Agent-Centric ML Platform

Implementation Plan

A Reproducible, Governed ML Infrastructure for Regulated Finance

Version V2

February 9th 2026

1. Purpose and Overview

This document is a step-by-step implementation plan for an agent-centric ML platform designed for regulated banking and financial consulting. It consolidates a high-level architecture plan, a technical specification, and expert review amendments into a single, actionable guide.

The system serves a small team with high expected churn. One person will likely be the sole long-term operator, with part-time collaborators rotating in and out. This drives three overriding design priorities:

- **Reproducibility:** Every experiment must be fully traceable and re-runnable by anyone, at any time.
- **Logging and auditability:** If it isn't logged, it didn't happen. Logging is the backbone of institutional memory.
- **Simplicity:** Every component must justify its complexity. If a simpler approach works, use it. This mandate overrides all other considerations.

The platform integrates AI agents (starting with Claude Code) as controlled research collaborators. Agents can propose and run experiments, but they operate within strict boundaries: a single experiment entrypoint, protected configuration fields, data governance rules, and version-controlled workflows. The system enforces these boundaries mechanically (entrypoint checks, pre-commit hooks, pre-merge hooks), not by relying on agent compliance with instructions alone.

Target horizon: 2–3 years. The plan builds incrementally from a minimal working system to a full-featured platform.

2. Build Phases

The system is built in three phases. Each phase ends with a testable checkpoint: a concrete capability you can verify before moving on.

Phase	Scope	Duration	Checkpoint
Minimal	Entrypoint, config validation, MLflow, dependency hash, basic tests, AGENTS.md with risk checklist	1–2 days	Run an experiment, see it logged in MLflow
Core	Content-addressed data, feature hashing, protected fields, pre-commit hooks, training guard, git workflow with pre-merge verification, full tests	~1 week	Agent runs experiment within all constraints
Full	Copier template, reasoning logs, cost tracking, model promotion + model cards, Dockerfile, backup, cross-project queries	Following weeks	New client project from template with all features

Each phase is detailed in its own section below (Sections 5, 6, and 7). Sections 3 and 4 cover architectural decisions and repository structure.

3. Architectural Decisions

This section documents key design choices and their reasoning. It serves as a reference for future decisions and for reviewers.

3.1 Agent Tooling: Claude Code Only (For Now)

The platform starts with Claude Code as the sole AI agent. Claude Code runs locally in the terminal, scoped to a project directory. It reads an AGENTS.md file at the repository root for project context and rules.

The original plan described a generic "agent CLI wrapper" and "agent_runner." These are unnecessary for Claude Code, which already operates in the terminal and logs its own sessions. The constraints that matter (entrypoint, protected fields, allowed directories) are enforced by the entrypoint script and pre-commit hooks, not by a wrapper.

Future-proofing for Codex or similar remote agents: The architecture is deliberately agent-agnostic. The system's contract is: all work happens on feature branches, all experiments go through run_experiment.py, all PRs need an MLflow run ID. Whether the agent is local

(Claude Code) or remote (Codex) doesn't matter; the workflow is identical. The key design principle: AGENTS.md is a courtesy briefing that tells agents the rules, while the entrypoint and hooks are the actual enforcement. A remote agent that never reads AGENTS.md still cannot break the rules.

What would need to change for Codex: Nothing structural. Codex works on a sandboxed branch and submits PRs. Your existing PR review process and entrypoint checks apply directly. You would add Codex-specific notes to AGENTS.md (e.g., branch naming conventions) but no architectural changes are needed.

3.2 Project Templating: Copier

Each client project is a standalone repository generated from a template using **Copier** (copier.readthedocs.io), an open-source project templating tool. When you start a new client project, you run a single command and Copier generates a complete project with the scaffold baked in.

Why Copier over alternatives: Copier has a built-in `copier update` command that propagates template changes to existing projects. This directly mitigates the main risk of standalone project repos: drift.

Drift risk: The realistic bad case: after several months, you improve the template but existing client projects still use the old scaffold. If you've made client-specific edits to scaffold files, updating requires a manual merge. Mitigation: never edit scaffold files per client. Keep all client-specific logic in src/. If scaffold files stay untouched, Copier updates are seamless.

3.3 Environment Management: uv + pyproject.toml

Each project uses **uv** as the Python package manager, with a pyproject.toml for dependencies and a uv.lock for pinned versions. uv is fast, handles lockfiles natively, and is becoming the standard Python tool for environment management.

Dependency lock hash: The SHA256 hash of uv.lock is computed and logged as a Tier 1 MLflow parameter on every experiment run. This captures the full transitive dependency state without requiring Docker, enabling exact reproduction of the package environment for any historical run.

No Docker for the Minimal or Core phases. A Dockerfile is added in the Full phase as a reproducibility artifact for onboarding and auditing, not as the daily workflow.

The case for Docker later: (a) Onboarding speed: a new team member runs docker compose up instead of debugging OS-specific issues. (b) Exact reproducibility: eliminates 'works on my machine' problems, important when auditors ask you to reproduce a result from 18 months ago. (c) Agent environment consistency: a Dockerfile is the cleanest way to specify a reproducible environment for remote agents.

3.4 MLflow: SQLite Backend

MLflow runs on a local SQLite backend with local artifact storage. This is sufficient for a small team and avoids the operational overhead of running a database server.

Migration trigger: Migrate to Postgres + cloud artifact storage when multiple people need simultaneous remote access to the tracking server. Until then, SQLite is fine.

Backup: The MLflow SQLite database and artifact directory are excluded from git (binary files that change with every experiment would bloat the repo). They are backed up weekly to an external drive or cloud storage folder. Added in the Full phase.

3.5 Data Governance: Content-Addressed Storage

Amendment from expert review. Datasets are stored under directories named by their SHA256 content hash. This replaces mutable version strings (e.g., shadow_v3) with a structural guarantee of immutability. If the data changes, the hash changes, and a new directory is created. The old data is never modified.

The `dataset_version` field in experiment.yaml must equal the dataset's content hash. The data access layer (`data_access.py`) verifies the hash at load time, rejecting any mismatch. This prevents silent data drift and strengthens audit readiness without adding heavyweight tools like DVC.

3.6 Git Workflow

The main branch is protected. All work, whether by a human or by Claude Code, happens on feature branches. Pull requests must include an MLflow run ID in the description.

Why require an MLflow run ID: Traceability. Every code change that affects modeling must link to an experiment proving what happened. The run ID lets you trace from PR to experiment to metrics, dataset version, and feature hash. In regulated finance, this is your audit trail.

PR ↔ MLflow cryptographic verification: A pre-merge hook (scripts/verify_pr_run_match.py) automatically verifies that the MLflow run's commit hash matches the branch's head commit before merging. This is run locally as a git pre-merge hook, making it mechanical with zero attention cost. Cloud CI can replace this later when MLflow moves to a server.

No CI/CD pipeline initially. Pre-commit hooks handle config validation, protected field checks, and training guard enforcement. The pre-merge hook handles PR-to-run verification.

3.7 Centralized Project Standards

Amendment from expert review. A single file, `configs/project_standards.yaml`, serves as the source of truth for all governance rules: protected field reference values, allowed metrics, allowed split strategies, allowed model types, and the hypothesis category taxonomy. This replaces the earlier protected_defaults.yaml and avoids hard-coding evolving rules in Python. The Pydantic schema reads this file and validates experiment configs against it.

4. Repository Structure

There are two repositories: the platform template repository and the client project repositories generated from it.

4.1 Platform Template Repository (ml-platform/)

This is the Copier template. It contains the scaffold that all client projects inherit.

```
ml-platform/
├── copier.yaml                  # Copier template config
├── template/
│   ├── AGENTS.md                 # Agent instructions + risk checklist
│   ├── run_experiment.py          # Single experiment entrypoint
│   ├── pyproject.toml             # Dependencies template
│   ├── .pre-commit-config.yaml    # Pre-commit hook definitions
│   ├── .githooks/
│   │   └── pre-merge-commit       # Verifies PR-to-MLflow run match
│   ├── configs/
│   │   ├── experiment.yaml        # Experiment config template
│   │   ├── reasoning.yaml         # Reasoning metadata template
│   │   └── project_standards.yaml # Protected fields, allowed values, taxonomy
│   ├── src/
│   │   ├── data_access.py         # Governed data access layer (CAS-aware)
│   │   ├── features.py            # Feature engineering module
│   │   └── model.py               # Model training module
│   ├── scripts/
│   │   ├── verify_pr_run_match.py # Cryptographic PR/run verification
│   │   ├── promote_model.py       # Model promotion (Full phase)
│   │   ├── query_reasoning.py     # Cross-project meta-analysis
│   │   ├── cost_report.py         # Cost aggregation
│   │   └── backup_mlflow.sh       # Backup automation
│   ├── data/
│   │   └── shadow/                # Content-addressed dataset directories
│   │       └── {sha256_hash}/     # Each dataset in its hash-named directory
│   ├── tests/
│   │   ├── test_config.py         # Config schema validation tests
│   │   ├── test_entrypoint.py     # Entrypoint smoke test
│   │   ├── test_data_boundary.py  # Data governance + CAS tests
│   │   ├── test_protected.py      # Protected field tests
│   │   └── test_features.py       # Feature hash determinism tests
│   ├── logs/
│   │   └── protected_overrides.log  # Append-only override audit log
│   ├── skills/                    # Reusable agent task instructions (populated
later)
│   └── .gitignore
```

4.2 Client Project Repository (generated)

Generated via `copier copy ml-platform/ client-project/`. The structure is identical to the template with placeholder values filled in. Client-specific code goes in src/. Scaffold files (run_experiment.py, data_access.py, test files) should not be edited per client.

5. Phase 1: Minimal (1–2 Days)

Goal: A single client project where you can run an experiment via the standard entrypoint and see it logged in MLflow. Claude Code can operate in this project with basic instructions and a modeling risk checklist.

Checkpoint: Run `python run_experiment.py --config configs/experiment.yaml`, then open the MLflow UI and see the run with all Tier 1 fields logged, including dependency lock hash.

5.1 Step 1: Set Up the Project Directory

Create a single client project directory manually (no Copier yet; that comes in the Full phase). This is your working prototype.

```
mkdir -p client-project/{configs,src,data/shadow,tests,logs,scripts,.githooks}
cd client-project
uv init
```

Edit pyproject.toml to add core dependencies: `mlflow`, `pydantic`, `pyyaml`, `pytest`, and your modeling libraries (e.g., `xgboost`, `scikit-learn`). Then run `uv lock` and `uv sync` to install.

5.2 Step 2: Create the Configuration Schema

Create two YAML files. The experiment config defines what to run. The reasoning metadata defines why.

configs/experiment.yaml:

```
project_name: fraud_model_v1
client_name: bank_x
dataset_version: a1b2c3d4e5...   # SHA256 hash of dataset contents
feature_pipeline_version: auto    # Overwritten by computed hash
model_type: xgboost
random_seed: 42
hyperparameters:
  max_depth: 6
  learning_rate: 0.05
evaluation:
  metric: roc_auc
  split_strategy: stratified_kfold
protected:
  metric_definition: roc_auc
  split_strategy: stratified_kfold
```

configs/reasoning.yaml:

```
hypothesis_category: regularization
change_description: "Increased depth + reduced learning rate"
expected_effect: reduce_variance    # Required for agents, optional for humans
outcome: null                       # Filled in after experiment
metric_delta: null                  # Filled in automatically
```

Create a Pydantic model in `src/config_schema.py` that validates experiment.yaml on load. The schema should enforce types, required fields, and allowed values. In the Core phase, this schema will read from `configs/project_standards.yaml` for the list of allowed values; for now, hard-code them. Validation errors must produce clear messages so agents can self-correct.

5.3 Step 3: Build run_experiment.py

This is the core of the system. All experiments must go through this script. It does the following, in order:

1. **Load and validate the experiment config** against the Pydantic schema. Exit with a clear error if validation fails.
2. **Load reasoning metadata** from configs/reasoning.yaml.
3. **Capture environment metadata:** Git commit hash and dirty flag, Python version, key package versions.
4. **Compute dependency lock hash:** SHA256 of uv.lock. This captures the full transitive dependency state.
5. **Create an MLflow run** tagged with client_name and project_name.
6. **Log all Tier 1 parameters:** commit hash, dataset version (content hash), dependency lock hash, feature pipeline hash (placeholder for now), hyperparameters, random seed.
7. **Log reasoning metadata** as MLflow parameters.
8. **Execute the model training** by calling a function from src/model.py. The training function receives the config and returns metrics.
9. **Log metrics** (e.g., roc_auc, accuracy) to MLflow.
10. **Log artifacts** (model file, any plots) to MLflow.
11. **Close the MLflow run cleanly,** printing the run ID to stdout.

Critical rule: No modeling code may bypass this entrypoint. This is stated in AGENTS.md and enforced socially (via PR review) in the Minimal phase. Mechanical enforcement via pre-commit hook comes in the Core phase.

5.4 Step 4: Set Up MLflow

Initialize a local MLflow tracking server with SQLite backend. Set the `MLFLOW_TRACKING_URI` environment variable to a local path (e.g., `sqlite:///mlflow.db`). MLflow will create the database automatically on first run.

Add `mlflow.db` and `mlruns/` to .gitignore. These are binary/large files that don't belong in git.

Verify by running `mlflow ui` after your first experiment and confirming all Tier 1 fields are visible.

Tier 1 fields (mandatory per run):

- Git commit hash + dirty flag
- Dataset version (content hash)
- Dataset schema hash (column names, types)
- Dependency lock hash (SHA256 of uv.lock)
- Feature pipeline hash (placeholder in this phase)
- All hyperparameters
- Random seed
- All evaluation metrics

Tier 2 fields (recommended, add when convenient):

- CPU/GPU type, RAM
- Training time, inference latency
- Dataset size (rows, columns)

5.5 Step 5: Write AGENTS.md

Place this file at the repository root. Claude Code (and future agents) will read it automatically. The file should contain:

- **Project overview:** What this project does, who the client is, what the model predicts.
- **Entrypoint rule:** "All experiments must be run via python run_experiment.py --config configs/experiment.yaml. No ad hoc training scripts."
- **Config rules:** "Modify configs/experiment.yaml for hyperparameter changes. Fill in configs/reasoning.yaml with your hypothesis before each run."
- **Protected fields:** "The following fields in the 'protected' block of experiment.yaml must not be changed without human approval: metric_definition, split_strategy."
- **Data rules:** "Only access data through src/data_access.py. Only use data in data/shadow/. Never access or reference production data. Dataset directories are named by SHA256 hash; use the hash from experiment.yaml."
- **File scope:** "You may modify files in src/ and configs/. Do not modify run_experiment.py, test files, or data_access.py without explicit approval."
- **Git rules:** "Work on feature branches. Include the MLflow run ID in your commit messages and PR descriptions."
- **Testing:** "Run pytest tests/ after any change and before committing. All tests must pass."

Modeling Risk Checklist (amendment from expert review):

AGENTS.md must include a mandatory pre-run checklist that agents verify before each experiment:

1. **Missing data:** Have missing values been handled appropriately? Are there unexplained drops in feature coverage?
2. **Data leakage:** Does the feature pipeline use any information that would not be available at prediction time? Are there target-correlated features that should be excluded?
3. **Temporal validation:** If the data is time-series, is the train/test split temporal? Is the validation strategy appropriate for the data's time structure?
4. **Class imbalance:** Is the target variable imbalanced? Has this been accounted for in the model or evaluation metric?
5. **Protected fields:** Are the protected fields (metric_definition, split_strategy) unchanged from project_standards.yaml?

This checklist costs agents nothing to verify and catches the most common modeling errors in financial ML, especially under team churn where institutional knowledge about data quirks may be lost.

5.6 Step 6: Write Basic Tests

Tests are mission-critical. They serve two purposes: verifying the system works, and giving agents a feedback loop (an agent that can run pytest can self-correct).

tests/test_config.py:

- Test that a valid config loads without errors.
- Test that a config with a missing required field raises a clear validation error.
- Test that a config with an invalid value (e.g., model_type: "invalid") is rejected.

tests/test_entrypoint.py:

- Smoke test: run run_experiment.py with a toy dataset and config, verify it completes and creates an MLflow run.
- Verify all Tier 1 fields are present in the MLflow run, including dependency lock hash.

Run tests with `pytest tests/`. Include this command in AGENTS.md.

Checkpoint: At this point, you should be able to (1) run an experiment end-to-end, (2) see all Tier 1 fields (including dependency lock hash) in MLflow, (3) have Claude Code read AGENTS.md, modify a hyperparameter, run an experiment, and see the result logged. Verify all three before moving to Phase 2.

6. Phase 2: Core (~1 Week)

Goal: Full governance enforcement. An agent operates within all constraints: content-addressed data, protected fields, feature tracking, training guard, and cryptographic PR verification.

Checkpoint: Claude Code attempts to (a) modify a protected field — blocked, (b) access data outside data_access.py — test fails, (c) run an experiment with changed features — hash changes automatically in MLflow, (d) train a model outside run_experiment.py — commit blocked by pre-commit hook.

6.1 Step 7: Implement Content-Addressed Data Access

Amendment from expert review. This replaces the simpler data governance boundary from the original plan with content-addressed storage (CAS).

How it works: Datasets are stored in `data/shadow/{sha256_hash}/`. Each directory contains the dataset files and is named by the SHA256 hash of its contents. The `dataset_version` field in experiment.yaml must equal this hash.

src/data_access.py provides a single interface (e.g., `load_shadow_dataset(content_hash)`) and enforces:

- Only files in data/shadow/ can be loaded.
- The directory name must match the content_hash parameter.
- At load time, the actual SHA256 of the dataset files is recomputed and verified against the directory name. If they don't match, the load fails with a clear error.
- The dataset schema hash (column names, types) is computed and returned alongside the data.
- Both hashes (content hash and schema hash) are logged to MLflow by run_experiment.py as Tier 1 parameters.

Production data (if it exists locally) lives in a separate directory excluded from agent scope. AGENTS.md states the rule; the data access layer enforces it by only looking in data/shadow/.

Adding a new dataset: Compute the SHA256 of the dataset files, create a directory with that hash as the name, and place the files inside. Update `dataset_version` in experiment.yaml to the new hash. The old dataset directory remains untouched.

6.2 Step 8: Implement Feature Pipeline Hashing

What this is: An automatic change-detection mechanism for feature engineering code. The system computes SHA256 over the contents of src/features.py plus the feature-related section of the experiment config. This hash is logged to MLflow on every run.

Why it matters: Without this, you can change feature logic, run the "same" experiment (same config, same hyperparameters), and get different results with no way to tell why. The hash makes invisible changes visible. If the code or config changed, the hash changes, and MLflow shows it. This is especially important in financial ML where feature engineering drives most performance gains, and in a high-churn team where the next person may not know what changed.

Implementation: In run_experiment.py, before training, compute: `SHA256(contents of src/features.py + feature config as sorted JSON string)`. Log the result as `feature_pipeline_hash` in MLflow. No AST parsing, no complex machinery.

6.3 Step 9: Create project_standards.yaml and Enforce Protected Fields

Amendment from expert review. This step combines the original protected field enforcement with the centralized project standards configuration.

configs/project_standards.yaml is the single source of truth for governance rules:

```
protected_fields:
  metric_definition: roc_auc
  split_strategy: stratified_kfold
allowed_metrics: [roc_auc, f1, precision, recall, log_loss]
allowed_split_strategies: [stratified_kfold, temporal_split]
allowed_model_types: [xgboost, lightgbm, logistic_regression, random_forest]
hypothesis_categories:
  - feature_addition
  - feature_transformation
  - feature_removal
  - hyperparameter_tuning
  - regularization
  - model_architecture
  - data_cleaning
  - data_expansion
  - threshold_tuning
  - ensemble
  - pipeline_refactor
```

The Pydantic schema in config_schema.py reads this file and validates experiment configs against it. This avoids hard-coding evolving rules in Python.

Enforcement (two points):

A. Entrypoint check: run_experiment.py loads project_standards.yaml and compares each protected field in the experiment config against it. If any differ and the `--override-protected`

flag was not passed, the script exits with a clear error listing the mismatched fields. If `--override-protected` is passed, the override is allowed but an entry is appended to `logs/protected_overrides.log` with: timestamp, field name, old value, new value, operator (from git config or environment variable), and the experiment config file path.

B. Pre-commit hook: A hook runs on every commit and checks whether any protected field in any configs/experiment*.yaml file differs from project_standards.yaml. If so, the commit is blocked with a message explaining which fields changed and how to proceed.

The append-only override log: logs/protected_overrides.log is committed to git, giving it its own version history. This provides an audit trail of every protected field change without adding workflow burden.

6.4 Step 10: Add Mechanical Training Guard

Amendment from expert review. A pre-commit hook scans Python files for patterns that indicate training or MLflow usage outside the approved entrypoint: `mlflow.start_run`, `mlflow.log_`, `.fit(`, `.train(`, `.partial_fit(`.

Exclusion paths (files and directories where these patterns are legitimate and should not trigger the hook):

- `run_experiment.py` — the approved entrypoint, which legitimately uses all these patterns.
- `tests/` — test files need to call .fit() on toy models and may reference MLflow in integration tests.
- `scripts/` — utility scripts like promote_model.py may legitimately interact with MLflow.
- `*.md` files — documentation will mention these patterns in prose.

In practice, the hook scans `src/` and any other .py files not in the exclusion list. This is where an agent or human would most likely put ad hoc training code. The guard is not bulletproof (patterns in comments or strings will trigger false positives), but it raises the cost of accidental bypass significantly.

6.5 Step 11: Set Up Git Workflow and Pre-Merge Verification

Initialize the project as a git repository (if not already). Set up the following:

- **Branch protection:** The main branch requires pull requests. Direct pushes are blocked.
- **Pre-commit hooks:** Install the pre-commit framework. Add hooks for: (a) protected field check, (b) config schema validation, (c) training guard (Step 10).
- **Pre-merge hook:** Install scripts/verify_pr_run_match.py as a local git pre-merge hook (.githooks/pre-merge-commit). This script extracts the MLflow run ID from the merge

commit message or branch metadata, queries MLflow for the run's commit hash, and verifies it matches the branch head. If it doesn't match, the merge is blocked. Configure git to use the .githooks directory: `git config core.hooksPath .githooks`

- **PR template:** Add a pull request template that includes a field for the MLflow run ID.
- **Agent workflow:** Claude Code creates feature branches, makes changes, runs experiments, and commits. You review and merge. Include this workflow in AGENTS.md.

6.6 Step 12: Expand the Test Suite

Add tests for the new enforcement mechanisms:

tests/test_data_boundary.py:

- Test that data_access.py successfully loads a valid shadow dataset by content hash.
- Test that attempting to load a file outside data/shadow/ raises an error.
- Test that a dataset whose contents don't match the directory name hash is rejected.
- Test that dataset hash and schema hash are returned and are deterministic.

tests/test_protected.py:

- Test that run_experiment.py refuses to run when a protected field is modified (without --override-protected).
- Test that run_experiment.py runs successfully when --override-protected is passed.
- Test that the override log is appended to correctly.
- Test that config values are validated against project_standards.yaml (e.g., an invalid model_type is rejected).

tests/test_features.py:

- Test that the feature hash is deterministic (same input, same hash).
- Test that modifying src/features.py changes the hash.
- Test that modifying the feature config section changes the hash.

Checkpoint: All tests pass. Claude Code can run a full experiment cycle within all constraints. Manually verify: (1) modify a protected field and confirm the entrypoint blocks it, (2) change feature code and confirm the hash changes in MLflow, (3) attempt to access data outside the shadow directory and confirm it fails, (4) attempt to commit code with .fit() in src/ and confirm the pre-commit hook blocks it, (5) attempt to merge with a mismatched run ID and confirm the pre-merge hook blocks it.

7. Phase 3: Full (Following Weeks)

Goal: The system is templatized, fully instrumented, and ready for multiple client projects with cross-project learning. Includes model promotion, model cards, and automated backup.

Checkpoint: Create a new client project from the Copier template, run an experiment, promote the model, and confirm all features work out of the box.

7.1 Step 13: Templatize with Copier

Extract the working client project into a Copier template in the ml-platform/ repository.

- Create copier.yaml with prompts for: project_name, client_name, initial model_type, initial metric.
- Replace hardcoded values in config files and AGENTS.md with Copier template variables (e.g., `{{ project_name }}`).
- Test by generating a new project: `copier copy ml-platform/ test-project/` and verifying it runs end-to-end.
- Document the update workflow: `copier update` in existing projects to pull template improvements.

7.2 Step 14: Add Reasoning Logging and Queries

Reasoning metadata is already captured in configs/reasoning.yaml (from Phase 1). In this step, make it queryable and useful.

Hypothesis categories are defined in project_standards.yaml (see Section 6.3). The Pydantic schema validates that reasoning.yaml uses one of the allowed categories.

Outcome classification: `confirmed`, `neutral`, `negative`. Filled in after the experiment, either manually or automatically by comparing the primary metric to the previous best run for the same project.

Metric delta: Computed automatically by run_experiment.py: query MLflow for the best prior run in the same project, compute the difference in the primary metric, and log it. This enables the two key queries:

- *"For project X, which experiment changes led to the largest metric improvements?"*
- *"Across all projects, which hypothesis categories most often led to confirmed improvements?"*

Create `scripts/query_reasoning.py` that runs these queries against MLflow and outputs a summary table.

7.3 Step 15: Add Cost Tracking

Track agent token usage and cost per experiment. Claude Code outputs token usage in its session logs. The run_experiment.py entrypoint accepts optional cost parameters (tokens_used, estimated_cost) that agents fill in.

- Log cost fields as Tier 3 MLflow parameters.
- Create `scripts/cost_report.py` that aggregates costs per project and per week.
- Per-task budget ceilings can be added as a follow-up.

7.4 Step 16: Model Promotion and Model Cards

Amendment from expert review. This establishes a governance boundary between research experiments and approved models.

scripts/promote_model.py performs the following steps:

1. **Verify prerequisites:** Git commit is clean (no dirty flag), dataset hash is verified (CAS integrity), all protected fields match project_standards.yaml.
2. **Tag the MLflow run:** Set `promoted=true` and `promotion_timestamp` on the MLflow run. This marks it as an approved model without creating a separate artifact store.
3. **Log model card fields** as MLflow parameters on the promoted run: intended use, dataset content hash, commit hash, dependency lock hash, all evaluation metrics, and approval timestamp.

When a deployment pipeline is added later, it queries MLflow for runs tagged promoted=true. The model card fields are already attached to the run and can be exported as needed.

7.5 Step 17: Add Dockerfile

Create a Dockerfile at the project template root that reproduces the development environment. Purpose: onboarding and audit reproducibility, not daily workflow.

- Base image: `python:3.11-slim` (or your standardized version).
- Install uv, copy pyproject.toml and uv.lock, run `uv sync`.
- Copy project files. Expose MLflow UI port.
- Test: `docker build -t ml-platform . && docker run ml-platform pytest tests/`.

7.6 Step 18: Set Up Backup

Create `scripts/backup_mlflow.sh` that copies the MLflow SQLite database and the artifact directory (mlruns/) to a designated backup location (external drive or cloud storage folder). The script creates timestamped backup directories to maintain history.

Schedule weekly, either via cron or as a manual Friday task. Perform one restore test when setting up the backup to verify it works.

Key files to back up: the SQLite database file and the mlruns/ artifact directory. Everything else is in git.

7.7 Step 19: Populate Skills Directory

The skills/ directory holds reusable instructions for specific types of repeated agent tasks. Each skill is a markdown file that an agent can be pointed to for context. Examples:

- `skills/new_feature.md` — How to add a new feature to the pipeline.
- `skills/hyperparameter_search.md` — How to conduct a structured hyperparameter search.
- `skills/model_comparison.md` — How to compare model types.

These are populated over time as patterns emerge. Start with blank stubs and fill them in as you actually perform each task type.

7.8 Step 20: Cross-Project Query Scripts

Build scripts that query MLflow across all client projects to answer institutional questions:

- *"What modeling approaches have worked best for fraud detection across clients?"*
- *"Which feature categories drive the most gain?"*
- *"What is the average number of experiments needed to reach a target metric?"*

These scripts read from the shared MLflow database and filter by client_name and project_name tags. Output: summary tables and simple plots.

8. Configuration Reference

8.1 experiment.yaml — Full Schema

Field	Type	Required	Notes
project_name	string	Yes	Unique project identifier
client_name	string	Yes	Client identifier, used for MLflow tagging

dataset_version	string (hash)	Yes	SHA256 content hash of the dataset
feature_pipeline_version	string	Auto	Overwritten by computed hash at runtime
model_type	enum	Yes	Validated against project_standards.yaml
random_seed	integer	Yes	For reproducibility
hyperparameters	dict	Yes	Model-specific; validated per model_type
evaluation.metric	enum	Yes	Validated against project_standards.yaml
evaluation.split_strategy	enum	Yes	Validated against project_standards.yaml
protected.metric_definition	enum	Yes	Must match project_standards.yaml
protected.split_strategy	enum	Yes	Must match project_standards.yaml

8.2 reasoning.yaml — Full Schema

Field	Type	Required	Notes
hypothesis_category	enum	Yes	Validated against project_standards.yaml
change_description	string	Yes	Free text: what was changed and why
expected_effect	enum	Agents only	improve, reduce_variance, explore
outcome	enum	Post-run	confirmed, neutral, negative
metric_delta	float	Auto	Computed by comparing to best prior run

8.3 project_standards.yaml — Full Schema

Field	Type	Notes
protected_fields	dict	Reference values for protected config fields
allowed_metrics	list[string]	Valid evaluation metrics
allowed_split_strategies	list[string]	Valid split strategies
allowed_model_types	list[string]	Valid model types
hypothesis_categories	list[string]	Valid hypothesis categories for reasoning.yaml

9. MLflow Logging Tiers

Tier	Fields	When
Tier 1 (mandatory)	Commit hash, dirty flag, dataset content hash, dataset schema hash, dependency lock hash, feature pipeline hash, all hyperparameters, random seed, all metrics	Every run, from Phase 1
Tier 2 (recommended)	CPU/GPU type, RAM, training time, inference latency, dataset size (rows/columns)	Add as convenient
Tier 3 (telemetry)	Agent tokens used, agent cost estimate, agent model used, agent duration, reasoning fields, model card fields (on promoted runs)	From Phase 3

All client projects log to the same MLflow instance, distinguished by client_name and project_name tags.

10. Governance Rules

These rules apply to all contributors, human and agent. They are stated in AGENTS.md and enforced mechanically where noted.

1. **No experiment without an MLflow run ID.** All modeling work must produce a logged run.

2. **No training outside run_experiment.py.** Enforced by pre-commit hook scanning for training patterns outside approved files.
3. **No direct production data access.** Agents may only use shadow data through data_access.py. Enforced by the CAS-aware data access layer.
4. **All PRs must reference an MLflow run ID.** Verified mechanically by pre-merge hook matching run commit to branch head.
5. **Protected fields cannot be changed without explicit override.** Enforced by entrypoint check and pre-commit hook. Overrides logged to append-only audit log.
6. **All config changes validated against project_standards.yaml.** Enforced by Pydantic schema at runtime and pre-commit hook at commit time.
7. **Agents must fill in reasoning metadata.** hypothesis_category, change_description, and expected_effect are required for agent-initiated experiments.
8. **Datasets are immutable.** Content-addressed storage ensures data cannot be silently modified. Hash verified at load time.
9. **Model promotion requires clean state.** promote_model.py verifies commit cleanliness, dataset integrity, and protected field compliance before tagging a run as promoted.

11. Testing Strategy

Tests are organized by phase. All tests run via pytest. Agents are instructed to run pytest tests/ after any change and before committing.

Phase	Test File	What It Tests
Minimal	test_config.py	Valid config loads; invalid configs rejected with clear errors
Minimal	test_entrypoint.py	Smoke test: experiment runs end-to-end; all Tier 1 fields logged including dependency lock hash
Core	test_data_boundary.py	CAS data loads correctly; out-of-scope access blocked; hash mismatch rejected; hashes deterministic
Core	test_protected.py	Protected field modification blocked; override flag works; log appended; values validated against project_standards.yaml
Core	test_features.py	Feature hash deterministic; changes to code or config change hash
Full	test_reasoning.py	Reasoning schema validated against project_standards.yaml; metric_delta computed correctly

12. Strategic Rationale

This system creates a controlled AI-augmented research lab. The core bet is that structured logging and mechanical governance, combined with agent capabilities, compound into an institutional asset over 2–3 years. Specifically:

- **Reproducibility under churn:** When team members rotate, the system preserves all experimental knowledge. A new person can query MLflow to understand what was tried, what worked, and why. Content-addressed data and dependency lock hashes ensure any historical result can be exactly reproduced.
- **Agent leverage without risk:** Agents accelerate experimentation but operate within strict, mechanically-enforced boundaries. Protected fields, data governance, the training guard, and the single entrypoint prevent agents from producing untraceable or non-compliant results.
- **Cross-project learning:** Centralized MLflow logging with structured reasoning fields enables meta-analysis across clients. Over time, this reveals which modeling strategies work in which contexts — knowledge that would otherwise be lost to team churn.
- **Regulatory readiness:** Every model run is traceable from PR (cryptographically verified) to experiment to data version (content-addressed) to feature pipeline (hashed) to dependency state (lock hash). Model promotion creates a clear governance boundary with auto-generated model card fields. This audit trail is designed for banking and financial consulting.
- **Simplicity as strategy:** The system uses standard, well-supported tools (Python, uv, MLflow, Copier, pytest, git). No proprietary frameworks, no heavy infrastructure, no Docker in daily workflow. Mechanical enforcement (hooks, entrypoint checks) replaces reliance on human discipline. This makes the system maintainable by a solo operator and easy to explain to new team members and auditors.

13. Amendment Log

The following amendments from expert review have been incorporated into this version:

Amendment	Section	Summary
Content-Addressed Storage	6.1	Replace mutable dataset versioning with SHA256 hash-named directories, verified at load time
Model Promotion Boundary	7.4	promote_model.py tags MLflow runs as promoted with model card fields, after verification checks

Model Cards as MLflow Parameters	7.4	Auto-generated model card fields logged to MLflow on promoted runs (no separate file)
PR ↔ Run Cryptographic Coupling	6.5	Pre-merge hook verifies MLflow run commit hash matches branch head commit
Dependency Lock Hash	5.3	SHA256 of uv.lock logged as Tier 1 parameter on every run
Centralized Project Standards	6.3	project_standards.yaml replaces protected_defaults.yaml; single source for all governance rules
Mechanical Training Guard	6.4	Pre-commit hook blocks training patterns outside approved files (with exclusions for tests/, scripts/, run_experiment.py, *.md)
Modeling Risk Checklist	5.5	Mandatory agent pre-run checklist in AGENTS.md covering leakage, temporal validation, imbalance, etc.
Backup Hardening	7.6	Timestamped backup directories; one-time restore test on setup