Engineering Manifest – Guidelines for Coding in the Strategy Framework

0. Purpose & Scope

This document defines how we design and implement code in the strategy framework (strategies, pipelines, UI integration, tests).

It is aimed at:

- **Developers**: day-to-day coding decisions.
- **Architects**: keeping the system extensible, robust, and clean.
- **Reviewers**: consistent criteria for code review and “Definition of Done”.

Language conventions:
- **MUST** = non-negotiable.
- **SHOULD** = recommended, devs need a strong reason to deviate.
- **MAY** = optional / context-dependent.

---

1. Strategy Metadata & Capabilities

1.1 Strategy metadata is the single source of truth

- Every strategy MUST be described by a **strongly-typed `StrategyMetadata` dataclass** (or equivalent).
- `StrategyMetadata` MUST contain:
  - **Explicit capability flags**, e.g.:
    - `requires_universe: bool`
    - `supports_two_stage_pipeline: bool`
  - **Config defaults**, e.g.:
    - `default_strategy_config: dict[str, Any]` (sane defaults for required parameters).

Why: avoids hidden behavior, enables generic orchestration, and makes misconfiguration easy to detect in tests.

1.2 Capabilities describe *what*, not *who*

- Capability flags MUST describe **what a strategy needs or can do**, not which strategy it is.
  - ■ `requires_universe=True`
  - ■ `is_rudometkin=True` (strategy-name flags are forbidden).
- New capabilities MUST be documented by:
  - A new field on `StrategyMetadata`.
  - A short inline docstring or comment explaining what the capability means.

---

2. Robustness & Error Handling

2.1 Fail fast, in the right layer

For operations like **universe loading, date ranges, symbol selection**:

- The responsible component (e.g. universe loader, validator) MUST:
  - Validate **inputs and invariants early**.
  - Return a clear status (OK / warning / error) and message.
  - **Abort gracefully** before deep pipeline execution if a fatal error occurs.

Example invariants:

- Universe parquet:
  - File exists.
  - Contains required columns (e.g. `symbol`, `date`).

  - Is not empty when a universe is required.

2.2 Narrow exceptions with context

- Broad `except Exception:` MUST be avoided in new code and SHOULD be gradually replaced in existing code.
- When catching, prefer **specific exception types**, e.g.:
  - `except (OSError, ValueError, pd.errors.EmptyDataError) as exc:`
- Error messages MUST include **key context**:
  - File path, strategy name, shape of data, parameter values relevant to debugging.

Rule of thumb: If someone sees the log line in isolation, they should know *what failed and where*.

2.3 Don’t silently swallow programming errors

- When constructing strategies (via factories) or pipelines:
  - User-facing errors (invalid config values, missing required parameter) MAY be converted into friendly messages.
  - **Internal programming errors** (wrong parameter name, failed import, attribute errors) MUST:
    - Be logged with high detail.
    - Be re-raised as a more specific exception or allowed to propagate.
- Optional: Provide a “developer details” section in error logs that exposes tracebacks/config for debugging.

Guideline: We never hide bugs by turning them into “user” errors.

---

3. Modularity & Separation of Concerns

3.1 Strategy capabilities, not strategy names, drive logic

- Core pipeline execution (e.g. `execute_pipeline`) MUST NOT branch on concrete strategy names:
  - ■ `if strategy_name == "rudometkin_moc": ...`
- Instead, it MUST branch on **capabilities**, for example:
  ```python
  if strategy.supports_two_stage_pipeline:
      daily_scan = strategy_hooks.get_daily_scan(strategy)
      filtered_symbols = daily_scan(...)
  ```
- Any remaining name-based checks MUST be treated as technical debt and removed.

3.2 Dedicated pipeline modules per complex strategy

- Strategies with non-trivial behavior (e.g. universe, daily scan, two-stage pipeline) MUST have a **dedicated pipeline module**, e.g.:
  - `strategies.<strategy_name>.pipeline`
- This module SHOULD encapsulate:
  - Universe loading & validation.
  - Daily scan / candidate selection.
  - Strategy-specific pipeline steps.
  - Optional integration flags (e.g. `HAS_STREAMLIT`) to avoid hard dependencies.

Goal: UI and generic pipeline code must not own strategy-specific business logic.

3.3 Clear UI vs orchestration boundaries

- The UI layer (e.g. Streamlit app) MUST:
  - Own **user interaction & layout**.
  - Map UI inputs to **plain Python structures** (`dict`, `StrategyConfig`).
  - Delegate all non-UI work to dedicated modules.

- The UI layer SHOULD NOT:
  - Reach deeply into strategy internals.
  - Hardcode per-strategy config shape everywhere.

Pattern:

- For each strategy, provide small adapter helpers, e.g.:
  - `build_config_for(strategy_name, ui_values) -> dict`
- The UI then only:
  - Collects `ui_values`.
  - Calls `build_config_for(...)`.
  - Displays a config preview.

---

4. Configuration, Validation & UX Transparency

4.1 Validate inputs before running pipelines

- Before any pipeline execution, the system MUST validate:
  - Symbols: non-empty set after combining manual entries and universe.
  - Dates: start/end dates, ordering, allowed ranges.
  - Strategy-specific required params.

- Invalid inputs MUST:
  - Be surfaced clearly in the UI (or CLI).
  - Block execution until fixed.

4.2 Make the effective config visible

- For any non-trivial pipeline, there MUST be a way to **preview the effective config**, e.g.:
  - UI “Preview config payload” expander.
  - CLI `--dry-run` / `--print-config` flag.
- This config preview MUST reflect:
  - Strategy, capabilities.
  - Universe path, symbols, dates.
  - Risk parameters and key thresholds.

Reason: This is invaluable both for users and maintainers to understand what is really being run.

---

5. Testing & CI

5.1 Single, canonical test entrypoint

- CI MUST call tests using the **same command the team uses locally**, e.g.:
  ```bash
  PYTHONPATH=src python -m pytest -q tests
  ```
- The repository MUST be structured so this command:
  - Picks up only tests, not random scripts.
  - Runs all integration/unit tests deterministically.

5.2 What to test

New features / modules MUST be accompanied by tests that cover:

1. **Happy path** behavior:
   - Correct pipeline execution with valid inputs.
2. **Validation and error paths**:
   - Missing universe.

   - Malformed parquet (missing columns).
   - Empty universe.
   - Invalid dates.
   - Invalid symbols.
3. **Architecture constraints**:
   - Capabilities correctness (strategy metadata).
   - Separation tests ensuring strategies don’t leak into each other’s metadata or methods.
4. **Export / integration contracts**:
   - Schema of exported orders.
   - Name and presence of required columns for downstream systems (IB, etc).

5.3 Architecture & separation tests

There MUST be dedicated tests asserting **architectural intent**, e.g.:

- Strategy A and B do not share universe capabilities unless explicitly intended.
- Each universe-based strategy has:
  - `requires_universe=True`.
  - A pipeline module with `run_daily_scan` defined.
- Optional dependencies (e.g., Streamlit) do not break core imports in non-UI environments.

Architecture tests are the safety net that keeps the codebase clean as it evolves.

5.4 Coverage expectations

- The project SHOULD maintain **high logical coverage**, especially on:
  - Validation logic.
  - Strategy/pipeline orchestration.
  - Universe handling and error paths.

A rough target is **80–90%** statement coverage, but more important is:
- Every important behavior and failure mode is tested.

---

6. Extensibility & Future-Proofing

6.1 Adding a new strategy – required steps

When adding a new strategy, the minimal “Definition of Done” is:

1. **Define `StrategyMetadata`**
   - With correct capability flags.
   - With sensible defaults (`default_strategy_config`, default universe path if applicable).

2. **Provide a pipeline module** (if non-trivial)
   - Example: `strategies.new_strategy.pipeline`.
   - Implements `run_daily_scan(...)` with a standard signature for two-stage pipelines.
   - Handles its own universe loading/validation if `requires_universe=True`.

3. **Register hooks / plug-ins**
   - Strategy MUST be discoverable via capabilities or a small hook registry:
     - e.g. `strategy_hooks.register(strategy_name, run_daily_scan=...)`.

4. **UI/CLI integration**
   - Provide adapter functions to build config from UI/CLI values.
   - Do **not** embed strategy logic directly in UI.

5. **Tests**
   - Strategy-specific unit tests (signals, config).
   - Pipeline tests integrating it into the common execution flow.

  - Architecture tests (capabilities and separation).

6.2 Plugin-style hook architecture (medium-term goal)

Longer term, strategy pipelines SHOULD follow a simple plug-in convention:

- `StrategyMetadata.pipeline_module: Optional[str]`
- The named module SHOULD expose standardized hook functions:
  - `run_daily_scan(...)`
  - `run_backtest(...)`
- Core orchestration:
  - Dynamically imports the module.
  - Calls hooks based on capabilities, not names.

Goal: Adding a new strategy should not require changes to the central pipeline logic.

---

7. Optional Dependencies & Environment Independence

- Core libraries MUST NOT hard-depend on optional UI frameworks (e.g. Streamlit).
- Use feature flags such as `HAS_STREAMLIT` in UI-specific modules only.
- Import order MUST be designed so that running tests or CLI in a headless environment:
  - Does not require UI libraries to be installed.
  - Does not crash due to missing UI modules.

Tests SHOULD include a scenario that ensures:
- Pipeline modules can be imported and executed without Streamlit (or any other optional component).

---

8. Definition of Done – Pull Request Checklist

For any non-trivial change, the author and reviewer SHOULD check:

1. **Metadata & Capabilities**
   - [ ] New or changed strategies have correct, documented `StrategyMetadata`.
   - [ ] No name-based branching introduced in core code.

2. **Robustness**
   - [ ] Inputs are validated early in the right layer.
   - [ ] Exceptions are specific; broad `except Exception` avoided or justified.
   - [ ] Error messages include useful context (paths, shapes, params).

3. **Modularity**
   - [ ] Strategy-specific logic resides in dedicated modules, not in generic pipeline or UI.
   - [ ] UI code is limited to presentation and config assembly.

4. **Config & UX**
   - [ ] Effective config is visible (UI preview or CLI print option).
   - [ ] Invalid user inputs are blocked with clear messages.

5. **Testing**
   - [ ] New tests added for happy path AND key failure paths.
   - [ ] Architecture/separation tests updated if relevant.
   - [ ] `PYTHONPATH=src python -m pytest tests` passes locally.

6. **Future-Proofing**
   - [ ] New design doesn’t paint us into a corner (no new strategy-specific hacks in core).
   - [ ] Hooks/capabilities are well documented for future contributors.