Course Syllabus - Building Deterministic Agentic Systems

Format: Project-based workshop with incremental validation
Duration: 16 lessons × 2h = 32h total
Approach: Analysis, design and guided implementation with verification criteria
Stack: LangChain/LangGraph + Gemini/Groq + LangSmith + MCP
Validation: Testing on Campaign 1 fixtures → generalization to new datasets

Project Overview

This course guides students through building a complete agentic system structured in three macro-areas:

1. **Data Collection** - Interpreting natural language requests and identifying required data
2. **Processing** - Executing deterministic processing pipelines with MCP tools
3. **Reporting** - Analyzing results and generating technical reports

Each area is developed and tested independently, then integrated into a unified, autonomous pipeline. The system is validated first on known data (Campaign 1 fixtures with golden outputs), then generalized to unseen datasets.

Module 1 — API Input Agent (Lessons 1-5)

Focus: Building an agent that interprets natural language requests and identifies necessary data

Lesson 1: Project Analysis and System Architecture

What students explore:

- Complete project vision: from user input to final report
- Analysis of the three macro-areas (data collection, processing, reporting)
- Study of Campaign 1 structure: fixtures, configs, goldens
- Understanding the flow: NL query → tool calls → processing → diff → report
- Guiding principles: local-first, traceability, reproducibility
- Identifying constraints: table allowlist, exclusions, golden comparison

What students build:

- Conceptual map of the entire system
- Flow diagrams for each macro-area
- Complete environment setup (LangChain, LangGraph, LangSmith, API keys)

- First basic agent with LangSmith tracing

How to verify:

- Environment correctly configured
- Base agent executable with visible trace in LangSmith
- Documented understanding of component dependencies

Lesson 2: Tool Calling and Schema Design

What students build:

- Pydantic schemas for input validation
- Structured tools implementation with StructuredTool
- LangGraph compatibility handling (kwargs flattening)
- `list_available_fixtures` tool that explores available fixtures

How to verify:

- Tool returns complete and correct list of Campaign 1 fixtures
- Schema validation works on both valid and invalid inputs
- Tool is callable by agent via LangGraph

Lesson 3: Allowlist Rules and Filtering Logic

What students build:

- Implementation of inclusion rules (TRP, TabSummary)
- Implementation of exclusion rules (plot, .30eq)
- Tool to filter lists of table names
- Edge case handling (different notations: dot vs underscore)

How to verify:

- Tests on mixed lists: plot and 30eq elements are excluded
- TRP and TabSummary elements are correctly included
- Robustness against syntactic variations in names

Lesson 4: Mock API Call Generation

What students build:

- Mapping from natural language requests to specific data types
- Generation of simulated API calls with local fixture references
- Handling of missing fixtures with alternative suggestions
- `generate_api_call` tool that produces complete specifications

How to verify:

- Request like "TRP data for adults 18-49" produces correct paths

- Unavailable fixtures generate informative messages
- Output includes complete metadata (campaign, target audience, etc.)

Lesson 5: Agent Orchestration and Message Handling

What students build:

- Complete agent using LangGraph's create_agent
- AIMessage handling with heterogeneous content (strings, lists, blocks)
- JSON-safe output serialization
- End-to-end system: query → tool calls → structured response

How to verify:

- Query "Download TRP and contact data for Campaign 1" produces correct output
- All tool calls are traced in LangSmith
- Output is serializable and contains complete information

Module 2 — Processing Engine (Lessons 6-10)

Focus: Building the processing system with MCP tools and postprocessing pipeline

Lesson 6: Path Management and Configurations

What students build:

- CampaignPaths class for managing campaign directory structure
- Loading and validation of configurations (element_config, label_objects, label_attribute)
- Dynamic filtering of element_config based on allowlist
- Helpers for configuration completeness checks

How to verify:

- CampaignPaths correctly identifies all Campaign 1 paths
- Loaded configs are valid and complete
- Filtering produces configurations consistent with allowlist

Lesson 7: Postprocessing Pipeline Wrapper

What students build:

- Wrapper around existing `main_postprocess_request`
- Logic to ensure label_attribute completeness
- System for writing output to dedicated directory
- Run manifest generation (inputs, applied allowlist, used configs)

How to verify:

- Run generates JSON output files in specified directory
- Manifest includes all information necessary to reproduce the run
- Pipeline completes without errors on Campaign 1

Lesson 8: CLI Interface with Click

What students build:

- Multi-command CLI (validate, run, diff)
- Error handling with clear user messages
- Progress reporting during execution
- Integration with runner built in previous lesson

How to verify:

- `databreeders run --campaign "..." --out runs/test` completes correctly
- Errors produce understandable messages
- Help messages are informative

Lesson 9: Golden Comparison System

What students build:

- Module for comparing generated JSONs vs reference goldens
- Logic to identify first mismatch per file
- Tolerance handling for floating point values
- Structured difference reporting

How to verify:

- Diff on correct run produces positive outcome
- Diff on modified file detects and reports mismatch
- Tolerance correctly handles numerical rounding

Lesson 10: MCP Server with FastMCP

What students build:

- MCP server exposing processing tools
- Resources for browsing: fixtures, goldens, element catalog
- Tools: validate_inputs, run_tables_allowlist, diff_vs_goldens
- Integration with existing CLI logic

How to verify:

- MCP tools are callable from standard client
- MCP tool output is identical to CLI output
- Resources are navigable and complete

Module 3 — Reporting and Integration (Lessons 11-14)

Focus: Building the results analysis system and integrating the three macro-areas

Lesson 11: Diff Analysis Agent

What students build:

- Agent specialized in interpreting diff results
- Generation of understandable explanations for discrepancies
- Logic to prioritize errors (focus on first failure)
- Debugging suggestions based on common patterns

How to verify:

- Given diff with mismatch, agent clearly explains the cause
- Explanations are human-readable and actionable
- Prioritization helps quickly identify the problem

Lesson 12: Technical Report Generation

What students build:

- Tool for producing reports in markdown/JSON format
- Metrics aggregation: counts of generated tables, successes, failures
- Inclusion of links to files with issues
- Templates for different report types (summary, detailed, debug)

How to verify:

- Report contains all relevant information
- Format is readable and navigable
- Metrics are accurate

Lesson 13: Workflow Composition with LangGraph

What students build:

- Multi-step workflow using StateGraph
- Nodes: interpret_request → run_processing → diff_analysis → generate_report
- Conditional edges for error handling and alternative branches
- State management across steps

How to verify:

- End-to-end workflow from NL request to final report works
- Errors in one step are handled correctly
- State is consistent across transitions

Lesson 14: Complete Tracing and Debugging

What students build:

- LangSmith spans for each workflow step
- Automatic tags with metadata (campaign, allowlist, outcome)
- Debug playbook for common errors
- Dashboard for performance and correctness analysis

How to verify:

- Each run produces complete trace in LangSmith
- Metadata is accurate and useful for debugging
- Playbook covers most frequent error cases

Module 4 — Generalization and Consolidation (Lessons 15-16)

Focus: Testing on new data and consolidating best practices

Lesson 15: Adaptation to New Campaigns

What students build:

- System adaptation for Campaign 2 (unseen data)
- Validation of allowlist and config management robustness
- Testing without available goldens (smoke tests)
- Identification of fragility points and hardcoding

How to verify:

- System works on Campaign 2 without code modifications
- Outputs are plausible (no crashes)
- Any hardcoded assumptions are identified and documented

Lesson 16: Production Readiness and Consolidation

What students build:

- Robust error handling (retry logic, fallback strategies)
- Structured logging and observability
- Complete packaging (pyproject.toml, installation, optional containerization)
- Complete documentation and consolidated best practices
- Maintainability principles: reversibility, small changes, test-first approach

How to verify:

- System is installable by third parties
- Executable on different machines without manual setup
- Documentation enables rapid onboarding
- Test suite is complete and maintainable

Cross-Cutting Verification Criteria

Each component is verified through:

1. **Unit tests**: isolated component works correctly
2. **Integration tests**: interactions with upstream/downstream components
3. **Golden comparison**: output compared with references (where applicable)
4. **Trace analysis**: LangSmith traces are complete and informative

Structure of Each Lesson

Each session provides:

- **Problem analysis**: understanding context and objectives
- **Architectural constraints**: rules to respect (e.g., "tool must accept **kwargs")
- **Business rules**: specific logic (e.g., "exclude plot and .30eq")
- **Success criteria**: how to verify correctness
- **Implementation directions**: suggested ideas and patterns (not complete code)
- **Anti-patterns**: approaches to avoid with justifications

Students implement autonomously, instructor facilitates and validates.

Pedagogical Approach

The course follows a **workshop model** where:

- Theory has already been covered; focus is on practical implementation
- Each lesson presents challenges, not solutions
- Students must think critically about design decisions
- Validation happens incrementally at each step
- The final system must work on both known (Campaign 1) and unknown (Campaign 2) data

Success is measured by:

- Code that passes verification criteria
- System that generalizes to new datasets
- Understanding of trade-offs and design decisions

- Ability to debug and maintain the system independently