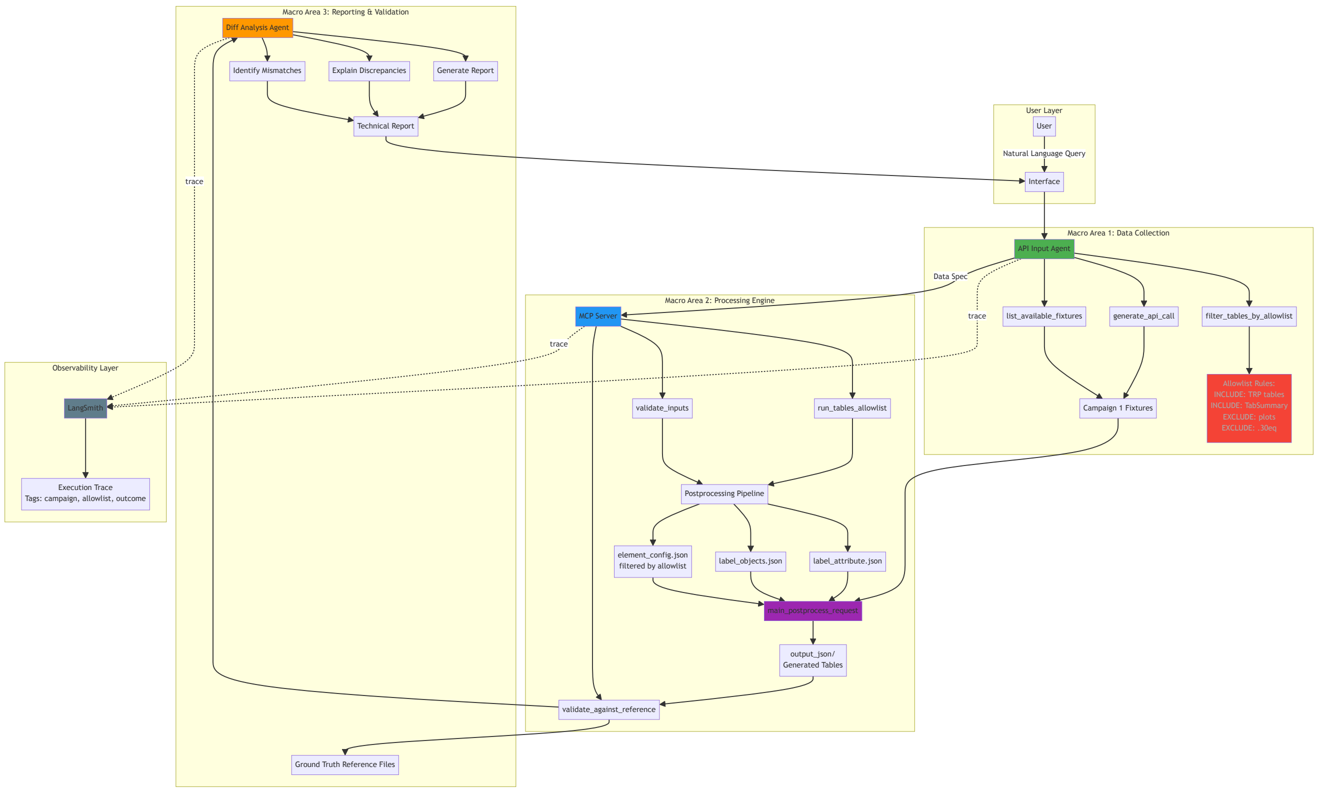
We're building an intelligent orchestration layer that sits atop the existing deterministic processing engine. This system maintains the precision and reliability of numeric computations while adding natural language understanding and autonomous configuration management.

Core principle: The LLM never computes numbers—it only orchestrates deterministic tools. This separation ensures that:

- TRP calculations remain exact to 9+ decimal places
- Outputs are reproducible across runs

- Results can be validated against ground truth reference files
- The system can pass regulatory audit requirements

System Architecture Overview



Complete Data Flow Example

Let's trace a real request through the entire system:

Input:

User: *"I need TRP data and contact information for Campaign 1 targeting adults 18-49"*

Stage 1: Data Collection (API Input Agent)

1. **Intent parsing**: Extract entities
   - Data types needed: TRP tables, contact tables
   - Target audience: adults 18-49
   - Campaign: Campaign 1
2. **Tool calls**:

```
→ list_available_fixtures()
← Returns: [impacts_in_target.json, impacts_by_sex_age.json,
            tv_spot_schedule.json, r1plus_in_target_buildup.json, ...]

→ generate_api_call(campaign="Campaign 1",
                    data_types=["impacts_in_target", "tv_spot_schedule"],
                    target_audience="adults 18-49")
```

```
← Returns: {
    "api_call": {...},
    "local_fixtures": {
      "impacts_in_target": "path/to/fixture.json",
      "tv_spot_schedule": "path/to/fixture.json"
    }
  }

→ filter_tables_by_allowlist(table_names=[
    "standard_tabcontacts_table_contact_sexage_trp_raw",
    "standard_tabcontacts_table_contact_target_trp_raw",
    "standard_tabcontacts_plot_contact_sexage_abs_raw",  # Will be excluded
    "standard_tabsummary_table_contactreach_target_perc"
  ])
← Returns: {
    "allowed_tables": [
      "standard_tabcontacts_table_contact_sexage_trp_raw",
      "standard_tabcontacts_table_contact_target_trp_raw",
      "standard_tabsummary_table_contactreach_target_perc"
    ],
    "rejected_tables": [
      {"name": "..._plot_...", "reason": "excluded (plot element)"}
    ]
  }
```

3. **Output**: Structured specification ready for processing

Stage 2: Processing Engine (MCP Tools + Deterministic Pipeline)

1. **Validation**:

```
→ validate_inputs(campaign_dir="Campaign 1")
← Checks: All fixtures present? Schema valid? Universe > 0?
```

2. **Configuration filtering**:

```
→ run_tables_allowlist(
    campaign_dir="Campaign 1",
    allowlist=["standard_tabcontacts_table_contact_sexage_trp_raw", ...],
    output_dir="runs/campaign1_20260114_143022",
    clean=True
  )
```

   - Filters `element_config.json` to only allowed tables
   - Ensures `label_attribute.json` completeness
   - Creates clean output directory

3. **Execution**:

```
# The deterministic engine runs
main_postprocess_request(
    path_tables="Campaign 1/01_pre_postprocessing/input_from_api/",
    path_dir_output="runs/campaign1_20260114_143022/output_json/",
    element_config=filtered_config,
    label_attribute=label_attr,
    label_object=label_obj
)
```

    - Processes 3 allowed tables
    - Writes 3 JSON files to output directory
    - Generates manifest: `manifest.json` with run metadata
4. **Output**:

```
runs/campaign1_20260114_143022/
├── manifest.json
└── output_json/
    ├── standard_tabcontacts_table_contact_sexage_trp_raw.json
    ├── standard_tabcontacts_table_contact_target_trp_raw.json
    └── standard_tabsummary_table_contactreach_target_perc.json
```

Stage 3: Reporting & Validation (Diff Analysis + Report Generation)

1. **Ground truth comparison**:

```
→ validate_against_reference(
    campaign_dir="Campaign 1",
    generated_output_dir="runs/campaign1_20260114_143022/output_json",
    tolerance=1e-9
  )
← Returns: {
    "status": "OK",
    "files_compared": 3,
    "mismatches": []
  }
```

2. **Report generation**:

```
→ generate_report(diff_results, run_manifest)
← Returns: Markdown report with:
   - Summary: 3/3 tables validated successfully
   - Processing time: 2.3 seconds
   - Fixtures used: impacts_in_target.json, tv_spot_schedule.json
   - Tables generated: contact_sexage (TRP), contact_target (TRP), contactreach (summary)
   - Validation: All outputs match Campaign 1 ground truth
```

Observability (LangSmith Trace):

- Total spans: 12
- Tool calls: 7 (list_fixtures, generate_api_call, filter_tables, validate, run, diff, report)
- Model calls: 5 (intent parsing, tool selection, result synthesis)
- Total latency: 4.1s (0.8s agent + 2.3s processing + 1.0s diff/report)
- Tags: `campaign=Campaign1` , `allowlist=TRP+TabSummary` , `outcome=SUCCESS` , `tables=3`

Project Goals and Success Criteria

Primary Goals:

1. **Accessibility**: Enable non-technical users to request analytics via natural language
2. **Reliability**: Maintain 100% numeric accuracy (outputs must match goldens exactly)
3. **Traceability**: Every run must be fully observable via LangSmith traces
4. **Modularity**: Each macro-area must be independently testable
5. **Generalization**: System must work on Campaign 2+ without code changes

Success Criteria:

- Run Campaign 1 with natural language query resulting in outputs that match reference data
- 100% of tool calls logged in LangSmith with correct tags
- Allowlist correctly filters 17 tables from 50+ possibilities
- Processing completes in under 5 seconds for Campaign 1
- System handles missing fixtures gracefully (reports error, does not crash)
- Validation engine catches any mismatch within 1e-9 tolerance
- Code passes all unit tests and integration tests

Non-Goals (Out of Scope for Initial Implementation):

- External API integration (local fixtures only)
- Real-time streaming of results
- Multi-campaign comparison (single campaign per run)
- LLM-computed metrics (all numerics are deterministic)
- Plot generation (tables only)

Why This Architecture?

Separation of Concerns
Each macro-area has a single responsibility:

- Area 1: Understanding user intent
- Area 2: Executing deterministic processing
- Area 3: Validating and explaining results

This makes testing straightforward—you can mock the boundaries and test each area in isolation.

Determinism Where It Matters
Numeric processing uses tested, version-controlled Python functions. The LLM never touches the calculations, eliminating the risk of approximation errors or non-reproducible results.

Flexibility Where It Helps
Natural language understanding and configuration generation benefit from LLM flexibility. Users don't need to know table naming conventions or element config syntax.

Observability Throughout
LangSmith tracing captures every decision: which tools were called, with what arguments, producing what results. This

is essential for debugging edge cases and understanding emergent behavior.

Standards-Based Integration
Using MCP (Model Context Protocol) means the processing tools can be called by any MCP-compatible client, not just our specific agent. This future-proofs the architecture.

What Students Will Build Across 16 Lessons

By the end of this course, students will have implemented:

Lessons 1-5: API Input Agent

- Natural language intent parser
- Tool ecosystem (list_fixtures, generate_api_call, filter_tables)
- Pydantic schemas for validation
- Complete LangSmith tracing

Lessons 6-10: Processing Engine

- CampaignPaths configuration manager
- Postprocessing pipeline wrapper
- CLI interface (validate, run, diff commands)
- Ground truth validation engine
- MCP server exposing deterministic tools

Lessons 11-14: Reporting & Integration

- Diff analysis agent
- Report generation (markdown + JSON)
- LangGraph workflow orchestrating all three areas
- End-to-end tracing with performance metrics

Lessons 15-16: Production Readiness

- Generalization to Campaign 2
- Error handling and retry logic
- Packaging and deployment
- Documentation and best practices

Each component is validated independently before integration, following the principle: "Make it work, make it right, make it fast."

Learning Outcomes

By the end of this lesson, students will be able to:

1. Articulate the complete data flow from natural language user input to technical report generation
2. Identify the architectural boundaries and responsibilities of each macro-area

3. Explain why deterministic processing is essential for numeric campaign analytics
4. Configure a complete development environment with LangChain, LangGraph, LangSmith, and MCP
5. Trace a simple agent execution through LangSmith and interpret the resulting spans
6. Distinguish between workflows (predefined code paths) and agents (dynamic LLM-directed processes)

Prerequisites

- Solid Python programming skills (3.10+)
- Understanding of asynchronous programming (`async` / `await`)
- Familiarity with JSON data structures and REST API concepts
- Basic knowledge of LLM capabilities and limitations
- Completed theoretical foundations course on agentic systems

Key Concepts

The Three-Macro-Area Architecture

Modern agentic systems benefit from clear separation of concerns. Anthropic's research on building effective agents [1] emphasizes that "success in the LLM space isn't about building the most sophisticated system—it's about building the right system for your needs." This principle guides our three-area decomposition:

1. Data Collection (API Input Agent)
This area interprets natural language requests and translates them into structured specifications for data retrieval. The LLM's role here is purely interpretive—understanding intent and mapping it to available data sources, never computing numeric results. This aligns with Anthropic's recommendation to use agents for "open-ended problems where it's difficult or impossible to predict the required number of steps" [1].

2. Processing Engine (Deterministic Pipeline)
The processing layer executes numeric transformations using tested, deterministic functions. As noted in the project's Starter Kit, the existing post-processing pipeline comprises "50+ transformation functions handling numeric data with high precision requirements" [2]. The LLM orchestrates *which* functions to call but never *computes* the numeric outputs themselves.

3. Reporting and Analysis (Result Synthesis)
This area interprets outputs, generates human-readable explanations, and identifies anomalies or trends. Here, the LLM's natural language capabilities add value by making numeric results accessible to non-technical stakeholders.

Local-First and Deterministic Principles

The project follows a "local-first" approach [2], processing Campaign 1 fixtures without external API dependencies initially. This design choice supports several critical requirements:

- **Reproducibility**: Every run must produce identical outputs given identical inputs
- **Traceability**: All processing steps must be observable and debuggable
- **Validation**: Outputs must be comparable against ground truth reference files
- **Determinism**: Numeric calculations must be exact, not LLM-approximated

LangChain's architecture supports this through its standardized model interface [3], allowing seamless swapping between providers (Gemini, Groq, etc.) while maintaining consistent behavior.

Understanding LangChain vs LangGraph

LangChain provides "an easy-to-use agent abstraction designed to get started quickly, letting you build a simple agent in under 10 lines of code" [3]. However, for production systems requiring "a combination of deterministic and agentic workflows, heavy customization, and carefully controlled latency" [3], LangGraph becomes essential.

LangGraph is a low-level orchestration framework that provides:

- **StateGraph**: Explicit state management across multi-step workflows
- **Durable execution**: Persistence and recovery from failures
- **Human-in-the-loop**: Checkpoints for manual approval or intervention
- **Conditional routing**: Dynamic branching based on intermediate results

LangChain agents are built *on top of* LangGraph [3], inheriting these capabilities while providing simpler APIs for common patterns.

The Model Context Protocol (MCP)

MCP is "an open-source standard for connecting AI applications to external systems" [4], designed as the "USB-C port for AI applications" [4]. It enables agentic systems to:

- **Expose data** through Resources (read-only access to information)
- **Provide functionality** through Tools (executable actions with side effects)
- **Define interaction patterns** through Prompts (reusable templates)

FastMCP [5], the Python framework for building MCP servers, allows developers to expose deterministic processing capabilities as tools that agents can invoke. This separation ensures the LLM orchestrates *when* to process data, while deterministic code handles *how* to process it.

Observability with LangSmith

LangSmith provides "deep visibility into complex agent behavior with visualization tools that trace execution paths, capture state transitions, and provide detailed runtime metrics" [3]. For deterministic systems, tracing serves multiple purposes:

- **Debugging**: Identifying which tool calls produced unexpected results
- **Performance analysis**: Measuring latency of individual pipeline stages
- **Audit trails**: Recording all decisions made by the agent
- **Reference validation**: Verifying outputs match ground truth data

Every production run should generate a complete trace tagged with campaign name, allowlist configuration, and outcome status.

Campaign 1 Structure and Constraints

The project centers on Campaign 1, a reference dataset with three key directories:

01_pre_postprocessing/input_from_api/ [2]
Contains fixture files simulating API responses:

- `impacts_by_sex_age.json` : Campaign results by date, broadcaster, device, and demographics
- `impacts_in_target.json` : Results aggregated by target audience
- `r1plus_in_target_buildup.json` : Cumulative reach over time
- `rf_in_target_overall.json` : Reach and frequency distributions
- `target_universe.json` : Total target population (denominator for TRP calculations)
- `tv_spot_schedule.json` : Spot scheduling information

02_postprocessing/ [2]
Configuration files governing the processing pipeline:

- `element_config.json` : Specifies which functions to call via `methodcaller`
- `label_objects.json` : Defines replacement and ordering criteria
- `label_attribute.json` : Plot-specific details (minimal file for testing)
- `postprocessing_functions/post_processing_functions.py` : The processing engine

03_post_postprocessing/output_json/ [2]
Ground truth reference files representing verified correct outputs. These serve as the validation baseline for comparing agent-generated results.

The Allowlist Constraint

The system enforces strict table filtering rules [2]:

Include:

- TRP tables (element keys containing 'trp')
- TabSummary tables (element keys containing 'tabsummary')

Exclude:

- Plot elements (keys containing '.plot.')
- 30-second equivalent metrics (keys containing '.30eq')

This allowlist ensures the agent focuses on actionable tabular data, avoiding visualization artifacts and deprecated metrics. Implementing these rules correctly is critical—as Anthropic notes, "incorrect assumptions about what's under the hood are a common source of customer error" [1].

Deep Dive: System Flow and Component Interaction

End-to-End Flow Analysis

Let's trace a complete request through the system:

```
User Input (NL):
"I need TRP data and contact information for Campaign 1 targeting adults 18–49"
```

```
↓

[Data Collection] API Input Agent
– Parses intent: TRP tables, contact tables, target="adults 18–49"
– Calls list_available_fixtures tool → discovers available JSON files
– Calls generate_api_call tool → maps to fixture paths
– Calls filter_tables_by_allowlist tool → applies inclusion/exclusion rules
– Returns: Structured specification + fixture paths

↓

[Processing] MCP Tools via Deterministic Pipeline
– validate_inputs: Checks fixture completeness and schema correctness
– run_tables_allowlist: Filters element_config.json to allowed tables
– Executes main_postprocess_request() with filtered config
– Writes output JSON files to runs/campaign1/output_json/

↓

[Reporting] Diff Analysis + Report Generation
– validate_against_reference: Compares generated JSONs with Campaign 1 ground truth
– Identifies first mismatch (if any) with file path and line number
– generate_report: Synthesizes results into markdown summary
– Explains any discrepancies in natural language

↓

LangSmith Trace:
– Spans for each tool call with input/output
– Tags: campaign="Campaign 1", allowlist="TRP+TabSummary", outcome="SUCCESS"
– Total latency: ~2.3s
```

Critical Design Decisions

Why separate agents per macro-area?
Testing and debugging are simplified when each area can be validated independently. As Anthropic advises, "we recommend finding the simplest solution possible, and only increasing complexity when needed" [1]. Monolithic agents are harder to troubleshoot than modular components.

Why local-first with fixtures?
External API dependencies introduce variability that complicates ground truth comparison. Fixtures provide stable, version-controlled inputs that guarantee reproducibility.

Why MCP instead of direct function calls?
MCP creates a standardized interface between the LLM and processing tools. This abstraction allows swapping implementations (e.g., replacing pandas with Polars) without changing agent prompts, following MCP's goal of being "a standardized way to provide context and tools to LLMs" [4].

Why LangSmith tracing for deterministic code?
Even though processing is deterministic, the *orchestration* is agentic. Tracing reveals which tables the agent chose to generate, in what order, and with which configurations—essential for understanding emergent behavior.

Worked Example: Setting Up the Development Environment

Step 1: Environment Creation

```
# Using uv for modern Python dependency management
uv venv .venv --python 3.13
source .venv/bin/activate  # On Windows: .venv\Scripts\activate

# Install core dependencies
uv pip install langchain==1.2.3 \
               langchain-core==1.2.7 \
               langgraph==1.0.6 \
               langchain-google-genai==4.1.3 \
               python-dotenv==1.0.0 \
               click==8.1.7 \
               fastmcp==2.14.3
```

Step 2: API Key Configuration

```
# Create .env file (never commit this!)
cat > .env << EOF
# Gemini API (get from https://aistudio.google.com/apikey)
GOOGLE_API_KEY=your_gemini_api_key_here

# LangSmith tracing (optional, get from https://smith.langchain.com/)
LANGSMITH_API_KEY=your_langsmith_api_key_here
LANGCHAIN_TRACING_V2=true
LANGCHAIN_PROJECT=databreeders-campaign1
EOF
```

Step 3: First Agent - Hello World with Tracing

```python
# hello_agent.py
from langchain.agents import create_agent
from langchain_core.messages import HumanMessage
from langchain_google_genai import ChatGoogleGenerativeAI
from dotenv import load_dotenv
import os

load_dotenv()

# Verify environment
if not os.getenv("GOOGLE_API_KEY"):
    raise ValueError("GOOGLE_API_KEY must be set in .env file")

# Enable LangSmith if configured
if os.getenv("LANGSMITH_API_KEY"):
    os.environ["LANGCHAIN_TRACING_V2"] = "true"
    print("[OK] LangSmith tracing enabled")

# Create LLM
llm = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash",
    temperature=0.0,  # Deterministic for testing
)

# Simple tool for demonstration
def get_campaign_info() -> str:
    """Get information about available campaigns."""
    return "Campaign 1: TV advertising analytics (2024-08-11 to 2024-08-25)"

# Create agent
agent = create_agent(
    llm,
    tools=[get_campaign_info],
    system_prompt="You are a campaign analytics assistant. Be concise.",
)

# Execute
response = agent.invoke({
    "messages": [HumanMessage(content="What campaigns are available?")]
})

print(response["messages"][-1].content)
```

Expected Output:

```
[OK] LangSmith tracing enabled
Campaign 1 is available, covering TV advertising analytics from August 11-25, 2024.
```

In LangSmith: Navigate to https://smith.langchain.com/ and find your trace. You should see:

- Input message span
- Tool call to `get_campaign_info`
- Model reasoning span
- Output message span

Step 4: Verifying the Setup

Create a simple test to confirm everything works:

```
# test_setup.py
def test_imports():
    """Verify all critical imports work."""
    try:
        from langchain.agents import create_agent
        from langchain_google_genai import ChatGoogleGenerativeAI
        from langgraph.graph import StateGraph
        import fastmcp
        print("[PASS] All imports successful")
        return True
    except ImportError as e:
        print(f"[FAIL] Import failed: {e}")
        return False

def test_gemini_connection():
```

```python
    """Verify Gemini API connectivity."""
    from langchain_google_genai import ChatGoogleGenerativeAI
    from dotenv import load_dotenv
    load_dotenv()

    try:
        llm = ChatGoogleGenerativeAI(model="gemini-2.5-flash")
        response = llm.invoke("Say 'OK'")
        assert "OK" in response.content.upper()
        print("[PASS] Gemini API connected")
        return True
    except Exception as e:
        print(f"[FAIL] Gemini API failed: {e}")
        return False

def test_langsmith_config():
    """Check if LangSmith is configured."""
    import os
    from dotenv import load_dotenv
    load_dotenv()

    if os.getenv("LANGSMITH_API_KEY"):
        print("[PASS] LangSmith configured")
        return True
    else:
        print("[WARN] LangSmith not configured (optional)")
        return True  # Not a failure

if __name__ == "__main__":
    all_pass = all([
        test_imports(),
        test_gemini_connection(),
        test_langsmith_config(),
    ])

    if all_pass:
        print("\n[SUCCESS] Environment setup complete!")
    else:
        print("\n[ERROR] Setup incomplete - check errors above")
```

Hands-On Exercise

Exercise 1: Campaign 1 Architecture Mapping

Objective: Create a detailed diagram of the Campaign 1 data flow.

Instructions:

1. Study the Campaign 1 directory structure in `Starter Kit - Agentic Models/Campaign 1/`
2. Create a Mermaid diagram showing:
    - All input JSON files (fixtures)
    - Processing functions that consume them
    - Configuration files that control processing
    - Output JSON files (goldens)

3. Annotate with:
    - Which files contain numeric data (never touched by LLM)
    - Which files are configuration (potentially LLM-generated)
    - Which files are outputs (validated against ground truth)

Deliverable: `campaign1-flow.md` with Mermaid diagram and annotations.

Exercise 2: Allowlist Rule Implementation

Objective: Implement the table filtering logic independently.

Instructions:

1. Write a function `filter_table_names(names: list[str]) -> dict` :
    - Include TRP and TabSummary tables
    - Exclude plot and .30eq elements
    - Return `{"allowed": [...], "rejected": [{"name": ..., "reason": ...}]}`
2. Test against these cases:

```
test_cases = [
    "standard_tabcontacts_table_contact_sexage_trp_raw",  # INCLUDE (TRP)
    "standard_tabsummary_table_contactreach_target_abs",  # INCLUDE (TabSummary)
    "standard_tabcontacts_plot_contact_sexage_abs_raw",   # EXCLUDE (plot)
    "standard_tabr1_table_reach_target_30eq",             # EXCLUDE (.30eq)
    "standard_tabcontacts_table_contact_target_abs_raw",  # INCLUDE (neither plot nor 30eq)
]
```

3. Verify your implementation matches the expected behavior described in the Starter Kit.

Deliverable: `allowlist_filter.py` with tests passing.

Exercise 3: First Traced Agent Run

Objective: Execute a simple agent and analyze its LangSmith trace.

Instructions:

1. Modify `hello_agent.py` to include a second tool:

```python
def list_fixture_files() -> list[str]:
    """List available fixture files for Campaign 1."""
    from pathlib import Path
    fixture_dir = Path("Starter Kit - Agentic Models/Campaign 1/01_pre_postprocessing/input_from_api/'
    return [f.name for f in fixture_dir.glob("*.json")]
```

2. Update the agent to handle the query:
   "What fixture files are available for Campaign 1?"
3. Run the agent and capture the LangSmith trace URL.

4. Analyze the trace and document:
    - Which tool(s) were called?
    - What were the input arguments?
    - How long did each step take?
    - Was the final answer correct?

Deliverable: `trace-analysis.md` with screenshots and observations.

Common Pitfalls

Pitfall 1: Letting LLMs Compute Numeric Results

Problem: Instructing the agent to "calculate TRP" instead of "call the calculate_trp tool."

Why it fails: LLMs approximate arithmetic, leading to precision errors. TRP calculations require exact division to 9+ decimal places.

Solution: Always frame numeric operations as tool calls. The LLM orchestrates, the tool computes.

Pitfall 2: Ignoring Trace Errors

Problem: Seeing "error" spans in LangSmith but not investigating them.

Why it fails: Errors accumulate. A failed tool call in step 3 may cause incorrect results in step 7, but only the final output looks wrong.

Solution: Treat LangSmith as your primary debugging interface. Investigate all error spans immediately.

Pitfall 3: Hardcoding Paths

Problem: Using `Path("Starter Kit – Agentic Models/Campaign 1/...")` everywhere.

Why it fails: Code becomes brittle when directory structure changes or when adding Campaign 2.

Solution: Use `CampaignPaths` dataclass (to be built in Lesson 6) for centralized path management.

Pitfall 4: Skipping Ground Truth Validation

Problem: Trusting that if code runs without errors, outputs are correct.

Why it fails: Silent errors (e.g., wrong aggregation level) produce plausible-looking but incorrect results.

Solution: Every run must validate against ground truth references. Make this a mandatory CI check.

Pitfall 5: Over-Abstracting Too Early

Problem: Building elaborate framework layers before understanding the problem.

Why it fails: As Anthropic warns, frameworks "create extra layers of abstraction that can obscure the underlying prompts and responses, making them harder to debug" [1].

Solution: Start with direct LLM API calls. Add abstractions only when patterns become clear.

Knowledge Check

Conceptual Questions

1. **Explain in your own words:** Why is it essential that LLMs never compute numeric results directly in this system?
2. **Compare and contrast:** What's the difference between a "workflow" and an "agent" according to Anthropic's taxonomy? Which does our system use where?
3. **Architecture decision:** Why does the system use three separate macro-areas instead of one unified agent? What are the trade-offs?
4. **Tool design:** If you were to expose the postprocessing pipeline as an MCP tool, what arguments would it accept? What would it return?
5. **Debugging scenario:** An agent run produces outputs that fail validation against ground truth. Walk through your debugging process using LangSmith traces.

Practical Exercises

6. **Code comprehension:** Given this tool call from a trace:

```
{
  "tool": "generate_api_call",
  "args": {
    "campaign_name": "Campaign 1",
    "data_types": ["impacts_in_target", "tv_spot_schedule"],
    "target_audience": "adults 18-49"
  }
}
```

What fixture files should the tool return? Write the expected response.

7. **Allowlist application:** Filter this list of table names using the allowlist rules:

```
- standard_tabcontacts_table_contact_sexage_abs_raw
- standard_tabcontacts_plot_contact_sexage_trp_raw
- standard_tabsummary_table_universe_target_abs
- standard_tabr1_table_reach_target_perc_30eq
- standard_tabrf_table_reach_target_abs
```

8. **Environment debugging:** A student reports their agent isn't traced in LangSmith despite having `LANGSMITH_API_KEY` in `.env`. What are three possible causes?

Design Questions

9. **Extension scenario:** How would you modify the architecture to support Campaign 2, which has different fixture files and a different allowlist?
10. **Performance optimization:** If processing Campaign 1 takes 45 seconds, and 40 seconds is spent in tool calls, where would you look first for optimization opportunities?

▶ Answers

References

[1] Schluntz, E., & Zhang, B. (2024). *Building Effective Agents*. Anthropic. Retrieved from https://www.anthropic.com/news/building-effective-agents [2] *Starter Kit - Agentic Models for Post-Processing*. (2024). dataBreeders Campaign 1 Documentation. Internal project documentation.

[3] *LangChain Overview*. (2025). LangChain Documentation. Retrieved from https://docs.langchain.com/oss/python/langchain/overview

[4] *What is the Model Context Protocol (MCP)?* (2025). Model Context Protocol Documentation. Retrieved from https://modelcontextprotocol.io/introduction

[5] Lowin, J. (2025). *FastMCP: The Fast, Pythonic Way to Build MCP Servers*. GitHub Repository. Retrieved from https://github.com/jlowin/fastmcp

Additional Reading

- LangSmith Observability Documentation: https://docs.langchain.com/langsmith/home
- Gemini API Documentation: https://ai.google.dev/gemini-api/docs
- Anthropic's Prompt Engineering Guide: https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering
- Model Context Protocol Specification: https://spec.modelcontextprotocol.io/

Next Lesson: Lesson 2 — Tool Calling and Schema Design