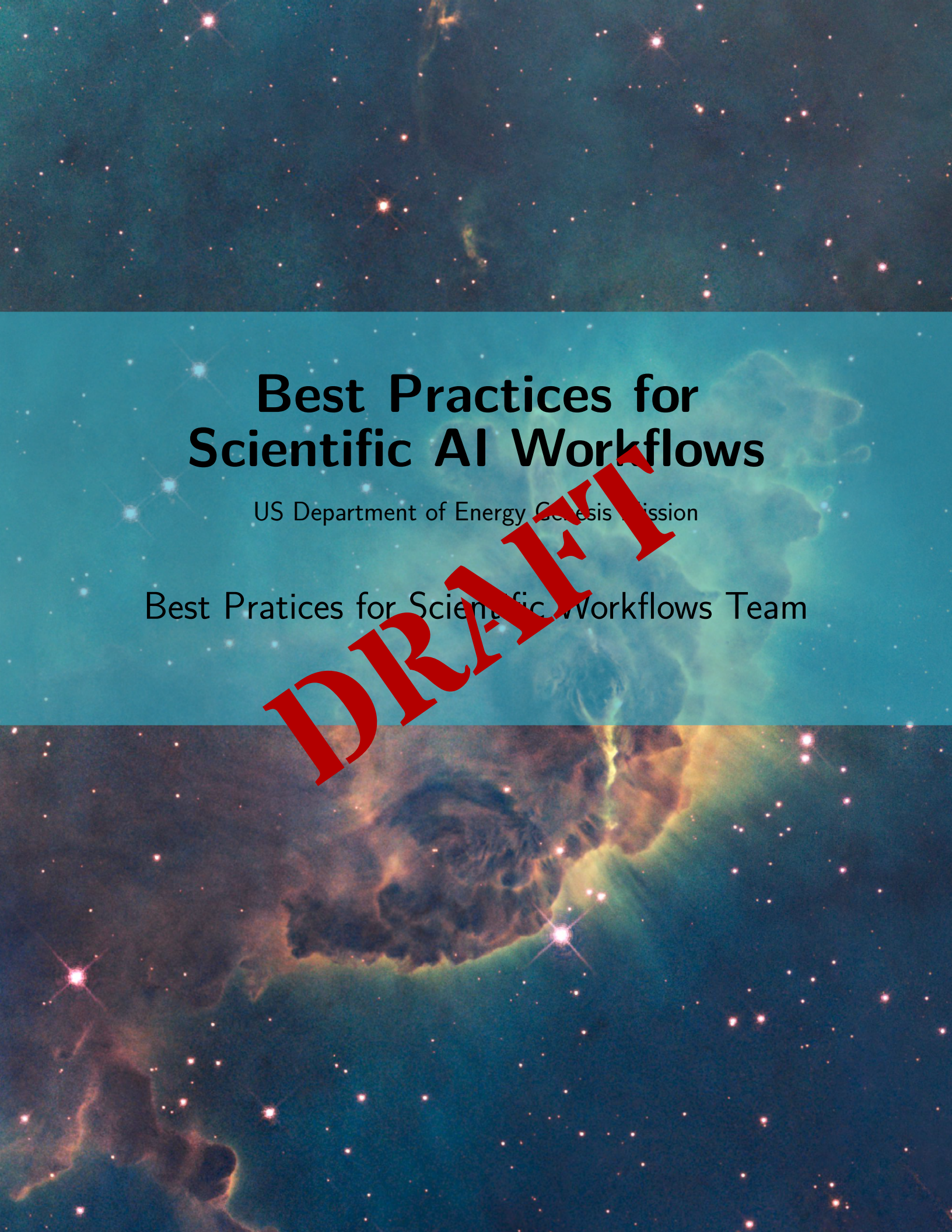


# Best Practices for Scientific AI Workflows

US Department of Energy Genesis Mission

Best Pratices for Scientific Workflows Team

## Editors

The following contributed editorial oversight of the report and serve as the corresponding authors for the report.


- Robert Underwood runderwood@anl.gov, ANL
- Rafael Ferreira da Silva silvarf@ornl.gov, ORNL
- Massimiliano Lupo Pasini lupopasinim@ornl.gov, ORNL
- Rinku Gupta rgupta@anl.gov, ANL
- Franck Cappello cappello@anl.gov, ANL


## Authors

The following individuals authored portions of the final report. There names are listed in alphabetical order by last name. Affiliations are listed in Table 1

- Thomas Britton BPSW, JLAB – Chapter 5, and 2.
- Franck Cappello BPSW, ANL – Chapters TODO Chapter Contributions
- Murat Keçeli BPSW, ANL, – Chapters 14, 2, 8.
- Jonghwan Kwon, TODO affiliations, GRID – Chapters 14, 15, 19
- Hongwei Jin TODO affiliations, GRID – Chapters 3, 8
- Yijiang Li TODO affiliations, GRID – Chapters 10, 18 20
- Massimiliano Lupo Pasini, ORNL – Chapters 7, 2, 20, and 15
- Nagi Rao, BPSW, ORNL – Chapters 2 and 9
- Nathan Tallent, BPSW, PNNL – Chapters 12, 11, 5
- Mikhail Titov, BPSW, BNL – Chapters 15, 11, 10
- Robert Underwood BPSW, ANL – Chapters 1 - 23
- Helia Zandi, BPSW, ORNL – Chapters 2, 3, 8, 11, 15, 16, 19, 20, 21

Table 1: Table of Affiliations

**Institutions**
**ANL** – Argonne National Laboratory
**BNL** – Brookhaven National Laboratory
**JLAB** – Jefferson Laboratory
**ORNL** – Oak Ridge National Laboratory
**PNNL** – Pacific Northwest National Laboratory

**Genesis Mission Teams**
**BPSW** – Best Practices for Scientific Workflows Team
**GRID** – GRID AI Model Team

As explained in Section 1.6, large portions of this report were initially drafted using generative AI using source documents from the AI model teams that are part of the US Department of Energy Genesis Mission.

## Acknowedgements

This report documents the efforts of the US Department of Energy Genesis Mission to utilize AI to accelerate the productivity of science over the course of the decade. As such, it reflects the contributions of hundreds of scientists across the Department of Energy National Laboratories as well as their partners in universities, industry, and other governmental agencies. We acknowledge their contributions in developing the science reported here.

This work was produced at Argonne National Laboratory under Contract No. DE-AC02-06CH11357 with the U.S. Department of Energy.

## License information

*First release, May 2026*

# Contents

# Part I

# Introduction

# 1. Best Practices for Scientific Workflows?

## 1.1 What are Best Practices?

Best Practices are a term of art referring to commonly used practices to address common patterns. Best Practices can be conceptualized as "Patterns," which were popularized by Gamma et al., who borrowed the term and concept from the work of Architect Christopher Alexander [Gam+]. As quoted in Design Patterns of Object-Oriented Software, a pattern "describes a problem which occurs over and over again in our environment" and presents "the core of the solution to that problem, in such a way that you can use the solution a million times over, without ever using it the same way twice." This definition emphasizes that Best Practices are universal, reusable solutions that must be adapted to the specific context of their application.

These conceptual "Patterns" are fundamentally structured around four essential elements, each contributing to a complete understanding and application of the Best Practice. These four elements are: the Name of the pattern, which serves aid communication and memory of the pattern; the Problem that describes the context in which the pattern is applicable; the specific Techniques used to solve the problem; and the resulting Consequences (positive and negative) that occur upon the pattern's implementation.

Adopting and mastering design patterns equips AI engineers with a repertoire of proven, evidence-based techniques that serve as industry-wide standards and benchmarks, allowing teams to consistently achieve superior results while minimizing architectural risk and maximizing performance; by encapsulating best-practice solutions for recurring challenges-such as structuring modular model pipelines, orchestrating data preprocessing, abstracting training-loop logic, and standardising evaluation and deployment workflows-design patterns provide clear, reusable blueprints that streamline decision-making, enhance code readability and maintainability, and accelerate the delivery of robust, scalable AI systems, as highlighted in contemporary write-ups on object-oriented design methodologies applied to machine-learning contexts.

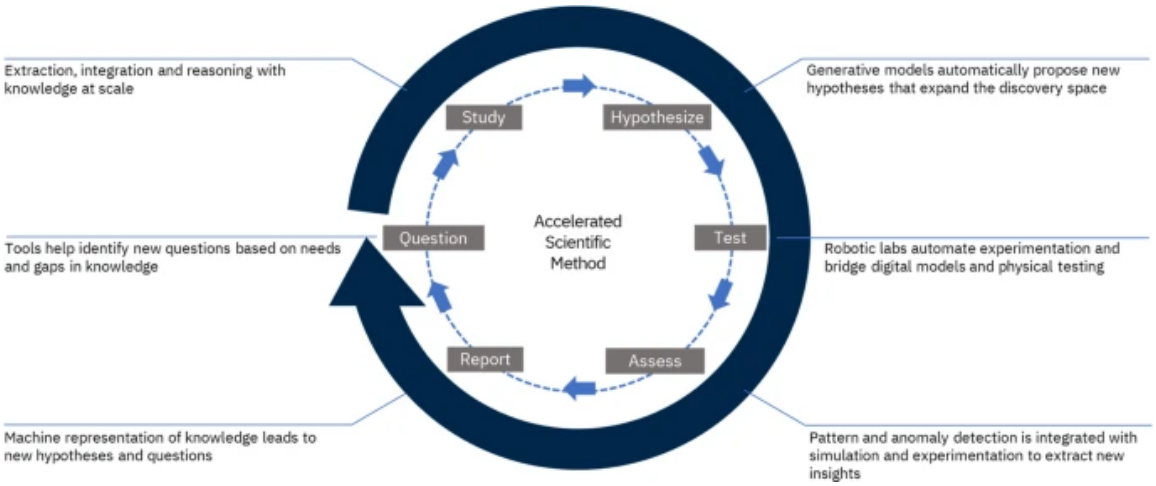



Figure 1.1: Overview of the scientific workflow accelerated by AI from [Pyz+22]

## 1.2 What are Scientific Workflows?

A scientific workflow accelerated by AI-also called an AI-based scientific "campaign"-is a sequence of research-oriented tasks (such as problem formulation, hypothesis generation, literature review, experimental design, execution or acceleration of experiments/simulations/observations, results analysis, and findings presentation) in which at least one critical task is carried out by an AI model (e.g., a large language model, deep- or convolutional neural network, graph neural network, diffusion model, Bayesian system, or any other advanced AI technique); the workflow may be fully automated or may involve humans in the loop, allowing researchers to intervene, validate, or steer the process while still benefiting from AI-driven speed, insight, and scalability.

Figure 1.1 illustrates an accelerated scientific method that integrates various tools and technologies at each stage of the research cycle. The process begins with the Study phase, where researchers extract, integrate, and reason with existing knowledge at scale to understand the current state of a field. Next, in the Hypothesize stage, new hypotheses are systematically proposed that expand the discovery space beyond what might be immediately apparent. These hypotheses then move to the Test phase, where experiments are conducted to bridge computational models with physical testing and validate or refute the proposed ideas. Following experimentation, the Assess stage employs pattern and anomaly detection integrated with simulation and experimentation to extract new insights from the results. The findings are then documented in the Report phase, where structured representation of knowledge enables the generation of new hypotheses and questions. Finally, the cycle returns to the Question stage, where analytical tools help identify new research questions based on needs and gaps in knowledge, ensuring the process continuously builds upon itself. This iterative loop creates a self-reinforcing system where each completed cycle informs and accelerates the next round of scientific discovery. Artificial intelligence has the potential to dramatically accelerate each stage of this workflow: from mining and synthesizing vast literature databases during the study phase, to generating novel hypotheses, designing and executing experiments, detecting subtle patterns in complex data, and identifying promising new research directions that might otherwise be overlooked.

## 1.3 How was this report generated?

This report was developed to answer urgent questions needed to help the Genesis Mission scale AI-enabled scientific workflows across the Department of Energy. A best-practices catalog is necessary because teams need a shared way to identify what is working, why it is working, and how those lessons can be reused across domains rather than rediscovered independently. For scientists, such a catalog helps showcase genuine impact, validate that AI is aligned with the scientific task, refine workflows by saving time and avoiding repeated bugs through reuse, and scaffold more structured training for teams. AI also the development and information in this report itself by scaling the task of helping probe teams making submissions for incompleteness in their responses, assisting the survey team with the analysis and synthesis of responses at scale where human review is impractical enabling rapidly surfacing credible recurring patterns. With this document, teams can quickly use AI to support adaptation of those patterns to different scientific application contexts and search to find the specific patterns most relevant to their current challenges.

The material in this report was collected through a staged process that combined structured questionnaires, broad outreach, AI-assisted synthesis, and human review. The team first developed and refined the core questions around domains, funding, AI advantage, workflow integration points, supporting evidence, AI models and capabilities, evaluation methods, success stories, and priority tasks. That approach was then iterated through roughly ten pilot-study refinements, expanded to about a dozen Genesis Mission seed teams using a detailed form, and finally scaled to approximately 143 responses across DOE using a lighter-weight form. Pilot studies were reviewed by humans, AI-based methods were developed and validated to help scale the analysis to complex-wide results, and the resulting core patterns are being systematically reviewed and curated by the team.

## 1.4 How is this Report Organized?

The remainder of the report is organized around recurring AI usage patterns rather than around individual model teams. Each pattern section synthesizes the workflows observed across the current document, identifies the teams that appear to be using the pattern, and summarizes the common problem, solution characteristics, design tradeoffs, reasons the pattern generalizes, and best-practice guidance.

To make those patterns easier to navigate, the catalog is grouped into five higher-level categories. *Foundation Patterns* describe the data, artifact, retrieval, and knowledge substrate that scientific AI workflows depend on. *Reasoning and Discovery Patterns* focus on how AI contributes directly to scientific exploration, modeling, optimization, and engineering support for discovery. *Operational Deployment Patterns* cover the challenges that arise when AI is embedded in live, large-scale, time-sensitive, or cross-facility systems. *Control and Assurance Patterns* collect the mechanisms used to govern AI behavior, protect safety and confidentiality, and keep workflows inside acceptable technical and institutional boundaries. Finally, *Assessment Patterns* address how teams evaluate whether AI creates real workflow-level advantage once it has been integrated into practice.

This organization is intended to move from underlying substrate, to scientific reasoning, to operational use, to governance, and finally to end-to-end assessment. The categories are not perfectly exclusive, since several patterns are intentionally cross-cutting, but they

provide a useful structure for understanding how the different best practices relate to one another.

## 1.5 Structure of a Pattern

The individual patterns are organized as follows:

- **Problem** describes the context in which the pattern applies
- **Characteristics of the solution** describes the components or steps to the solution and how they interact
- **Design Tradeoffs**: the key tradeoffs to consider when using the pattern
- **Implementation Advice** practical things to consider when implementing the pattern
- **Relationships to other Patterns**: how this pattern should be used in the context of other patterns

## 1.6 Note on Use of Generative AI

The contents of this report were generated in part using artificial intelligence and are in the process of being reviewed for accuracy by the team. As such, take appropriate precautions regarding the accuracy of the materials presented herein.

# AI Usage Patterns

# Part II

# Foundational Data Patterns

Foundation patterns establish the information substrate that the rest of an AI-enabled scientific workflow depends on. These chapters focus on how data, artifacts, prior cases, and organizational knowledge are captured, versioned, retrieved, and made reusable across time, teams, and tools. They come first because later reasoning, deployment, and governance patterns are much harder to implement well when the underlying data and knowledge layer is fragmented, weakly described, or difficult to trace.

# 2. Data Pipelines, Collection, and Automation

## 2.1 Problem

Scientific AI workflows depend on data that is rarely born in a clean, model-ready form. Instead, teams must gather information from instruments, simulations, archives, logs, operator notes, derived features, and external reference sources, often across different formats, time scales, quality levels, and access controls. The challenge is not just moving bytes from one place to another. Teams must determine what each datum means, whether it is trustworthy, how it relates to other records, how it was produced, and whether it can be reused safely for training, inference, or evaluation.

Naive approaches usually begin with notebooks, one-off scripts, manual file naming conventions, and ad hoc transformations embedded directly into downstream analyses. That works briefly for a single contributor, but it breaks under growth. Inputs change format, labels drift, files are overwritten, assumptions about units or coordinate frames become implicit, and retraining can no longer be reproduced. Scientific AI adds further pressure because models are sensitive to subtle inconsistencies: duplicated records, hidden leakage between train and test, stale labels, missing metadata, and silent preprocessing changes can alter results as much as model architecture choices. Data collection is also often adaptive; model failures reveal missing regions of the design space, which means the pipeline must support not only cleaning and delivery but also targeted acquisition of new data.

This pattern is therefore about building data preparation as a first-class scientific system rather than a prelude to the real AI work. Teams such as Fusion, GRID, and Axess plausibly face heterogeneous technical inputs and feedback from modeling outcomes; Prometheus and CM2US fit the pattern where scaling, reuse, and reproducibility depend on disciplined data handling. The lesson generalizes widely because almost every AI-enabled scientific workflow is limited by the quality, traceability, and maintainability of its data pipeline.

A growing extension of this problem is that many workflows are no longer preparing data only for humans and models in isolation, but also for AI agents, orchestration systems, and retrieval layers that need machine-consumable interfaces. In those settings, a pipeline

fails not only when data are dirty or incomplete, but also when artifacts are not structured enough for downstream tools to consume safely. Free-form files, undocumented intermediate outputs, and one-off tool interfaces create friction for agentic workflows in the same way that inconsistent labels and missing provenance create friction for training and evaluation.

MAIQMag is a direct instance of this pattern because its inference depends on standardizing multimodal measurements produced across DOE laboratories and user facilities together with synthetic data and forward-solver outputs generated on HPC resources. The pipeline should preserve sample and experiment identifiers, modality, units, calibration state, uncertainty, facility provenance, solver configuration, and the links among a measured observable, its synthetic counterpart, and the candidate Hamiltonian that produced it. Automated validation and alignment at these boundaries are prerequisites for joint inference: without a shared representation, disagreement caused by units, metadata, or preprocessing can be mistaken for disagreement with the physical model.

GridAI ingested raw data from several established external sources — Google DeepMind's OPFData [Lov+24] (pulled from Google Cloud Storage as '.tar.gz' archives covering the 10 PGLib benchmark cases), Texas A&M's ActivSg2000 synthetic Texas grid [Bir+17](Series24 dynamics cases used as base networks for hurricane and ExaGrid scenarios), the PGLib-OPF benchmark library [Bab+21] (66 cases solved with PowerModels.jl), ERCOT historical hourly load data [Ele26](driving the Texas2k ExaGrid simulations), and Argonne's A-LEAF reliability-assessment exports — and then re-ran or post-processed each through HPC pipelines on Perlmutter, Aurora, and Frontier using ACOPF and contingency solvers like Ipopt, ExaModels.jl, and PowerModels.jl. These heterogeneous raw inputs were normalized into a common heterogeneous-graph (bus/generator/branch/transformer) schema and stored in SQLite, HDF5, or PyTorch Geometric formats, with a strict verify-and-resume pipeline that checked every database for completeness and reprocessed corrupted or missing files.

Key challenges included race conditions in parallel group-level processing, corrupted '.tar.gz' downloads from OPFData producing gzip/JSON errors, an MLD dataloader bug that mislabeled load-served targets as slack-generator targets, missing transformer labels in hurricane contingencies that required patched datasets, and pervasive ACOPF infeasibility under combined branch/generator outages that forced a multi-stage progressive relaxation strategy. The main lessons learned were to parallelize at the case level rather than the group level, keep temporary files to avoid cleanup races, always pair generation with automated verification and resumable workflows, treat cross-source schema unification as a first-class engineering task, and explicitly version corrected datasets so downstream training never silently consumes buggy or incomplete data.

## 2.2 Characteristics of the Solution

The common solution begins with an explicit *ingestion layer*. Raw data enters through controlled interfaces that identify source, timestamp, provenance, access restrictions, and basic integrity properties. This matters because reproducibility starts at the point of entry, not after transformation.

A second recurring component is the use of *canonical schemas or data contracts*. These define required fields, units, coordinate conventions, label semantics, and relationships among records. Data contracts matter because they reduce hidden assumptions and allow multiple teams or tools to interoperate without constant manual interpretation.

Third, effective systems decompose processing into *modular transformation stages*. Cleaning, normalization, feature extraction, label generation, alignment, splitting, and packaging are separated into inspectable stages rather than merged into a single script. Modularity improves testing, selective reuse, and fault isolation.

Fourth, robust implementations use *workflow orchestration and checkpointing*. DAG-based execution, resumable stages, and intermediate materialization allow teams to recover from partial failure and recompute only the affected portions when logic or source data changes. This matters when datasets are large or expensive to regenerate.

Fifth, strong pipelines capture *metadata and lineage by default*. Each dataset or derived artifact should be traceable to source inputs, transformation code, parameters, and validation outcomes. Without lineage, teams cannot explain why a model changed or whether a dataset can be trusted.

Finally, mature pipelines include *feedback loops from model behavior back to collection*. Error analysis, uncertainty maps, failure clusters, and coverage gaps should inform what data to collect next. This closes the loop between data operations and scientific utility.

A further recurring component in newer workflows is *AI-ready artifact design*. Intermediate outputs, tool interfaces, and final data products are emitted in structured, schema-constrained, and machine-consumable forms rather than only as human-oriented files or ad hoc logs. This matters because retrieval systems, agent frameworks, and downstream automation depend on stable contracts: typed records, explicit metadata, versioned payloads, and well-defined tool boundaries. Without those properties, teams end up rebuilding brittle wrappers around every workflow stage.

## 2.3 Design Tradeoffs

*Modularity vs. upfront engineering cost* – Modular pipelines are easier to test and evolve, but they require schema design, interface discipline, and orchestration infrastructure. Scientists should invest more heavily when data sources are expected to change, multiple contributors will reuse the pipeline, or the dataset will outlive a single experiment.

*Generality vs. domain-specific richness* – Generic infrastructure improves portability, but scientific value often depends on domain-aware validators, physical-unit handling, and custom label logic. A practical landing point is to keep orchestration and storage generic while allowing domain-specific validation and enrichment modules at the edges.

*Reproducibility vs. metadata overhead* – Rich provenance and immutable dataset versions improve auditability, but they also add storage and bookkeeping. Teams should capture enough metadata to reconstruct results and detect drift, while being selective about expensive intermediate retention when full materialization is unnecessary.

*Scalability vs. operational complexity* – Distributed executors, message buses, and DAG engines can handle large or continuous data streams, but they raise the operational bar. Small teams should avoid prematurely adopting heavy infrastructure unless data volume, scheduling needs, or collaboration patterns clearly justify it.

*Strict validation vs. edge-case flexibility* – Early rejection of malformed data prevents silent corruption, but unusual records may contain scientifically important novelty. Good practice is to quarantine suspicious inputs rather than deleting them outright, allowing expert review and later rule refinement.

*Human-readable convenience vs. machine-consumable structure* – Scientists often prefer

flexible notebooks, narrative logs, and bespoke outputs during early development, while agents and downstream automation need stable schemas, typed interfaces, and explicit metadata. The right landing point is usually not to eliminate human-readable artifacts, but to pair them with structured counterparts so the same workflow can support both expert interpretation and reliable AI integration.

## 2.4 Implementation Advice

- Define canonical data contracts early.
  Specify schemas, units, naming conventions, coordinate systems, allowed ranges, label semantics, and split policies before the dataset becomes large. Practical implementations may use JSON Schema, typed tabular schemas, or domain-specific manifest formats. Early contracts reduce downstream rework and make collaboration materially easier.
- Capture provenance by default.
  Every derived dataset should record source identifiers, transformation code versions, parameters, timestamps, and validation outcomes. This can be stored in manifests, registry tables, dataset cards, or workflow metadata stores. Provenance is essential for reproducing training corpora, tracing regressions, and answering whether two datasets differ only in content or also in preprocessing logic.
- Make pipeline stages modular and testable.
  Separate ingestion, cleaning, normalization, labeling, feature extraction, splitting, and packaging into well-defined steps with narrow interfaces. Then test them individually using fixture datasets, golden records, and contract tests. Modularization reduces the blast radius of changes and makes selective reuse possible across projects.
- Support resumability and selective recomputation.
  Large scientific datasets are too expensive to rebuild from scratch after every failure or small code change. Use checkpoints, content hashes, stage outputs, or DAG caches so the system can rerun only the affected steps. This is especially important when some stages depend on costly simulations, remote archives, or manual labeling.
- Validate data at multiple points, not just at ingestion.
  Data can become invalid after joins, unit conversions, feature generation, or labeling. Add checks at stage boundaries for shape, range, completeness, duplication, temporal leakage, class balance, and domain-specific physical plausibility. Multi-point validation catches errors where they are introduced instead of attributing them vaguely to bad data later.
- Emit AI-ready artifacts alongside human-readable outputs.
  Do not assume that a spreadsheet, PDF, free-form log, or one-off text file is sufficient for downstream AI use. When a workflow output may later be consumed by retrieval systems, agents, or automation, pair the human-readable representation with a structured artifact such as schema-validated JSON, typed tables, explicit metadata manifests, or versioned tool outputs. The Quarks to Cosmos (Q2C) cosmological search example illustrates the need: the final answer is useful to scientists because it includes visual and quantitative evidence, but it becomes reusable by AI systems only if the workflow also emits machine-consumable records describing the simulation, timestep, mass threshold, halo identifiers, classification evidence, workflow run, access

context, and links to derived artifacts. This reduces wrapper debt and makes later integration far less brittle.

- Close the loop between model failures and new data generation.
  Use model residuals, uncertainty hotspots, cluster-level error analysis, or red-team findings to identify missing or weakly covered regions of the dataset. Then feed those findings into simulation campaigns, new measurements, targeted curation, or relabeling queues. This turns the data pipeline into an adaptive system rather than a static warehouse.

**CM2US example: restartable and portable atomistic workflows.**
In CM2US, the `matsim-agents` workflow exposed three practical data-pipeline and deployment challenges: keeping long atomistic campaigns restartable, separating Python-based AI environments from the compiler/MPI environments required by DFT codes, and making the same workflow portable across DOE leadership-class systems with different accelerator and software stacks. The solution was to serialize workflow state across the full loop, including relaxation results, uncertainty scores, DFT handoff decisions, DFT convergence status, active-learning iteration metadata, and timing information. This allowed failed or incomplete campaigns to resume without repeating completed expensive calculations.

The workflow also separated the HydraGNN/ASE/Python execution path from the Quantum ESPRESSO or VASP execution path. HydraGNN inference and structure relaxation ran in a Python environment appropriate for AI workloads, while DFT jobs were launched through backend-specific scripts that loaded the compiler, MPI, GPU runtime, and site modules required by the selected first-principles code. This separation avoided fragile monolithic environments in which the AI surrogate, the atomistic simulation tools, and the DFT backend all had to share the same software stack.

Portability across DOE supercomputers was addressed by isolating site-specific setup into platform-specific launch and build scripts. On NERSC Perlmutter, the workflow targets NVIDIA GPU software environments; on OLCF Frontier, it must operate with AMD GPU and ROCm-based environments; and on ALCF Aurora, it must account for Intel GPU and oneAPI-based environments. Rather than hiding these differences, `matsim-agents` makes them explicit through separate deployment paths for each system. This allowed the same scientific workflow pattern—LLM planning, HydraGNN relaxation, uncertainty-gated validation, DFT escalation, active learning, and reporting—to be reused across facilities while respecting each machine's scheduler, module environment, accelerator backend, and DFT build requirements.

The main lesson learned was that portable AI-driven scientific workflows on leadership-class HPC systems require more than portable Python code. The workflow must also make restartability, environment isolation, backend-specific DFT execution, and facility-specific launch logic first-class design concerns from the beginning.

**Questions for Future Expansion**
Which teams currently use explicit data contracts or schema registries, and what form do those contracts take?
Are there documented examples of model-error analysis driving new data acquisition in Fusion, Axess, or CM2US?

What orchestration tools or storage layers are actually in use across the referenced workflows?
Which metadata fields have proven essential for reproducibility, and which have been costly without much benefit?
Which workflows are already emitting AI-ready artifacts such as schema-constrained outputs, manifests, or agent-consumable tool interfaces?
Are there recurring failure modes such as unit mismatch, temporal leakage, label drift, or unstructured outputs that deserve concrete case studies?

**Figure Suggestions**
Primary figure: Scientific data pipeline DAG showing ingestion, schema validation, cleaning, normalization, labeling, feature extraction, packaging, lineage capture, and model-feedback-driven recollection.
Optional figure: Human-readable versus AI-ready artifact comparison showing a free-form note or report paired with a structured manifest, typed record, or schema-validated output.

## 2.5 Relationships to Other Patterns

- Use with *At-Scale AI Training and Inference* when reproducible retraining and serving depend on modular, reliable data preparation.
- Use with *Physics-Informed AI Patterns* when units, boundary conditions, and derived labels must preserve scientific meaning.
- Use with *Federated and Confidential Data Patterns* when pipelines must package and protect the data artifacts, metadata, and lineage that can be shared across restricted environments.
- Use with *Advanced AI Data Search* when the pipeline is responsible for creating the metadata, manifests, and structured artifacts that search systems later depend on.
- Use instead of *Institutional Memory and Knowledge Capture* when the primary problem is preparing operational data for AI use rather than preserving human and organizational knowledge.

# 3. Versioning and Change Management

## 3.1 Problem

Scientific AI workflows evolve continuously, but the outputs are often treated as if only source code changes matter. In reality, the behavior of a workflow can change because training data was refreshed, a preprocessor was updated, a prompt template changed, a retrieval index was rebuilt, a model checkpoint was replaced, a dependency version shifted, or an orchestration policy was tuned. In AI-heavy work, these non-code changes are often exactly what determine the result. The core problem is therefore broader than ordinary software version control: teams must be able to explain what was run, on which inputs, with which model and metadata, under which policies, and how those ingredients differ from prior runs.

Naive solutions fail because they track only one layer of the system. Git alone cannot tell a scientist which dataset snapshot produced a figure, which model checkpoint generated a recommendation, or whether a performance regression came from a prompt change rather than a code change. Shared folders and mutable file names create an even worse failure mode: important artifacts are silently overwritten, so there is no trustworthy baseline to recover. Spreadsheet-based bookkeeping can work briefly for a single researcher, but breaks under collaboration, automation, and branching experimentation. The problem is especially acute for teams such as Prometheus, MOAT, GRID, Microelectronics, and MAIQMag because they combine code, models, data products, and operational logic in workflows where reproducibility and rollback matter.

AI systems also magnify the consequences of weak change management. Two outputs can look similar while being generated by meaningfully different model states. Small prompt or retrieval changes can alter qualitative behavior without leaving any obvious trace in standard logs. Regenerating an artifact may be expensive or impossible if the original upstream data or environment has moved on. Scientists therefore need change management that treats data, models, prompts, workflow outputs, and execution metadata as first-class artifacts rather than incidental by-products.

This pattern generalizes to any collaborative scientific program that wants to move fast without losing the ability to defend a result. The question is not whether change will happen; it is whether the team can understand and govern that change when it does.

The GridAI team currently relies on Git for code versioning, using version tags to mark releases. Once a model finishes training and passes validation, it is hosted on Hugging Face with a corresponding version tag and a model card, which makes it easy to track and retrieve specific model versions. On the data side, the datasets come from public sources, and formal data versioning is not yet in place—this is one of the gaps the team has identified. Since the data can go through different preprocessing steps, the lack of versioning makes reproducibility harder than it should be, and the processed data is essentially a mutable artifact that causes some pain when reproducing earlier results.

Based on these observations, a few suggestions could help the broader team. First, adopting data version control (DVC) to track datasets—especially since data often gets reprocessed—would pair well with Git and help close the reproducibility gap. Second, because training runs can involve different data sources, models, and training methods, it is worth clearly tracking the data and configuration tied to each model checkpoint so they can be traced later. Existing MLOps tooling can also help here—for example, integrating with AmSC's MLflow service to capture training logs, data references, and evaluation results in one place for clear traceability. Finally, when training across different infrastructure, distinguishing runs clearly is important, since different hardware architectures can lead to subtle differences in results.

## 3.2 Characteristics of the Solution

The recurring solution treats models, data, prompts, workflow outputs, and metadata as first-class versioned artifacts. It favors immutable objects, automatic provenance capture, branching and merge workflows, searchable registries or knowledge graphs, and downstream hooks that react to changes proportionally.

A strong implementation starts with artifact identity. Every important dataset snapshot, model checkpoint, prompt set, evaluation report, and derived output gets a stable identifier. This matters because teams cannot reason about lineage if they cannot unambiguously name what was used.

Immutability is the next key component. Instead of editing important artifacts in place, teams publish new versions and preserve old ones. This matters because reproducibility depends on the ability to return to exactly the prior state, not an approximation of it.

The pattern also requires automatic provenance capture at execution time. The workflow should record code revision, input artifact versions, environment information, parameters, prompts, and output identifiers without depending on manual note-taking. This matters because manual provenance is incomplete precisely when the workflow becomes complex enough to need it.

A searchable registry or lineage store is usually necessary as well. Scientists need to answer questions such as "Which runs used this model version?" or "What changed between these two outputs?" Registries, metadata stores, or graph-based lineage services make those questions operational rather than forensic.

Finally, mature teams connect change events to downstream validation. If a dataset, prompt template, or model changes, the system should trigger proportional reevaluation

rather than assuming the old evidence still applies. This matters because stale validation is one of the most common sources of false confidence in AI workflows.

## 3.3 Design Tradeoffs

*Provenance completeness vs. storage cost* – Full traces improve reproducibility and auditability, but can be expensive to store and manage. Scientists should preserve exhaustive provenance for release candidates, published results, and safety-relevant workflows, while using summarized provenance for low-stakes exploratory runs when storage is constrained.

*Immutability vs. ease of use* – Immutable artifacts make lineage clear, but require teams and tools to stop editing important assets in place. The right landing point is usually mutable workspaces, immutable published artifacts: researchers can iterate freely locally, but anything shared, cited, or deployed receives a new immutable version.

*Branching flexibility vs. merge complexity* – Parallel exploration becomes safer, but reconciling diverged binary or heterogeneous artifacts is difficult. Branches are valuable for experiments, but teams should avoid long-lived divergence for data schemas, prompts, or model registries unless they also define clear merge or supersession rules.

*Automation of change propagation vs. cascade failures* – Hooks improve freshness and coordination, but can trigger expensive or noisy recomputation chains. In practice, not every change should trigger the same response. Small documentation changes may need no action; a new model checkpoint may require full regression testing. Proportional triggering is more sustainable than universal rebuilds.

*Reproducibility guarantees vs. workflow speed* – Strong revalidation after change improves trust, but can slow exploratory research. A common compromise is to separate exploratory mode from governed mode: fast iteration for local work, then stricter capture and validation for shared milestones, release candidates, and published conclusions.

## 3.4 Implementation Advice

- Version all critical artifacts, not just source code.
  Treat datasets, model weights, prompt templates, retrieval indexes, evaluation outputs, configuration files, and generated reports as versioned assets. Tools can vary, but the principle is stable: if changing an object can change a scientific conclusion or system behavior, it needs an identity and a history. This closes the common gap where code is well tracked but the behavior-defining artifacts are not.
- Prefer immutable artifacts with explicit lineage.
  Publish a new version instead of overwriting an existing checkpoint, dataset, or report. Record parent-child relationships such as derived from dataset X with model Y and configuration Z. This practice makes rollback possible and sharply reduces ambiguity when comparing results across time. Content hashes, append-only object stores, and registry-assigned version IDs are common implementation choices.
- Capture provenance automatically at execution time.
  Execution systems should write metadata without asking the researcher to remember every detail. Useful fields include git commit, container or environment identifier, parameter set, random seed, input artifact IDs, prompt version, model version, start

and end times, and output locations. Automatic capture is critical because manual provenance tends to fail exactly during urgent, complex, or collaborative work.

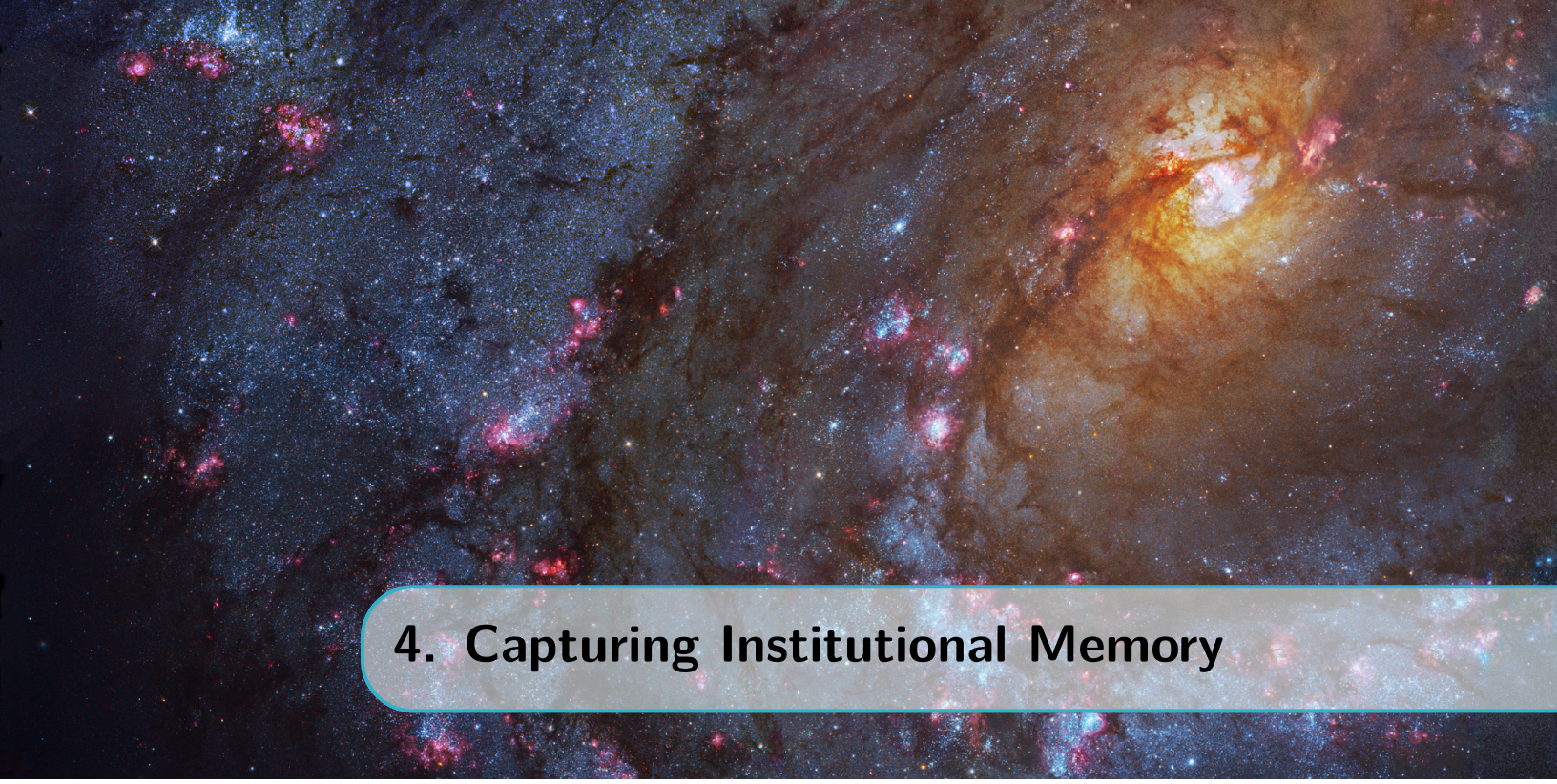
- Establish rollback and known-good release points.
  Define trusted baselines such as validated model releases, frozen datasets for a publication, or approved workflow configurations for operations. Label these clearly in the registry and preserve their associated evidence. Scientists then have a safe place to return when a new model degrades performance or a data refresh introduces regressions.
- Use branching and review gates for experimental changes.
  Branches are useful for trying new model families, preprocessing ideas, or prompt strategies without destabilizing the shared baseline. Pair that flexibility with review gates for changes that affect shared artifacts or downstream users. For example, require approval before promoting a new checkpoint to production, before changing a canonical schema, or before replacing a retrieval index used by other teams.
- Tie change events to proportional validation.
  Connect artifact changes to the right level of testing. A prompt edit may require targeted behavioral regression tests; a new dataset version may require full retraining plus benchmark comparison; an orchestration change may need end-to-end replay. The practical goal is to avoid both underreaction and overreaction by matching the validation burden to the likely blast radius of the change.

**TODO for Future Expansion**
Which artifact registries or metadata systems are actually used by Prometheus, MOAT, GRID, Microelectronics, or MAIQMag?
What is the recommended minimum provenance schema for this report's target audience?
Which artifacts are currently mutable in practice and causing reproducibility pain?
Are there existing release-gate or rollback procedures in the repository context that should be named directly?
How should binary artifacts, large datasets, and prompt libraries be merged, superseded, or archived across branches?

**Figure Suggestions**
Primary figure: Artifact lineage and change propagation graph showing code, datasets, prompts, model checkpoints, indexes, configurations, and outputs connected through immutable versions and downstream validation triggers.
Optional figure: Change impact matrix showing different artifact changes and their corresponding validation requirements, such as prompt regression tests, retraining, or end-to-end replay.

## 3.5 Relationships to Other Patterns

- Use with *AI-Augmented Software Development* when AI-generated modifications must remain reviewable and reversible.
- Use with *Agent-as-User Delegation* when agent actions need explicit audit trails and rollback paths.
- Use with *Data Pipelines and Collection Patterns* when data transformations and datasets need lineage across time.
- Use instead of *Institutional Memory and Knowledge Capture* when the primary requirement is controlled evolution of artifacts rather than broader retention of organizational knowledge.



# 4. Capturing Institutional Memory

## 4.1 Problem

Scientific workflows routinely depend on knowledge that is only partially encoded in formal artifacts. Important context lives in troubleshooting notes, shift logs, operator experience, internal documentation, prior runs, literature collections, postmortems, informal conventions, and remembered exceptions. In many labs and facilities, experts know which failure modes are common, which datasets are trustworthy, which analyses were previously tried, and which workarounds are safe, but that knowledge is fragmented across people, tools, and time. The AI-specific problem is that useful context exists, but is rarely available in a form that can be queried, summarized, compared, or reused reliably during live work.

Naive approaches fail because they assume that storing documents is the same as preserving knowledge. Shared drives, ticket systems, wikis, and chat logs accumulate information, but under pressure users still cannot quickly answer questions such as "Have we seen this before?", "What fixed it last time?", "Which assumptions went into this prior analysis?", or "What does the local expert community already know about this class of case?" Search alone is not enough when the real need is case-based reasoning, synthesis across many records, or continuity after personnel change. Another failure mode is overreliance on a few subject-matter experts whose judgment becomes a bottleneck and whose departure creates sudden capability loss. These issues are increasingly important in AI-enabled scientific operations and data-intensive research programs, where prior cases and accumulated context can materially change what action is best.

The recurring challenge is therefore not just information retrieval, but converting scattered organizational knowledge into a usable memory layer for future workflows. This generalizes across scientific settings: troubleshooting large facilities, understanding datasets, onboarding new contributors, revisiting prior analyses, and preserving the rationale behind earlier decisions.

In many facilities, this tacit-knowledge problem is not secondary; it is an operational bottleneck. Naming conventions, unwritten procedures, equipment context, recovery prac-

tices, and decision rationales often live only in expert memory. As personnel rotate or retire, that knowledge loss directly increases training burden, recovery time, and operational risk.

## 4.2 Characteristics of the Solution

The recurring solution captures institutional knowledge as structured, queryable, and reusable workflow context rather than leaving it trapped in isolated files or human memory.

A first component is *case-oriented ingestion*. The system collects not only raw documents, but also prior incidents, analysis records, decisions, outcomes, and supporting evidence as identifiable cases. This matters because users often reason by analogy to prior events, not by reading a whole corpus from scratch.

A second component is *evidence-linked summarization*. Summaries, explanations, and recommended next steps should remain tied to the underlying records, logs, publications, or notes that support them. This matters because institutional memory is valuable only if users can verify where the remembered claim came from.

A third component is *cross-source synthesis*. Useful memory systems combine multiple sources such as documents, logs, datasets, prior runs, troubleshooting records, and literature rather than treating each repository in isolation. This matters because important operational or scientific knowledge is often distributed across several partial records.

A fourth component is *human feedback and curation*. Experts should be able to correct bad summaries, mark useful prior cases, annotate misleading results, and capture what actually resolved a situation. This matters because institutional memory improves over time only if the organization can reinforce good knowledge and suppress stale or incorrect guidance.

A fifth component is *observational capture of expert practice*. The system should preserve not only finalized notes, but also demonstrations of how experts work: annotated logbooks, decision rationales, screen recordings, voice notes, and structured records of deviations from nominal procedure. This matters because many high-value lessons are embedded in expert process rather than in final artifacts alone.

A sixth component is *continuity across time and personnel*. The workflow should preserve not only what happened, but also the rationale, assumptions, and outcomes attached to earlier work. This matters because turnover, long experimental gaps, and shifting responsibilities are common sources of knowledge loss.

Finally, mature systems expose the memory layer through *workflow-facing interfaces*. Scientists, operators, dashboards, and AI assistants should all be able to query the same governed source of prior cases and supporting context. This matters because knowledge capture has the most value when it is integrated into real decisions rather than consulted only after the fact.

## 4.3 Design Tradeoffs

*Breadth of capture vs. curation burden* – Capturing more records improves recall, but also increases noise and maintenance cost. Teams should favor broad ingestion for raw preservation, while investing curation effort in the highest-value case types such as fault recovery, validated analyses, and authoritative documentation.

*Summarization convenience vs. source fidelity* – Concise AI summaries reduce user effort, but can oversimplify, omit caveats, or misstate the evidentiary basis. The right landing point is usually summary plus provenance: users get a short answer first, with direct links to the supporting cases and records.

*Expert centralization vs. scalable reuse* – Heavy expert curation improves quality, but does not scale. Fully automated memory extraction scales better, but may capture too much weak or ambiguous content. A practical compromise is automated ingestion with lightweight expert correction and ranking signals.

*Local context vs. cross-project reuse* – Knowledge systems are most accurate when grounded in local conventions, but scientific organizations also benefit from reuse across teams and facilities. Scientists should preserve local nuance while normalizing enough metadata and case structure to make cross-project transfer possible where appropriate.

*Freshness vs. stability* – Continuously updated knowledge bases reflect recent operations, but they can also surface unreviewed or transient conclusions. Stable curated views are more trustworthy for high-consequence decisions. A useful compromise is to distinguish newly ingested material from validated or high-confidence institutional knowledge.

## 4.4 Implementation Advice

- Capture prior cases as first-class artifacts.
  Do not limit preservation to raw files or narrative notes. Represent incidents, analyses, troubleshooting episodes, and decision records as identifiable cases with timestamps, participants, outcomes, and links to supporting evidence. This makes the organizational memory searchable and reusable in a way that mirrors how practitioners actually reason.
- Tie generated summaries and recommendations to source evidence.
  When AI summarizes prior work or suggests related cases, show the user which documents, logs, or records support that output. Evidence-linked answers improve trust, make expert review faster, and reduce the chance that institutional memory becomes a source of untraceable folklore.
- Support expert correction and reinforcement.
  Let users mark whether a retrieved case was helpful, whether a summary was accurate, and what the actual resolution or lesson learned was. These feedback signals are essential for separating durable knowledge from stale or misleading material.
- Preserve rationale, not only outcomes.
  Store assumptions, tradeoffs, rejected alternatives, and adjudication notes alongside the final answer or action taken. Future users often need to know why a prior team chose a path, not merely what result they produced.
- Capture expert practice before it disappears.
  Do not rely only on retrospective documentation after a veteran leaves or after an incident has passed. Use standardized electronic logbooks, structured after-action capture, screen or voice annotations when appropriate, and agent-assisted prompting for missing context so operational expertise becomes machine-accessible while it is still fresh.
- Integrate institutional memory into operational workflows.
  Expose the memory layer where work already happens: troubleshooting tools, dataset

portals, internal assistants, dashboards, or experiment-planning interfaces. Knowledge capture is most valuable when it reduces time-to-understanding during real tasks rather than living in a separate archive that users forget to consult.

- Distinguish raw memory from validated memory.
  Not every prior case should carry the same authority. Separate recently ingested or weakly supported material from curated, high-confidence institutional knowledge so users can judge how much to rely on a retrieved result.

**Questions for Future Expansion**

Which workflows in the survey provide the strongest concrete examples of institutional-memory use, such as troubleshooting systems, dataset assistants, or literature-backed internal copilots?
What common case schema should be recommended for capturing prior incidents, analyses, and outcomes?
Which feedback signals are most useful for improving institutional memory systems: user clicks, explicit ratings, expert annotations, or downstream success measures?
How should validated institutional knowledge be distinguished from newly ingested but weakly reviewed material?
What examples best show the value of preserved rationale, not just preserved outputs?

**Figure Suggestions**

Primary figure: Institutional memory layer architecture showing logs, notes, prior runs, documents, incidents, and literature feeding a case-based memory system with evidence-linked summaries and workflow-facing queries.
Optional figure: Example case retrieval card showing summary, resolution, supporting evidence links, confidence or validation status, and expert feedback.

## 4.5 Relationships to Other Patterns

- Use with *Advanced AI Data Search* when stored knowledge must be retrieved and synthesized rather than merely archived.
- Use with *Versioning and Change Management* when memory should preserve why systems changed, who approved the change, what alternatives were rejected, and what downstream effects were observed. Common examples include architectural decisions, incident reviews, code review rationale, deployment exceptions, and documented workarounds that would otherwise disappear.
- Use instead of *Data Pipelines and Collection Patterns* when the key asset is organizational knowledge, not model-ready datasets.

# 5. Advanced AI Data Search

## 5.1 Problem

Scientific work increasingly depends on locating the right artifact among many kinds of artifacts: raw data, processed datasets, simulation outputs, instrument logs, publications, notebooks, model checkpoints, prompts, workflow runs, and provenance records. The difficulty is not just scale, but heterogeneity. The same scientific question may require exact filtering over metadata, semantic matching over text, traversal of lineage relationships, and access to version or policy context. The AI-specific problem is that ordinary search modes each see only part of the picture. File trees know paths, databases know fields, full-text engines know documents, and vector indexes know semantic similarity, but scientists often need all of these at once.

Naive solutions fail because they assume one retrieval mechanism can stand in for the rest. Folder conventions and shared drives break as soon as artifacts are generated by many workflows or copied across projects. Simple keyword search misses relevant results when naming conventions differ across teams, instruments, or scientific subfields. Pure metadata filtering works well only when the needed fields were consistently captured ahead of time, which is rarely true in evolving research environments. Pure vector search improves fuzzy recall, but it is hard to explain, weak at strict constraints, and often blind to provenance or access-policy structure. In practice, users need to ask compound questions such as find simulation outputs related to this parameter regime that were derived from dataset version X, validated after model release Y, and discussed in notes similar to this abstract. Single-mode search cannot answer that reliably.

This challenge is especially relevant for Prometheus, MOAT, FAMOUS, and CM2US because those teams appear to work across evolving corpora, generated artifacts, and data products whose value depends on lineage and context as much as on content. In FAMOUS, embeddings from a bacterial phenomic foundation model utilize multi-omics data to represent proteins in their system context, allowing the model to understand differences in environmental factors and organism. A companion GNN model can leverage insights from very large multimodal

knowledge graphs. The problem generalizes quickly once a scientific program accumulates enough artifacts that human memory and naming conventions are no longer sufficient. At that point, retrieval becomes part of the scientific infrastructure, not just a convenience feature.

The deeper issue is trust. Scientists do not merely want a relevant-looking result; they need to know why it was returned, whether they are authorized to use it, which version it refers to, and how it relates to upstream and downstream artifacts. Advanced AI data search succeeds when it improves both recall and decision quality, not when it simply adds an embedding model on top of a storage system.

## 5.2 Characteristics of the Solution

The recurring solution combines multiple retrieval modes rather than depending on one: graph traversal for provenance and relationships, metadata or relational filtering for exact constraints, semantic retrieval for fuzzy similarity, and full-text search for documents. These are wrapped in a queryable layer that serves both humans and agents while preserving versioning and access control.

A common component is a unified retrieval contract. Different artifact types should expose a shared minimum schema such as identifier, type, title or summary, provenance links, access policy, version, and pointers to payloads. This matters because retrieval quality collapses when every data source expresses the basics differently.

Another key component is hybrid indexing. Metadata indexes support exact filters, text indexes support lexical matching, vector indexes support semantic recall, and graph indexes support lineage and relationship queries. This matters because scientific questions often mix exact and fuzzy criteria, and no single index type performs well across all of them.

The pattern also needs provenance-aware ingestion. During ingestion, the system should capture upstream dependencies, run identifiers, creators, timestamps, versions, and policy metadata. This matters because many retrieval tasks are really provenance tasks in disguise: scientists want not just a similar artifact, but the artifact that produced this result or the latest approved derivative of this dataset.

A ranking and explanation layer is equally important. Retrieved results should include enough evidence for users to understand why they were returned, such as matched fields, similar passages, graph paths, or policy constraints. This matters because opaque retrieval reduces trust and makes debugging impossible.

Finally, mature systems expose retrieval through stable APIs for both humans and agents. In Quark to Cosmos (Q2C)'s cosmological search, the user does not simply search for a file; they ask a high-level scientific question about cluster-scale objects in the Frontier-E simulations, and the AI agent turns that request into authenticated data access, workflow execution, HPC computation, and retrieval of petabyte-scale HACC/MACosmo simulation results. The plain-language query blends exact constraints, such as simulation selection, timestep, and mass threshold, with more interpretive goals, such as distinguishing relaxed and unrelaxed halos or identifying cool-core structure. The returned results include both visual and quantitative evidence for the classification. This example shows that advanced AI data search should be designed around governed, provenance-aware workflows: the system must resolve semantic intent, apply strict metadata constraints, execute or locate the appropriate computational workflow, and return results with enough context for both

human interpretation and downstream machine use. This allows notebooks, orchestration systems, and AI assistants to use the same governed search interface rather than bypassing it with ad hoc scripts.

## 5.3 Design Tradeoffs

*Retrieval richness vs. system complexity* – Combining graph, vector, relational, and text retrieval improves usefulness, but increases platform complexity. Teams should add modes only when they answer real user queries that the existing stack cannot handle. A small well-integrated hybrid system is better than a sprawling platform no one can maintain.

*Provenance fidelity vs. ingestion overhead* – Rich metadata and lineage improve trust, but require disciplined ingestion and wrappers for legacy tools. If the team cannot instrument everything immediately, it should prioritize high-value artifacts and canonical workflows first rather than waiting for perfect coverage.

*Semantic flexibility vs. explainability* – Vector retrieval can surface relevant results despite vocabulary mismatch, but is often harder to justify than exact filters or graph queries. In practice, semantic retrieval is most effective when paired with visible metadata filters and explanatory snippets so users can see why an item was ranked highly.

*Federated access vs. retrieval quality* – Distributed search respects sovereignty, but can weaken ranking and increase latency. Scientists should decide whether the corpus requires strong local control or whether some metadata can be centralized safely for better discovery. Often a middle ground works: federated payloads with centralized searchable summaries and policy-aware brokers.

*Security precision vs. usability* – Fine-grained access control reduces leakage, but can fragment the search experience. The right landing point is usually policy-aware retrieval in which users see only what they are entitled to see, while the system still presents coherent rankings and clear explanations for missing or partially visible results.

## 5.4 Implementation Advice

- Use hybrid retrieval rather than a single search mode.
  Combine exact metadata filtering, full-text search, semantic similarity, and graph traversal in one query pipeline. A practical implementation often applies hard filters first, then semantic or lexical ranking, then graph-based expansion or provenance checks. This approach reflects how real scientific questions are asked and avoids the common failure modes of purely keyword or purely embedding-based systems.
- Make provenance first-class in the index.
  Index not only the artifact content but also where it came from, what created it, which version it belongs to, and what depends on it. Provenance fields and lineage edges let users answer questions about derivation and trust, not just topical relevance. This is especially important for generated artifacts, model outputs, and workflow products that have little standalone meaning without context.
- Normalize artifacts into a common retrieval contract.
  Define a shared schema across documents, datasets, runs, models, and derived outputs. At minimum, include stable IDs, artifact type, ownership or access metadata, timestamps, version identifiers, and payload pointers. Normalization reduces query

fragmentation and makes it feasible to build shared retrieval APIs instead of one-off adapters for every source.

- Separate heavy binary payloads from searchable metadata.
  Large arrays, images, checkpoints, and other binaries often do not belong directly in the search index. Store compact summaries, embeddings, previews, extracted metadata, and lineage links in the index, while keeping the payload in object storage or domain-specific repositories. This keeps indexing and query latency manageable while preserving discoverability.
- Expose retrieval through stable APIs for both humans and agents.
  Provide one governed interface that notebooks, dashboards, batch workflows, and AI assistants can all call. Stable APIs make access control, logging, ranking behavior, and version handling consistent across clients. They also reduce the temptation for teams to build ad hoc search scripts that bypass policy and provenance capture.
- Evaluate retrieval with representative golden queries and relevance judgments.
  Search quality should be measured against real user questions, not only generic IR metrics. Build a small benchmark set of representative queries, expected results, and graded relevance labels drawn from actual scientific tasks. Then compare ranking changes, filter behavior, latency, and policy correctness as the system evolves. This is the only reliable way to know whether retrieval is improving for scientists rather than merely changing.

**Questions for Future Expansion**

Which artifact types are most important to support first in this repository context: documents, datasets, workflow runs, model outputs, or provenance records?

What concrete examples from Prometheus, MOAT, FAMOUS, or CM2US would best illustrate hybrid retrieval and lineage-aware search?

Which metadata fields are already captured consistently enough to form a shared retrieval contract?

What access-control model governs cross-team search, and how should partial visibility be represented in ranked results?

What golden queries from real scientists should be used to evaluate retrieval quality and trustworthiness?

**Figure Suggestions**

Primary figure: Hybrid retrieval architecture diagram showing one scientific query flowing through metadata filtering, full-text search, semantic retrieval, and provenance or graph traversal before results are merged into a ranked, explained, policy-aware output.

Optional figure: Artifact lineage search example showing dataset version X, downstream runs, validation records, and related notes connected in a provenance graph, with the queried path highlighted.

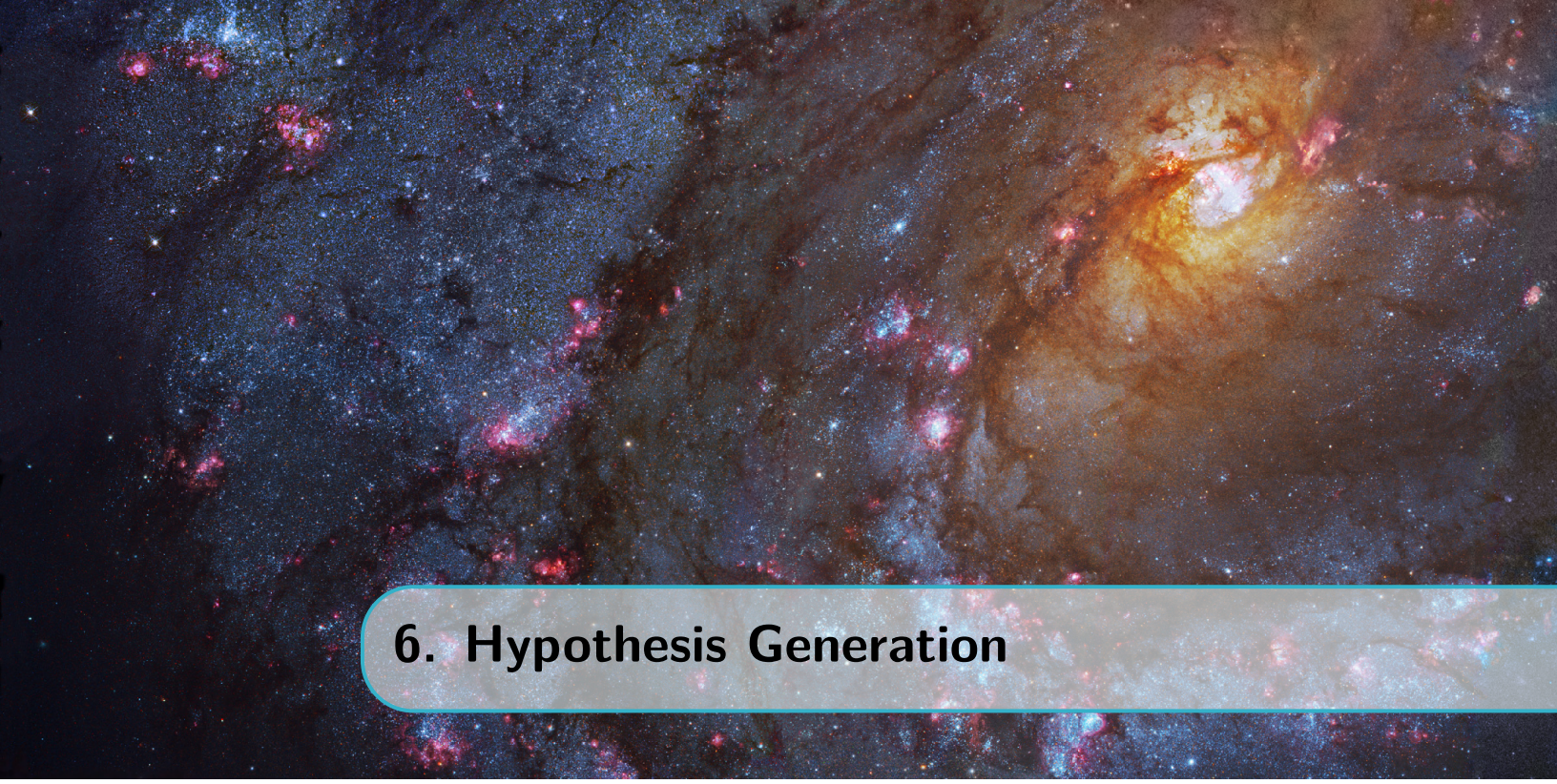
## 5.5 Relationships to Other Patterns

- Use with *Institutional Memory and Knowledge Capture* when stored knowledge must be made usable for current work.
- Use with *Hypothesis Generation* when retrieval should feed idea formation rather than stop at summarization. This pairing matters when search is not just collecting background material, but assembling the specific evidence, analogies, and prior attempts that make a new hypothesis more plausible or more testable.
- Use with *Data Pipelines and Collection Patterns* when retrieval quality depends on metadata, manifests, lineage, and structured artifacts created upstream.

# Part III

# Reasoning and Discovery Patterns

Reasoning and discovery patterns describe how AI contributes directly to scientific thinking, exploration, and model construction. These chapters cover workflows in which AI helps generate hypotheses, accelerate optimization, encode scientific structure, or assist with specialized engineering tasks that support discovery. Grouping them together emphasizes that these patterns are primarily about improving the quality, speed, and scope of scientific reasoning rather than about infrastructure alone.



# 6. Hypothesis Generation

## 6.1 Problem

Many scientific programs face a mismatch between the size of the idea space and the amount of human attention available to search it. Researchers may be choosing among candidate materials, mechanisms, control strategies, parameter regimes, experimental conditions, or theoretical explanations, while also deciding which limited subset deserves simulation time, instrument time, fabrication effort, or expert review. The AI-specific problem is not simply generating more ideas; it is generating candidate hypotheses or designs that are both scientifically meaningful and prioritizable under uncertainty. In other words, teams need help searching a large conceptual space without confusing novelty with usefulness.

Naive solutions often fail in one of two directions. Pure manual brainstorming preserves scientific judgment but is too slow and inconsistent for large combinatorial spaces. It also tends to revisit familiar regions because humans reason from known mechanisms and established literature. At the other extreme, unconstrained AI generation can flood the team with superficially plausible suggestions that are redundant, physically impossible, unsupported by available data, or misaligned with actual experimental constraints. Ranking candidates by a single predicted score is also usually inadequate: the highest-scoring option may be uninteresting, overfit to the model, too uncertain to trust, or infeasible to test. For teams such as FAMOUS, Axess, Fusion, MAIQMag, and Prometheus, the challenge is to make AI a disciplined collaborator in scientific exploration rather than a prolific source of ungrounded suggestions.

The difficulty is compounded by feedback delays. In many domains, validating a hypothesis requires expensive experiments, long-running simulations, or scarce expert interpretation. That means the system must decide what to test before it can know whether its generative assumptions were sensible. If the loop is badly designed, the AI quickly chases artifacts of its own scoring model, narrows into local optima, or overwhelms the lab with low-value proposals. The right abstraction is therefore a closed-loop discovery process: generate, score, select, test, learn, and repeat with explicit treatment of uncertainty, novelty,

and constraint satisfaction.

This pattern generalizes because the same logic appears across discovery workflows even when the subject matter changes. Whether the candidates are molecules, control policies, experiment plans, diagnostic leads, or explanatory mechanisms, the team still needs a method for expanding the search space and allocating limited validation resources wisely.

## 6.2 Characteristics of the Solution

The shared pattern uses AI to generate candidate hypotheses, experiments, or designs; a predictive model or surrogate scores them for value, uncertainty, or feasibility; selected candidates are tested; and the results are fed back into the generator or selector in a closed loop.

A useful system usually separates candidate generation from candidate evaluation. The generator can be optimized for breadth, creativity, or structured recombination, while the evaluator can be optimized for plausibility and decision quality. This separation matters because the same model is rarely best at both proposing ideas and judging them.

The pattern also needs an explicit representation of constraints. These may encode physical laws, fabrication limits, instrument ranges, budget limits, safety rules, or domain heuristics. This matters because unconstrained generation wastes scarce validation capacity on impossible or irrelevant candidates.

Uncertainty estimation is another common component. Whether implemented through ensembles, Bayesian methods, calibrated surrogates, or heuristic confidence measures, uncertainty helps the system avoid overcommitting to predictions in poorly understood regions. This matters because the best next test is often not the one with the highest expected score, but the one that most improves knowledge or reduces decision risk.

A feedback store for outcomes is equally important. The team should capture not only successes but also failed hypotheses, ambiguous results, and negative findings in structured form. This matters because learning from failure is often what prevents repeated waste and gradually reshapes the search policy.

Finally, practical deployments include human review at high-impact decision points. Scientists supply mechanism-level judgment, sanity checks, and prioritization criteria that are still hard to encode fully. The human role matters most when tests are expensive, irreversible, or scientifically consequential.

## 6.3 Design Tradeoffs

*Exploration vs. exploitation* – Broad exploration may find novel opportunities, while stronger exploitation uses resources more efficiently but risks local optima. Teams with little prior data should usually bias toward exploration early and then gradually tighten selection as confidence improves.

*Model richness vs. iteration speed* – Richer generators and evaluators may propose better candidates, but slow the loop. Scientists should be honest about the bottleneck: if experiments take months, a more sophisticated selector may be justified; if the loop is already fast, simpler models may yield more total learning per week.

*Autonomy vs. expert oversight* – More automation increases throughput, while expert review remains important for expensive or irreversible steps. A good landing point is often

machine-generated longlists with human-approved shortlists, especially in domains where mechanism knowledge is strong but search space breadth is overwhelming.

*Uncertainty-aware selection vs. simplicity* – Confidence estimates improve prioritization, but complicate the system. If the team can only implement one advanced feature beyond scoring, uncertainty handling is often the most valuable because it directly informs what to test next and where the model is likely to be misleading.

*Constraint enforcement vs. novelty* – Strong constraints improve plausibility, but can suppress surprising and valuable hypotheses. The practical choice is rarely all-or-nothing. Hard constraints should capture inviolate rules such as physics, safety, or fabrication impossibility, while softer preferences can remain adjustable to leave room for discovery.

## 6.4 Implementation Advice

- Separate generation from evaluation.
  Use one component to create candidate hypotheses or designs and another to rank or filter them. This reduces self-reinforcing error, because a model that is good at producing plausible language or structures is not automatically good at deciding scientific value. In practice, the evaluator may be a surrogate model, a rules engine, a simulator, a literature-backed scoring rubric, or a human review panel.
- Keep humans at high-impact decision points.
  Human review is most valuable when tests are expensive, hazardous, irreversible, or likely to define future research direction. A practical workflow is to let AI produce and preliminarily score many candidates, then ask scientists to inspect a narrower set with explanations, uncertainty indicators, and relevant evidence. This preserves throughput while preventing blind trust in generated proposals.
- Track uncertainty and novelty explicitly.
  Do not rank candidates only by predicted reward or expected performance. Include novelty measures such as distance from existing data, literature coverage gaps, structural diversity, or representation-space separation, alongside uncertainty estimates from ensembles, Bayesian surrogates, or calibration diagnostics. These signals help the team avoid repeatedly selecting minor variations of already-known good ideas.
- Capture outcomes from both successes and failures in structured form.
  Negative results are essential training data for future selection policies. Store candidate definitions, predicted scores, confidence estimates, actual outcomes, assay or simulation conditions, and adjudication notes in a machine-readable schema. This allows later models to learn not just what worked, but what looked promising and failed under real validation.
- Constrain proposals with domain knowledge.
  Encode hard limits and scientific priors directly into the generation or filtering process. Useful methods include grammar-constrained generation, rule-based rejection, physically informed parameter bounds, ontology-guided proposal templates, and simulator-backed checks. Constraints reduce wasted validation capacity and improve trust in the resulting suggestions.
- Use adaptive batching to balance throughput and learning.
  When multiple experiments or simulations can run in parallel, choose batches that mix high-confidence candidates with uncertain or diverse ones. Pure top-k selection often

collapses diversity and reduces learning value. Adaptive batching, batch Bayesian optimization, or diversity-aware selection heuristics usually produce better long-term search efficiency than repeatedly picking only the current best estimate.

**Questions for Future Expansion**
Which concrete FAMOUS, Axess, Fusion, MAIQMag, or Prometheus workflows should be cited as examples of closed-loop hypothesis generation?
What kinds of hypotheses are most common in scope here: materials, controls, diagnostics, mechanism explanations, or experiment plans?
Which uncertainty methods are already practical in the target environment: ensembles, Bayesian surrogates, conformal methods, or simpler heuristics?
Where are the hard domain constraints currently encoded, and which ones are still handled informally by experts?
What structured schema should be recommended for storing failed and ambiguous hypotheses as reusable training signal?

**Figure Suggestions**
Primary figure: Closed-loop discovery workflow showing candidate generation, scoring for value and uncertainty and feasibility, batch selection, experiment or simulation, outcome capture, and model update.
Optional figure: Candidate selection frontier plotting novelty versus predicted value versus uncertainty, with the chosen batch highlighted.

## 6.5 Relationships to Other Patterns

- Use with *Advanced AI Data Search* when the immediate need is finding relevant evidence, prior work, and analogies that can ground or sharpen candidate ideas.
- Use with *Institutional Memory and Knowledge Capture* when prior results, failures, and context should shape which hypotheses are worth pursuing.
- Use with *Surrogates for Optimization* when fast models help search large design spaces for promising candidates.
- Use with *Physics-Informed AI Patterns* when domain structure can filter implausible ideas early.

# 7. Surrogates for Optimization

## 7.1 Problem

Many scientific design and control problems require searching a large space of candidate configurations while evaluating each candidate with a high-fidelity simulator, experiment, or coupled multiphysics workflow. The bottleneck is that the true evaluator is often too expensive to call often enough for optimization to progress efficiently. If one simulation takes minutes, hours, or days, then even a sophisticated search strategy cannot explore enough of the design space to discover robust optima, characterize uncertainty, or adapt to changing objectives. Scientists are forced to choose between optimizing too little, narrowing the search prematurely, or replacing scientific ambition with engineering convenience.

Naive alternatives usually underperform. Brute-force search and dense parameter sweeps waste budget on unpromising regions. Gradient-free black-box optimization without acceleration often spends too many expensive queries learning coarse structure. Simplistic response surfaces built on too little data can be fast but dangerously misleading, especially in highly nonlinear regimes, near discontinuities, or outside the sampled region. A particularly AI-specific failure mode appears when teams optimize the surrogate itself rather than the real objective: the optimizer exploits approximation errors, finds adversarial corners of the learned model, and proposes designs that score well in prediction but fail under high-fidelity evaluation.

The real challenge, then, is not just to build a fast emulator, but to embed it into a disciplined decision loop that preserves scientific trust. This is naturally relevant for Axess and related optimization-heavy work in Fusion and Combustion and Flows, but the pattern generalizes to any setting where candidate generation is cheap and authoritative evaluation is expensive. A successful surrogate workflow accelerates search while explicitly managing uncertainty, domain validity, and the cost of occasional return trips to the expensive ground truth.

## 7.2 Characteristics of the Solution

The central component is a *learned surrogate model* trained to approximate the expensive evaluator over a scientifically meaningful input space. This matters because it converts a sparse set of costly evaluations into a cheap predictive oracle that can be queried inside an optimizer many times.

A second essential component is *careful training data selection.* Surrogates are only as useful as the regions they cover. Training data usually combines historical runs, designed experiments, and targeted sampling near important operating regimes. Good coverage matters because optimization quickly exposes weak spots in the surrogate.

Third, effective systems incorporate *uncertainty estimation or trust indicators.* These may come from ensembles, Bayesian methods, conformal intervals, distance-to-training-data heuristics, or disagreement among models. Uncertainty matters because it helps the optimizer decide when to exploit the surrogate and when to query the high-fidelity evaluator.

Fourth, mature workflows retain *high-fidelity evaluation in the loop.* Promising, novel, or high-uncertainty candidates are periodically checked with the expensive source of truth. Those results are then added back into the training set. This prevents drift toward unreal surrogate optima.

Fifth, strong implementations align the surrogate with *domain constraints and optimization objectives.* The model may predict multiple outputs, constraints, and failure probabilities, not just a single scalar objective. This matters because real optimization problems are usually constrained and multi-objective.

Finally, the workflow typically includes *versioning and retraining.* As new evaluations arrive or the optimization regime shifts, the surrogate must be updated intentionally so teams can compare behavior across model generations and reproduce prior decisions.

## 7.3 Design Tradeoffs

*Fidelity vs. speed* – Rich neural surrogates or structured probabilistic models may capture subtle nonlinear behavior, but they can increase inference cost and tuning complexity. For inner-loop optimization, a slightly less accurate surrogate that is dramatically faster may create more end-to-end value than a very accurate model that slows the search.

*Data cost vs. design-space coverage* – High-fidelity labels are expensive, so teams are tempted to train on sparse data. The risk is that optimization will drive immediately into under-sampled regions. Active learning, adaptive design of experiments, and uncertainty-guided acquisition are often the right middle ground when label cost dominates.

*Physical constraints vs. flexibility* – Physics-aware surrogates can improve plausibility and extrapolation, but over-constraining the model may suppress effects that matter in practice. The right choice depends on how trusted the governing assumptions are and whether the workflow is seeking incremental improvement or potentially surprising designs.

*Single-task vs. multi-task surrogates* – Separate models simplify debugging and allow objective-specific tuning, while shared models can exploit correlation among outputs such as performance, cost, and feasibility. The danger of multi-task learning is negative transfer when outputs behave differently across the design space.

*Static vs. continuously updated surrogates* – Frozen surrogates make studies reproducible and easier to compare, while online updates keep the model aligned with new data and

optimizer behavior. A useful compromise is to retrain in explicit batches or campaign phases rather than after every new evaluation.

## 7.4 Implementation Advice

- Train on diverse and physically meaningful data.
  Do not rely only on historical data concentrated in previously explored regions. Combine legacy runs with designed experiments, boundary cases, and representative failure regimes so the surrogate learns the geometry of the design space rather than a narrow corridor through it. Space-filling designs, active learning, and stratification by operating regime are often more valuable than simply adding more of the easiest samples.
- Quantify uncertainty and use it in optimization decisions.
  A surrogate without uncertainty encourages overconfident exploitation. Use ensembles, Gaussian-process style posteriors where feasible, dropout-based approximations, conformal methods, or novelty heuristics to estimate trust. Then incorporate those signals into acquisition functions, feasibility filters, or escalation thresholds for high-fidelity re-evaluation.
- Keep high-fidelity evaluation in the loop.
  Do not let the optimizer run exclusively against the surrogate for long stretches. Periodically validate promising candidates, uncertain candidates, and diversity-seeking candidates with the authoritative simulator or experiment. This limits reward hacking against surrogate error and supplies the new labels needed to keep the model honest.
- Encode domain knowledge where possible.
  If known symmetries, conservation laws, monotonic relationships, or feasibility constraints exist, incorporate them into features, architectures, losses, or post-processing. Even lightweight domain knowledge can materially reduce sample requirements and improve out-of-sample behavior, especially in sparse-data scientific regimes.
- Co-design the surrogate and the optimizer.
  The optimizer should consume the outputs the surrogate can provide well. If the workflow needs constraint handling, uncertainty-aware acquisition, or multi-objective ranking, design the surrogate to predict those quantities explicitly rather than hoping they can be recovered after the fact. Joint design reduces friction and helps the overall loop behave predictably.
- Retrain and version the surrogate as new data arrives.
  Store training sets, preprocessing logic, model configurations, and evaluation summaries for each surrogate generation. Versioning makes it possible to compare campaigns fairly, reproduce prior optimization decisions, and diagnose whether a behavior change came from the optimizer, the data, or the surrogate itself.

**CM2US example: uncertainty-gated atomistic surrogates.**

In CM2US, HydraGNN was used as a machine-learned interatomic potential surrogate inside the `matsim-agents` workflow [Lup26a; Lup26b] to accelerate atomistic relaxation and screening before invoking higher-fidelity density functional theory (DFT). HydraGNN is documented in its v5.0 user manual and official v5.0 OSTI software release [Lup26c; Oak26]. The workflow was designed not only as a local prototype, but as a portable scientific workflow

for DOE leadership-class supercomputers, including NERSC Perlmutter, ALCF Aurora, and OLCF Frontier. The HydraGNN model used in this workflow follows a multi-branch architecture: shared message-passing layers learn transferable atomistic representations, while multiple output decoding branches specialize in data coming from different sources, datasets, or fidelity levels. At inference time, the BranchWeightMLP router maps the chemical composition of a candidate structure to a set of softmax weights over these output branches. These weights are used to combine the energy and force predictions from the different branches into a single prediction returned by the model.

The key workflow design choice in CM2US was not to treat this blended HydraGNN prediction as an unconditional replacement for DFT. Instead, `matsim-agents` uses the routing distribution as a low-cost confidence heuristic. When the router assigns most of the weight to one branch, the prediction is treated as sharply routed and can proceed to analysis. When the routing is diffuse across several branches, the workflow interprets this as a sign that the model is blending information from multiple training sources and may require additional validation. Those cases can be escalated to DFT labeling and active-learning updates. The solution was therefore to make surrogate confidence an explicit workflow-control variable: HydraGNN accelerates routine cases, while DFT remains available whenever the workflow detects that the surrogate may be operating outside a clearly routed regime.

**Questions for Future Expansion**
Which Axess or Fusion workflows currently place surrogates inside optimization loops, and what are the expensive ground-truth evaluators?
Are uncertainty estimates already used operationally, or only reported offline?
What kinds of acquisition functions or optimizers have proven effective in these settings?
Where have surrogates failed because the optimizer exploited approximation error, and how was that detected?
Would the report benefit from a short comparison among Gaussian-process, neural, and hybrid physics-aware surrogates for different data regimes?

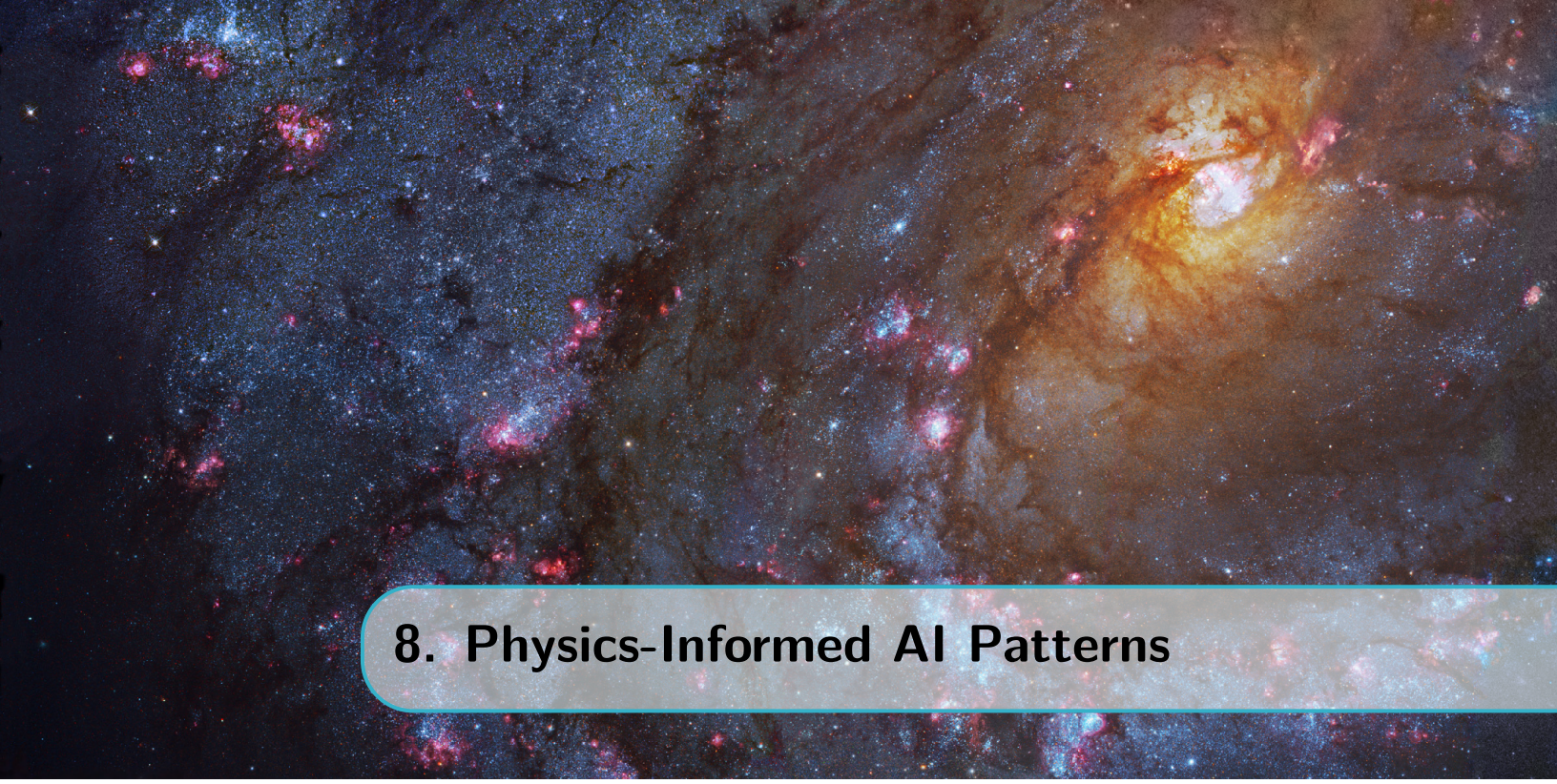
**Figure Suggestions**
Primary figure: Surrogate-assisted optimization loop showing expensive evaluations, surrogate training, optimizer proposals, uncertainty checks, selective high-fidelity validation, and retraining.
Optional figure: Surrogate exploitation failure sketch comparing the surrogate-predicted optimum with the true objective surface to show reward hacking against model error.

## 7.5 Relationships to Other Patterns

- Use with *Physics-Informed AI Patterns* when fast approximators need domain constraints to remain scientifically credible.
- Use with *At-Scale AI Training and Inference* when surrogate quality depends on substantial training infrastructure.
- Use with *Hypothesis Generation* when surrogates help explore large design spaces and

surface promising candidates.

- Use with *Workflow-Level Evaluation and AI Advantage* when the real question is whether the surrogate improves downstream decisions rather than only predictive fit.



# 8. Physics-Informed AI Patterns

## 8.1 Problem

Purely data-driven AI can fit observational or simulated data well while still violating the scientific structure that gives predictions meaning. In many domains, acceptable outputs must satisfy governing equations, conservation laws, topology, symmetry, boundary conditions, feasibility constraints, or other domain rules. A model that ignores those rules may interpolate smoothly in-distribution yet produce scientifically invalid outputs when asked to extrapolate, optimize, or operate in edge cases. In scientific workflows, these errors are not cosmetic. They can corrupt downstream simulations, suggest impossible designs, mislead interpretation, or create unsafe recommendations.

Naive solutions often assume that enough data will cause the model to learn the right invariants implicitly. Sometimes that works in narrow regimes, but it is brittle. Scientific datasets are often expensive, sparse, biased toward typical operating conditions, and incomplete with respect to rare but important boundary cases. Even strong models may learn shortcuts that mimic the training distribution without honoring the underlying physics. Another naive response is to apply domain checks only at the end, after training a fully unconstrained model. This can detect violations, but it does not necessarily prevent the model from using physically implausible internal representations or from wasting capacity on non-physical degrees of freedom.

The pattern is therefore not merely use physics in the loss. It is about inserting trusted domain knowledge across the workflow so that data curation, model structure, training objectives, and evaluation all reinforce scientific validity. This is naturally relevant to teams such as GRID, Synapse-I, Axess, and MAIQMag, where governing structure likely matters as much as predictive accuracy. The general lesson applies broadly: when the science already provides constraints, invariants, or simulators, AI systems should exploit them explicitly rather than relearning them weakly from data alone.

Physics-Informed AI is a strong fit for MAIQMag as the workflow centers on inferring spin Hamiltonians and their parameters from multimodal experimental and synthetic data

using physics-based forward solvers. Physics is therefore not an auxiliary regularizer: the forward model defines how a candidate Hamiltonian produces predicted observables, constrains which candidate models are meaningful, and provides an authoritative check on the inverse inference. A representative MAIQMag loop combines candidate Hamiltonians or parameter values, forward-solver predictions, comparison across experimental modalities, and uncertainty-aware updating. This hybrid structure allows AI to accelerate inference without severing the result from the physical model that gives the inferred Hamiltonian scientific meaning.

The Grid team's physics-informed approach centers on Lagrangian-based training objectives rather than standard mean squared error (MSE) supervision alone, since MSE only treats the AC power flow equations and operational limits as implicit patterns—small prediction errors can still push solutions across constraint boundaries and render them physically infeasible. Three objectives were benchmarked (plain MSE, violation-based Lagrangian, and augmented Lagrangian), and augmented Lagrangian (AL) was selected as the main backbone. AL incorporates the equality constraints (power balance) and inequality constraints (generation, voltage, and line-flow limits) directly into the loss via dual variables and a quadratic penalty, with dual variables updated periodically during training. It consistently gave the lowest constraint violations, especially under topology shift, while keeping accuracy comparable to MSE. A few practical notes for other teams: normalize constraint residuals across physical units, use an adaptive penalty schedule, and monitor dual variable convergence—AL is more hyperparameter-sensitive than MSE, so its penalty schedule and dual step sizes were tuned with Bayesian Hyperparameter Optimization (HPO).

Beyond the training loss, the team evaluates models with physics-specific metrics rather than accuracy alone. Since Alternating Current Optimal Power Flow (ACOPF) is NP-hard, accuracy against labels is not sufficient on its own—models must be assessed through the constraints of the underlying optimization problem. Two complementary criteria are tracked: predictive accuracy (MSE against solved ACOPF labels) and physical feasibility, where violations are aggregated per constraint family and normalized by the square root of the topology size for comparability across grid sizes. Stress tests further target the regimes where models break: high-load / near-capacity points and high-degree buses (node error correlates with node degree, r 0.51). The recommendation worth passing on is to report tail statistics (e.g., 95th/99th percentile violations) stratified by operating regime rather than means, and to use these diagnostics operationally—flagging high-load scenarios and high-degree nodes for solver verification, and oversampling hard cases during training.

## 8.2 Characteristics of the Solution

The shared solution inserts domain knowledge throughout the workflow rather than at a single point.

One common component is *physics-aware ingestion and preprocessing*. Raw data is normalized using correct units, coordinate frames, boundary conditions, and domain-specific filtering rules. This matters because a model cannot preserve structure that the data pipeline has already distorted.

A second component is the use of *solver-backed labels or derived quantities*. High-fidelity simulations, inverse solvers, or trusted analytic transformations can provide targets, constraints, or residuals that teach the model more than raw observations alone. These

labels matter because they encode authoritative structure even when data is limited.

Third, teams often employ *architectures with built-in inductive bias*. Examples include topology-aware representations, symmetry-respecting layers, graph-based formulations, or parameterizations that make some violations impossible by construction. Architecture-level bias matters because it reduces the search space and improves generalization.

Fourth, strong workflows use *hard or soft constraints during training*. Hard constraints enforce feasibility exactly where possible, while soft penalties encourage compliance when exact enforcement is impractical. This matters because different constraints carry different levels of scientific non-negotiability.

Fifth, mature implementations combine ordinary predictive evaluation with *physics-consistency audits*. Accuracy metrics alone can hide models that look good numerically but fail scientifically. Audit metrics based on residuals, conservation errors, invariant violations, or feasibility rates make the real tradeoffs visible.

Finally, robust systems include *feedback loops from audit failures*. Repeated violation patterns should trigger targeted data collection, architecture changes, stronger constraints, or fallback to hybrid solver-based inference in the affected regimes.

## 8.3 Design Tradeoffs

*Generality vs. specificity* – Physics-aware workflows are more trustworthy within the intended scientific regime, but they are often less portable because they depend on domain-specific assumptions, operators, or simulators. Teams should decide whether they are optimizing for broad reuse or for deep trust in a narrower domain.

*Hard constraints vs. learning flexibility* – Hard constraints eliminate certain classes of invalid output, but they can complicate optimization and sometimes overconstrain the model when the scientific picture is approximate or incomplete. Soft penalties allow more flexibility, but violations may persist unless the penalties are well tuned and carefully monitored.

*Fidelity vs. computational cost* – High-fidelity solvers improve supervision and auditing, but they can dominate cost. A practical compromise is often to use expensive physics selectively: for label generation in key regimes, for post hoc audit on sampled cases, or for fallback on high-risk predictions.

*Interpretability vs. expressiveness* – Physics-informed models can be easier to justify scientifically, while larger black-box models may fit complex data patterns better. Scientists should be cautious about giving up structure for a small gain in benchmark accuracy if the resulting model is harder to trust under extrapolation.

*Reproducibility vs. operational overhead* – Versioning solvers, mesh settings, boundary assumptions, and audit logic improves repeatability, but it adds engineering burden. This burden is usually justified once physics-based supervision influences scientific conclusions or operational decisions.

## 8.4 Implementation Advice

- Encode constraints at multiple workflow stages.
  Do not rely on a single penalty term late in training to enforce all scientific structure. Apply domain knowledge during data preparation, feature construction, model architecture, loss design, and post-training validation. Multi-stage encoding is more robust

because it reduces the number of ways the system can drift into non-physical behavior.

- Track both predictive and physics-consistency metrics.
  Report ordinary error metrics alongside residual norms, conservation errors, invariant violations, feasibility rates, or topology-specific checks. This prevents teams from shipping models that improve headline accuracy while degrading scientific validity. Dashboards and experiment logs should make both classes of metrics first-class citizens.
- Prefer hybrid AI plus solver designs for critical workflows.
  When consequences are high, use AI to accelerate or initialize a trusted solver rather than to replace it outright. Hybrid designs can deliver much of the speed benefit while preserving a principled path back to physical consistency. They are especially attractive when exact compliance matters more than raw throughput.
- Version all physics dependencies and audit logic.
  The scientific meaning of a model depends not only on training data and weights but also on solver versions, discretization settings, boundary assumptions, constants, and audit code. Record these explicitly so later users can reproduce results and understand whether behavior changed because of the model or because the underlying physics toolchain changed.
- Use targeted feedback loops when audits expose repeated violations.
  If the model consistently breaks a conservation law in a certain regime or fails near a boundary condition, treat that as a design signal. Add data in that regime, revise the architecture, strengthen the constraint, or route those cases to a hybrid solver path. Audit failures are most useful when they drive concrete workflow changes.
- Choose hard versus soft constraints intentionally.
  Classify constraints by their scientific role. Non-negotiable feasibility or safety conditions should usually be enforced hard when possible. Approximate empirical regularities may be better represented as soft penalties. Making this distinction explicit helps teams avoid both under-constraining and over-constraining the model.

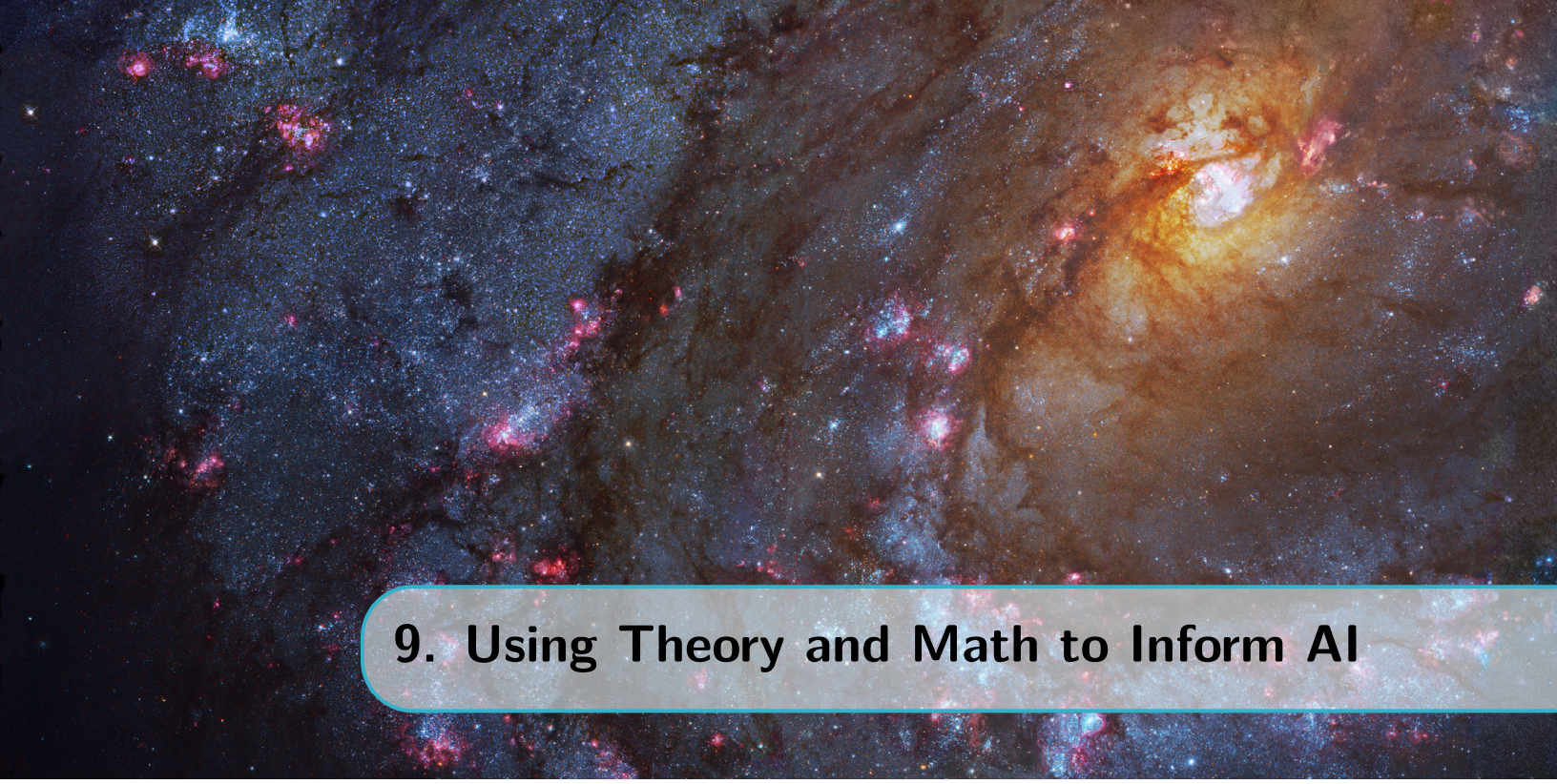
**Questions for Future Expansion**
Which teams have concrete examples of hard physical constraints versus soft penalty approaches?
What solver or audit tooling is currently used in GRID, Synapse-I, Axess, or MAIQMag workflows?
Are there documented cases where a purely data-driven model performed well on standard metrics but failed physics audits?
Which physics-consistency metrics are most actionable for scientists during development versus deployment?
Would a short taxonomy of architectures, losses, and hybrid solver patterns improve this chapter?

**Figure Suggestions**
Primary figure: Multi-stage physics-informed AI workflow showing where domain knowledge enters through preprocessing, solver-backed labels, inductive-bias architecture,

constrained loss, audits, and solver-based fallback.
Optional figure: Accuracy versus physics-consistency comparison plot showing how models differ on predictive error and invariant or conservation violations.

## 8.5 Relationships to Other Patterns

- Use with *Surrogates for Optimization* when fast models need physical constraints to stay useful in optimization or control loops.
- Use with *Cyber-Physical Systems* when real-world systems provide the domain signals, constraints, and operating regimes that make physical structure important.



# 9. Using Theory and Math to Inform AI

## 9.1 Problem

Scientific teams often ask AI to help with reasoning, design, approximation, control, inverse problems, and analysis in domains where applied mathematics, in particular computing and learning theories, already says something important about what is and is not achievable. In these settings, the main question is not only whether a model can be constrained to produce valid outputs. A prior question is whether the task itself is well posed for AI at all, with acceptable identifiability, conditioning, sample complexity, approximation error, and computational cost. Applied math enters here not as a decorative explanation layer, but as a way to estimate in advance where AI is likely to succeed, where it will be fragile, and what kind of guarantees or failure modes should be expected.

Naive solutions fail by treating theory as something to apply after model development. That mindset is close to constraint enforcement: train the model first, then add penalties, filters, or audits so the outputs look acceptable. But an earlier failure mode is often more important. Teams may use AI on inverse problems that are ill posed, on optimization problems whose surrogate is not accurate in the regimes that matter, on numerically unstable formulations, on tasks whose target is not identifiable from available data, or on open-ended design questions where the verification burden exceeds any likely savings. In those cases, a polished answer can still be incomplete, unsound, or operationally useless, in extreme case hallucinations that are hard to detect and mitigate. The problem is not only that the model violated known laws such as physics of a physical system or uncomputability of computing task. It is that applied mathematics could often have predicted the risk before the model was trusted.

The pattern is therefore to use theory as an advance decision tool for AI design and use. Theory helps decide whether the problem formulation is stable, what error cannot be eliminated, how uncertainty will propagate, whether decomposition is necessary, and when outputs require independent verification no matter how plausible they appear. This makes the pattern distinct from physics-informed AI. The emphasis is not primarily on forcing

the model to obey equations during training. It is on using applied mathematics to predict, before or alongside modeling, when AI is a sound instrument, when it is only a heuristic aid, and when another method should dominate.

## 9.2 Characteristics of the Solution

The shared solution uses applied mathematics to assess feasibility and expected reliability before treating AI output as scientifically meaningful.

A first component is *advance qualification of the task*. Teams ask whether the problem is well posed enough for AI to produce useful results: computable, learnable, identifiable, and sufficiently conditioned under available data and compute. This matters because some tasks should be reformulated, regularized, decomposed, or rejected before any model is trained.

A second component is *mathematical bounds on expected success*. Generalization theory, complexity arguments, approximation bounds, stability analysis, and prior results are used to estimate what guarantees are possible and which error regions must remain non-zero. This matters because it reframes AI output as a bounded numerical or statistical inference process rather than a generic source of answers.

A third component is *distinguishing supported regimes from extrapolative regimes*. Teams separate regimes where the target function, protocol class, or operating range is understood well enough for AI to interpolate from regimes where the model is being asked to extrapolate into poorly characterized territory. This matters because AI may be reliable for summarization, emulation, or local approximation while remaining unreliable for novel architecture, control, or inverse-design decisions.

A fourth component is *theory-guided decomposition of the problem*. When the full task is too broad or unstable, it is broken into subproblems with better conditioning, clearer error behavior, or tractable approximations. This matters because AI often becomes useful only after the applied math has identified the right unit of work.

A fifth component is *explicit treatment of incompleteness and unsoundness*. The workflow assumes that outputs may be partial, approximate, or structurally wrong even when fluent and numerically plausible. This matters because in this pattern failure is not just low benchmark accuracy; it is failure to meet the mathematical conditions of a satisfactory solution.

A sixth component is *independent verification where theory predicts risk*. Tasks with poor conditioning, unknown identifiability, open-ended extrapolation, or weak error guarantees trigger stronger simulation, residual analysis, formal checks, or expert review. This matters because theory should determine verification burden in advance rather than only after failures appear.

## 9.3 Design Tradeoffs

*Broad ambition vs. well-posed formulation* – A broader AI objective may sound more transformative, but narrowing the task to a mathematically tractable regime often yields more trustworthy progress. The cost is reduced scope; the benefit is fewer unsound conclusions.

*Open-ended generation vs. local approximation reliability* – AI is often strongest when approximating within a regime where the target behavior is smooth, stable, or sufficiently

sampled. Asking it to extrapolate into poorly characterized design space may create more apparent creativity but weaker scientific confidence.

*Empirical utility vs. advance guarantees* – Some workflows can tolerate heuristic performance with later review, while others need stronger ex ante justification for why AI should work at all. More mathematical analysis increases upfront effort but reduces false confidence.

*Single-pass answers vs. decomposed workflows* – A model asked to solve the whole problem in one step may appear efficient, but decomposition into better-conditioned subproblems usually improves interpretability, stability, and auditability.

*Fast approximations vs. error control* – Surrogates, reduced models, and learned emulators can dramatically accelerate workflows, but only if the approximation error is acceptable in the operating regime that matters. Speed gains that outrun error control can reduce overall scientific value.

*Known-regime trust vs. exploratory novelty* – Restricting AI to regions supported by mathematical analysis improves reliability, but may reduce the chance of discovering something genuinely new. Teams should decide explicitly when they want reliable support versus speculative exploration.

## 9.4 Implementation Advice

- Ask what applied math says about the task before asking the model for an answer.
  Check whether the problem is identifiable, stable, computable, plausibly learnable, and matched to available data and numerical tolerances. If the formulation is weak, redesign it before investing in model outputs.
- Use theory to choose the unit of approximation.
  Decide whether AI should approximate a full workflow, a reduced operator, a residual, a local map, a ranking function, or a subroutine. This choice is often the main determinant of whether the system is useful or brittle.
- Separate interpolation regimes from extrapolation regimes.
  Treat requests inside a characterized operating range differently from requests that move into new parameter regions, new geometries, or new design spaces. The latter should start with stronger skepticism and heavier validation.
- Use decomposition, regularization, and conditioning improvements first.
  If the end-to-end task is too open, unstable, or ill posed, apply mathematical reformulation before scaling model complexity. Better conditioning often improves AI reliability more than a larger model does.
- Make incompleteness and unsoundness first-class evaluation categories.
  Do not collapse all failure into low accuracy. Track whether an output omits essential terms, assumes unavailable mechanisms, fails sensitivity checks, or gives an answer that is numerically plausible but mathematically unjustified.
- Increase verification when theory predicts failure risk.
  Poor conditioning, weak identifiability, open-ended extrapolation, and loose error bounds should automatically require simulation, residual checks, uncertainty analysis, or expert review before adoption.
- Record the mathematical assumptions behind the workflow.
  Document what makes the task a reasonable AI target, what approximations are being

made, what error sources remain, and where the workflow transitions from supported inference to speculative generation.

**Questions for Future Expansion**
Which applied-math concepts should be named explicitly as recurring feasibility checks: conditioning, identifiability, approximation error, numerical stability, or uncertainty propagation?
Should this chapter include a compact taxonomy of advance-feasibility checks for AI workflows in optimization, inverse problems, and surrogate modeling?
Are there examples elsewhere in the report where reformulating the mathematics of the task mattered more than changing the model architecture?
Would it help to add a short sidebar on why incomplete and unsound outputs are different failure modes from ordinary prediction error?
Should the chapter include a checklist for deciding when AI output can be treated as advisory only versus decision-relevant?

**Figure Suggestions**
Primary figure: Advance-feasibility workflow showing task qualification, mathematical reformulation, decomposition into tractable subproblems, AI use, and verification burden based on predicted instability or error.
Optional figure: Matrix comparing task classes by conditioning, theory support, approximation regime, expected AI reliability, and required post-generation validation.

## 9.5 Relationships to Other Patterns

- Use with *Hypothesis Generation* when applied math is needed to decide which proposed hypotheses are even meaningful, identifiable, or testable.
- Use with *Surrogates for Optimization* when mathematical analysis must determine in advance whether a learned surrogate is likely to be accurate enough for optimization or control.
- Use with *Workflow-Level Evaluation and AI Advantage* when the team must show that mathematically qualified AI use improves the real workflow instead of only producing attractive model outputs.
- Use with *Evaluating Agentic AI Limits* when the question is how far autonomous or open-ended AI behavior can be trusted before additional structure, decomposition, or review is required.
- Use instead of *Physics-Informed AI Patterns* when the main role of theory is to predict feasibility, stability, and reliability in advance rather than to enforce physical constraints during model training or inference.

# 10. AI-Augmented Software Development

## 10.1 Problem

Scientific software development is constrained not only by ordinary programming effort but also by domain-specific complexity: legacy codebases, specialized hardware targets, brittle build environments, performance-sensitive kernels, evolving scientific requirements, and long-lived validation obligations. Long-lived scientific codes must be repeatedly translated, ported, and optimized for new accelerator architectures while preserving numerical and scientific validity. Engineers and researchers must modernize old implementations, refactor workflow glue, write tests for under-tested scientific logic, repair build and runtime failures, and preserve numerical behavior while making changes under schedule pressure. AI assistance is attractive because it can accelerate drafting, translation, explanation, and code transformation, but software in scientific settings is unusually unforgiving of shallow correctness.

Naive use of coding assistants often fails because generated code is plausible before it is trustworthy. A model can write syntactically correct code that changes numerical semantics, violates performance assumptions, drops edge-case handling, misunderstands scientific units, or invents APIs that do not exist. It may also produce tests that merely confirm its own mistaken interpretation of the task. In research software, these issues are amplified by sparse documentation, non-standard build systems, and domain knowledge that is encoded in conventions rather than explicit interfaces. The result is a dangerous asymmetry: AI can create changes faster than teams can safely understand and validate them.

The real problem, then, is how to use AI to reduce developer toil without weakening code quality, reproducibility, or governance. This is especially relevant for AI4HPC and Microelectronics where porting, optimization, and code generation are natural use cases, and for MOAT where review rigor and operational trust may be especially important. The general pattern applies whenever AI is asked to participate in software tasks, from exploratory scripts to production workflow code: treat the model as a fast but fallible collaborator whose outputs must be constrained by source context, automated verification,

and human judgment.

## 10.2 Characteristics of the Solution

The common solution treats AI as a *co-developer*: operationally, a fast transformation engine embedded inside a governed software workflow, rather than an autonomous coder. The workflow begins by supplying relevant *source context*: code snippets, interfaces, build constraints, style expectations, task intent, and sometimes examples of acceptable prior changes. Context matters because most coding failures are not syntax failures but misunderstandings of local conventions and hidden requirements.

A second component is *structured task framing*. Instead of asking for improve this code, teams ask for a focused transformation such as port this kernel, generate unit tests for this function, explain this failure, or refactor without changing public interfaces. Narrow tasks matter because they reduce ambiguity and simplify verification.

Third, robust workflows pair generation with *automated validation*. Candidate changes are compiled, linted, benchmarked, exercised by unit and regression tests, and sometimes compared against reference outputs. This is essential because software quality in scientific settings is determined by behavior, not fluency.

Fourth, mature implementations include *human review and governance*. Developers remain responsible for deciding whether the task framing was correct, whether generated tests are meaningful, and whether domain assumptions were preserved. Human judgment matters most where code changes can alter scientific conclusions, numerical stability, or system safety.

Fifth, successful teams curate a *domain-relevant corpus* of code, examples, and conventions. Whether used for retrieval, prompt examples, or fine-tuning, this corpus improves alignment with local APIs, naming, patterns, and scientific expectations.

Finally, the workflow usually includes *measurement*. Teams monitor acceptance rate, defect rate, review burden, latency, and use-case fit so the assistant is improved as the codebase and toolchain evolve.

## 10.3 Design Tradeoffs

*Model accuracy vs. trust* – A powerful model may produce impressive transformations, but if developers cannot reliably predict its failure modes, trust erodes quickly. Teams should favor workflows where verification is strong enough that imperfect generation is still net useful.

*Training breadth vs. domain depth* – Broad models are better at general coding tasks, while domain-adapted or retrieval-augmented systems better respect local idioms and scientific libraries. The right choice depends on whether the dominant pain points are common software chores or highly specialized scientific implementations.

*Generic coding capability vs. scientific acceptability* – Modern coding models may already generate plausible patches, refactorings, and translations for many programming tasks. In scientific workflows, however, the key question is not only whether AI can produce code, but whether the workflow can supply enough domain-specific context, structural code information, compiler and runtime feedback, numerical validation, performance evidence, and human governance to make generated changes scientifically acceptable.

*Automation overhead vs. payoff* – Building CI integration, benchmark harnesses, retrieval layers, and prompt libraries takes time. That investment pays off most for recurring tasks such as porting, repetitive refactors, test generation, and documentation support, less so for isolated tiny edits.

*Workflow integration vs. disruption* – Inline suggestions can speed individuals but also create noise and reviewer fatigue if too many low-value changes are proposed. Teams should place AI where it complements existing review and testing workflows rather than bypassing them.

*Explainability vs. speed* – Asking the assistant to justify changes or cite source context can improve review quality, but it adds latency and verbosity. This is usually worth it for non-trivial patches and less necessary for ephemeral developer assistance such as local code explanation.

## 10.4 Implementation Advice

- Start with narrow, high-confidence use cases.
  The fastest route to value is to use AI on tasks with clear success criteria: boilerplate test generation, code explanation, API migration, repetitive refactors, documentation drafting, or small helper functions. These are easier to review than open-ended architecture work and help teams learn the assistant's strengths and failure modes before expanding scope.
- Make verification mandatory and layered.
  Generated code should be checked through progressively stronger validators: syntax and schema checks, compilation and linking, runtime execution, unit and regression tests, reference-output or checksum comparison, numerical-tolerance checks, performance benchmarks, profiler signals, and domain-expert review where scientific semantics may have changed. In practice, this means integrating assistant-produced patches with existing continuous integration (CI) systems and refusing to treat model confidence or natural-language rationale as evidence of correctness.
- Keep humans in governance roles for critical changes.
  Developers and domain experts should approve changes that affect numerical kernels, scientific assumptions, security boundaries, deployment behavior, or interfaces used broadly across the codebase. The assistant can draft and iterate, but accountability should remain with humans who understand the scientific and operational context.
- Build a high-quality domain corpus.
  Collect representative internal code, accepted patterns, interface examples, style guidance, performance harnesses, and common failure cases. This material can support retrieval-augmented prompting, prompt templates, and future model adaptation. A small, clean corpus of local truth is often more valuable than a large undifferentiated code dump.
- Standardize prompts and task framing for repeated tasks.
  Teams get more reliable results when recurring requests follow stable templates: state the goal, constraints, files in scope, interfaces that must not change, performance expectations, and required tests. Standard framing reduces ambiguity and produces outputs that are easier to compare and review across users.
- Monitor assistant performance as the codebase evolves.

Track acceptance rates, reverted changes, review time, defect escapes, benchmark regressions, and the classes of tasks where AI helps or hurts. The utility of an assistant changes as dependencies, architectures, and coding conventions change. Measurement prevents the team from relying on stale impressions of model capability.

**Questions for Future Expansion**
What concrete AI4HPC or Microelectronics examples exist for code porting, performance tuning, test generation, or codebase-aware development?
Which scientific software workflows have reproducible before/after evidence showing gains in productivity, correctness, performance, or defect reduction? For example, which AI4HPC workflows provide such evidence across code translation, portability, optimization, and codebase-aware development?
Which validators are sufficient for different classes of AI-assisted software work, such as translation, portability, optimization, refactoring, test generation, and codebase-aware development? Which verification layers are currently strongest: compile/test, benchmark comparison, static analysis, numerical validation, or human domain review?
What kinds of code changes are currently considered out of bounds for autonomous assistance?
Should the report include practical examples of prompt templates for refactoring, translation, portability, optimization, and regression-test generation?

**Figure Suggestions**
Primary figure: AI co-developer workflow showing source context and task framing feeding AI patch generation, then compile, test, benchmark, human review, and accept or reject steps.
Optional figure: Acceptance funnel plot showing candidate AI-generated changes narrowing from proposed patches to builds passing, tests passing, benchmark-safe changes, and final accepted merges.

## 10.5 Relationships to Other Patterns

- Use with *Iterative Correction / Guardrails Patterns* when generated code or workflow edits need tests, compilers, schemas, or review gates.
- Use with *Versioning and Change Management* when AI-produced changes must remain auditable and reversible.
- Use with *Agent-as-User Delegation* when software tasks move beyond suggestion into action, such as preparing patches, updating build scripts, triaging failures, or executing repository workflows on the user's behalf.
- Use with *Institutional Memory and Knowledge Capture* when review comments, postmortems, recurring fixes, and design patterns should become reusable context for later development work.

# Part IV

# Operational Deployment Patterns

Operational deployment patterns address what happens when AI leaves an isolated prototype and becomes part of a live scientific system. These chapters focus on large-scale execution, streaming and time-sensitive workflows, interaction with physical systems, and the portability challenges that arise when workflows must run across heterogeneous facilities and computing environments. They are grouped together because each pattern deals with embedding AI into real operational settings where performance, reliability, and interoperability matter alongside model quality.

# 11. At-Scale AI Training and Inference

## 11.1 Problem

Scientific teams want to train or run large AI models over massive, distributed, heterogeneous data and compute environments, but they face bottlenecks in coordination, bandwidth, hardware heterogeneity, governance, and validation at scale. The AI-specific problem is that once models, datasets, or inference workloads become large enough, scientific progress depends as much on systems design as on modeling insight. Training can stall on communication overhead, checkpoint failures, skewed workers, and storage contention. Inference can become dominated by model loading, queueing, and unpredictable latency rather than arithmetic throughput. At this scale, success is no longer "the model trains" or "the model runs"; success means the full workflow remains efficient, reproducible, and governable across many devices, sites, and failure modes.

Naive solutions fail because they extend small-scale workflows linearly. A training script that works on one node may collapse when synchronization dominates compute, when data loaders cannot feed accelerators fast enough, or when a single failed worker invalidates an expensive run. Likewise, a model served successfully for a few users can become unstable when many concurrent jobs compete for memory, accelerator time, or network bandwidth. Another common failure mode is optimizing solely for peak throughput while ignoring provenance, fault tolerance, or fairness across heterogeneous resources. In scientific contexts this is especially problematic because teams such as GridAI,FAMOUS, AI4HPC, Fusion, and Axess often operate across mixed platforms, evolving datasets, and organizational constraints. The hard part is not just scaling the math; it is scaling the entire socio-technical workflow so that large runs are still explainable, restartable, and scientifically credible. FAMOUS federates datasets that are naturally distributed by their origin in bespoke laboratories and technical expertise. The datasets differ in modes, size, and structural characteristics, and therefore requires contrasting AI models, distinct data management methods, and diverse computing resources for training and inference.

The GridAI team currently operates in multi-GPU and multi-node regimes and is devel-

oping cross-site scaling via Federated Learning, which they have already demonstrated for training large-scale foundation models with the APPFL package (specifically a Llama2-7B LLM). For multi-GPU and multi-node scaling, we rely primarily on PyTorch DistributedDataParallel, with checkpointing handled through PyTorch's native capabilities. The dominant bottleneck we observe today is data loading: because our data is grouped by grid topology, larger topologies produce per-group sizes that no longer fit in memory, so we have developed and deployed on-disk / fully sharded dataset loading to handle this. GPU memory is not currently a pressure point, though we anticipate it becoming one as model size increases. Our target platforms are highly heterogeneous, spanning DOE LCF supercomputers and commercial clouds with NVIDIA, AMD, and Intel hardware under different scheduling policies, and we use a modular architecture design to execute and optimize training correctly across them. Beyond accuracy, our scale-validation evidence includes successful runs at 2048 nodes on both Aurora and Frontier with near-linear throughput scaling.

## 11.2 Characteristics of the Solution

The shared pattern uses distributed execution, orchestrated coordination, modular architectures, external tool and evaluator integration, scale-aware synchronization, system-level validation, and checkpoint-based fault tolerance.

A first component is *distributed execution with explicit partitioning*. Data, model states, and work units are partitioned across nodes, accelerators, or sites using data parallelism, model parallelism, pipeline parallelism, or hybrid strategies. This matters because scale is achieved by structuring work deliberately, not by hoping the runtime will hide all distribution costs.

A second component is *coordination and scheduling*. Job orchestration assigns resources, restarts failed tasks, places workloads near data when possible, and enforces priorities or quotas. This matters because at-scale AI workflows compete with other users and fail in operationally complex ways that single-node code never encounters.

A third component is *budget-aware synchronization and communication*. Gradient exchange, parameter updates, cache refreshes, and checkpoint writes are managed with awareness of network and storage limits. This matters because communication and I/O often become the dominant bottleneck once pure compute is parallelized.

A fourth component is *modular model and data architecture*. Encoders, adapters, experts, or stageable submodels allow the system to fit uneven hardware and evolving modalities. This matters because monolithic designs are often difficult to place, update, or serve efficiently across heterogeneous infrastructure.

A fifth component is *orchestration of external tools and evaluators*. Scientific AI workflows often call compilers, simulators, profilers, benchmark harnesses, data stores, and remote inference services as part of the loop. These tool calls must be scheduled, logged, retried, and associated with model outputs so that the workflow remains reproducible and cost-aware.

A sixth component is *fault tolerance through checkpointing and resumability*. Long-running jobs save state frequently enough to recover from node loss, preemption, or software faults. This matters because large runs are statistically likely to experience partial failure before completion.

A seventh component is *system-level validation and observability*. Teams monitor not just loss curves and benchmark scores, but also throughput, utilization, memory pressure,

straggler rates, restart frequency, and data-path integrity. This matters because poor systems behavior can silently invalidate conclusions about model quality or cost.

## 11.3 Design Tradeoffs

*Communication overhead vs. convergence speed* – More frequent synchronization improves consistency and can stabilize optimization, but it becomes expensive at scale. The right point depends on whether the run is compute-bound or communication-bound and whether small accuracy gains justify large infrastructure cost.

*Global model quality vs. governance constraints* – Broad data access and flexible scheduling can improve scientific performance, but may conflict with placement rules, confidentiality boundaries, or site-specific policy. Scientists should decide early which constraints are immovable and design the distributed strategy around them rather than treating governance as an afterthought.

*Hardware heterogeneity vs. fairness and efficiency* – Optimizing for the fastest accelerators maximizes throughput, but can disadvantage contributors with weaker systems or create models tuned to a subset of hardware. Mixed clusters often require tiered participation, adaptive batch sizes, or modular workloads to remain scientifically and organizationally viable.

*Model throughput vs. evaluator throughput* – In iterative scientific AI workflows, LLM inference may not be the dominant cost. Compilation, simulation, profiling, data staging, or benchmark execution can become the limiting step. Teams should profile the complete loop before optimizing model serving alone.

*Monolithic architecture vs. modular extensibility* – Large unified models simplify some workflows and can maximize parameter sharing, while modular designs better fit uneven modalities, memory budgets, and deployment targets. The better choice depends on whether the dominant challenge is cross-modal integration or operational deployability.

*Fault tolerance vs. raw efficiency* – Frequent checkpoints, rich telemetry, and restart logic improve robustness, but consume time and storage. For expensive multi-day runs, that overhead is usually justified; for short experiments, lighter protection may be sufficient.

## 11.4 Implementation Advice

- Design for distributed execution from the start.
  Choose partitioning and data layout strategies early instead of retrofitting them after the model grows too large. This includes deciding whether the workflow is primarily data-parallel, model-parallel, pipeline-parallel, or hybrid, and ensuring the input pipeline can keep pace with accelerators. Early scale-aware design prevents later rewrites around memory and bandwidth bottlenecks.
- Make synchronization budget-aware.
  Treat communication as a finite scientific resource. Measure how much time is spent on all-reduce, parameter exchange, checkpoint writes, and remote reads, and adapt batch sizes, synchronization frequency, precision, or compression accordingly. Teams often gain more from reducing communication overhead than from marginal arithmetic optimization.

- Validate both scientific quality and systems behavior at scale.
  A model that converges on a small test cluster may behave differently when stragglers, queueing, storage contention, and worker failure appear. Track scientific metrics alongside throughput, utilization, error rates, restart counts, and data-integrity checks. This helps distinguish true modeling problems from infrastructure-induced artifacts.
- Instrument the full AI workflow loop.
  Track inference latency, queue delay, tool-execution time, compile/test time, simulation or benchmark cost, accelerator utilization, retry counts, failure rates, checkpoint overhead, and artifact movement. At scale, the slowest or least reliable part of the loop often determines scientific throughput.
- Build in checkpoints, rollback, and provenance capture.
  Save enough state to resume long runs safely: model weights, optimizer state, scheduler state, data shard position, environment version, and configuration manifests. Use immutable checkpoint naming, retention policies, and model registries so that large experiments can be compared, resumed, and audited without guesswork.
- Plan explicitly for heterogeneous hardware and participation.
  Assume the cluster or collaboration will include different accelerators, CPU-only nodes, uneven memory sizes, and varying site availability. Use adaptive batch sizing, capability-aware scheduling, modular components, and placement rules that match workload shape to actual resources. For workflows that generate, transform, or optimize executable artifacts, also evaluate portability and performance behavior across the target execution matrix, including relevant compilers, programming models, accelerator architectures, and facility environments.
- Prefer modular architectures when modalities or platforms differ.
  If the workflow spans different data types or deployment environments, modular encoders, adapters, or expert components can reduce retraining cost and simplify placement. They also make it easier to update one part of the system without destabilizing the whole model, which is valuable in long-lived scientific programs.

**Questions for Future Expansion**

What concrete scaling regimes are represented in FAMOUS, AI4HPC, Fusion, and Axess: multi-GPU, multi-node, cross-site, or all three?

Which orchestration and checkpointing tools are actually used in the repository or related workflows?

Are there known bottlenecks dominated by network, storage, data loading, or model memory that should be called out explicitly?

How heterogeneous are the target platforms, and do any teams already use modular or adapter-based architectures to cope with that diversity?

What scale-validation evidence exists today beyond model accuracy, such as throughput measurements, failure rates, or replayable large-run manifests?

**Figure Suggestions**

Primary figure: Distributed training and inference systems map showing data shards, model partitions, scheduler, accelerators, checkpoint store, telemetry, and communication paths across heterogeneous infrastructure.
Optional figure: Bottleneck breakdown chart showing time or cost divided across compute, communication, data loading, checkpointing, and idle or straggler overhead.

## 11.5 Relationships to Other Patterns

- Use with *Data Pipelines and Collection Patterns* when scale depends on reliable, reproducible data preparation and lineage.
- Use with *AI-Augmented Software Development* when large training and serving systems need automation, tooling, and code-generation support.
- Use with *Surrogates for Optimization* when large-scale training is being used to create fast approximators for expensive tasks.
- Use instead of *Federated and Confidential Data Patterns* when centralization is feasible and the bottleneck is compute, throughput, or deployment scale rather than governance.

# 12. Near Real-Time / Streaming Systems

## 12.1 Problem

Some workflows must make scientifically meaningful decisions from continuously arriving data quickly enough to influence the ongoing experiment or system. The AI-specific challenge is that inference is no longer a detached batch analysis step; it becomes part of a live control or triage loop in which stale answers are often as harmful as wrong answers. Data arrive with jitter, missingness, clock skew, transient faults, and shifting operating regimes. Meanwhile the model is asked to summarize the current state, detect events, rank options, or issue recommendations before the relevant window closes. That creates a fundamentally different design problem from offline analysis, because success depends on timing guarantees, state management, and graceful behavior under uncertainty as much as on predictive quality.

Naive solutions fail because they import batch assumptions into a live setting. A pipeline that waits to collect complete windows, reprocess large histories, or invoke heavyweight models on every update may produce excellent outputs too late to matter. A second failure mode is ignoring state: streaming decisions often depend on temporal context, sensor freshness, and event ordering, not just the latest record. A third failure mode is treating training-time distributions as stable even though live streams drift, instruments recalibrate, or experimental conditions change mid-run. Finally, teams sometimes automate the decision step without engineering degraded modes for dropped messages, delayed inference, or uncertainty spikes. This is why the pattern matters for teams such as Fusion, FAMOUS, and Synapse-I: the common problem is not simply fast prediction, but maintaining a trustworthy observe-infer-decide-act loop when the world continues changing during computation. FAMOUS combines biological models at different time scales (sub-second to years) and size scales (nanometers to meters). Consequently, multiscale models naturally incorporate prediction and reasoning at different rates, resulting in varying latency requirements for the model's sub-components (loop nests).

## 12.2 Characteristics of the Solution

The common solution is a streaming loop with continuous ingestion, lightweight inference, state estimation, a decision engine, and immediate feedback from action to observation.

A first component is *message-based ingestion with timestamps*. Events are carried through a broker or event bus with explicit time metadata, source identity, and ordering semantics. This matters because downstream inference and fusion logic can only reason correctly if the system knows what happened when, from where, and in what sequence.

A second component is *stateful stream processing*. The system maintains rolling windows, feature stores, or latent state estimates instead of recomputing from scratch on each observation. This matters because scientific decisions in streaming environments depend on trajectory and recent context, not isolated points.

A third component is *fast-path inference*. A compact or distilled model handles the immediate decision loop, often paired with simpler heuristics or thresholds. This matters because meeting latency budgets usually requires a deliberately lightweight path that can respond predictably under load.

A fourth component is *slow-path analysis and recalibration*. Richer models, retrospective diagnostics, or human review run asynchronously on buffered data. This matters because not every scientific question can be answered in the fast loop, and the slow loop is where drift analysis, retraining triggers, and deeper interpretation live.

A fifth component is *uncertainty, drift, and quality monitoring*. The workflow continuously tracks confidence, input quality, freshness, and distribution change. This matters because live systems need to detect when their own assumptions are breaking before bad outputs propagate.

A sixth component is *degraded and fallback modes*. When deadlines are missed or confidence falls, the system falls back to a simpler controller, a rules-based alert, or operator notification. This matters because a stream-processing system must be safe under overload, not just accurate under ideal conditions.

## 12.3 Design Tradeoffs

*Latency vs. model complexity* – Richer models may capture nonlinearity and longer context, but can miss operational deadlines. A useful decision rule is to first establish the maximum tolerable end-to-end latency and then fit the model family to that budget, not the other way around.

*Immediate responsiveness vs. batch efficiency* – Micro-batching and windowing can improve throughput and hardware utilization, but they introduce delay and may blur short-lived events. Scientists should choose window sizes based on the time scale of the phenomenon they must detect, not solely on cluster efficiency.

*Determinism vs. adaptivity* – Fixed pipelines are easier to validate and debug, while adaptive pipelines handle drift better. In more tightly governed settings, adaptivity should be concentrated in bounded parameters or model-selection rules rather than unconstrained live retraining.

*Automation vs. human oversight* – Full autonomy improves reaction speed and reduces staffing burden, but it increases risk when the stream enters unfamiliar regimes. Human review remains especially valuable for rare events, anomaly confirmation, and interventions that cannot be reversed.

*Redundancy vs. resource utilization* – Duplicate services, shadow models, and fallback controllers improve resilience, but they consume compute and operational effort. The right landing point depends on whether the stream drives exploratory awareness, expensive experiments, or active control.

## 12.4 Implementation Advice

- Separate fast-path and slow-path models.
  Use a deliberately simple, well-profiled model or ruleset for immediate decisions, and push richer interpretation, retraining analysis, and human-oriented summaries into a slower asynchronous lane. This split is often more robust than trying to make one model satisfy both sub-second response needs and deep scientific interpretation.
- Monitor drift and uncertainty explicitly.
  Track input distribution changes, missingness, stale sensors, calibration shifts, confidence degradation, and changes in event rates. Practical methods include population statistics over rolling windows, calibration error monitoring, out-of-distribution scoring, and alarms on upstream data quality. If the workflow cannot tell when its assumptions have changed, it cannot decide when to trust itself.
- Use message-based, timestamped state management.
  Adopt event logs, brokers, or stream processors that preserve timestamps and identifiers, and maintain explicit state stores for rolling context. This supports replay, late-arriving data handling, and forensic analysis after a bad decision. Timestamp discipline is especially important when multiple sensors or subsystems feed the same decision engine.
- Design fallback and degraded modes from the start.
  Plan for dropped messages, slow inference, unavailable models, and uncertain predictions. Examples include reverting to threshold alarms, freezing the last trusted estimate for a bounded period, reducing actuation authority, or switching to operator review. Degraded behavior should be tested as a primary requirement rather than left as emergency improvisation.
- Validate timing behavior as a first-class requirement.
  Measure end-to-end latency, queue depth, jitter, and deadline miss rates under realistic load. Scientists often validate model quality thoroughly and system timing superficially, even though a high-quality answer that arrives too late is operationally equivalent to failure. Load tests, replay tests, and stress scenarios should therefore be part of acceptance criteria.
- Use controlled, reversible update policies for live models.
  Do not replace a production streaming model the instant a new checkpoint appears. Use canary deployment, shadow evaluation, rollback-ready registries, and bounded release windows so that model updates do not destabilize live operation. In most scientific settings, stability and interpretability of updates matter more than maximum update frequency.

**Questions for Future Expansion**
What are the actual latency budgets for Fusion, FAMOUS, and Synapse-I workflows,

and which decisions are deadline-sensitive versus freshness-sensitive?
Are there existing message buses, stream processors, or time-synchronization mechanisms in use that can be referenced concretely?
What forms of drift are most common in these systems: instrument drift, regime change, operator behavior, or data pipeline faults?
Which fallback behaviors are already deployed or planned, and have they been tested under overload or data-loss scenarios?
Are there examples of shadow deployment, canary release, or replay testing that could strengthen the best-practice guidance?

**Figure Suggestions**
Primary figure: Fast-path and slow-path streaming architecture showing event ingestion, timestamped state management, lightweight low-latency inference, decision engine, and a slower analysis or recalibration lane.
Optional figure: End-to-end latency budget waterfall showing ingest, queueing, feature computation, inference, decision, and actuation relative to a timing deadline.

## 12.5 Relationships to Other Patterns

- Use with *Cyber-Physical Systems* when streaming decisions depend on live sensors, actuators, or changing physical state.
- Use with *Safety-Critical Systems* when delayed or unstable responses can create high-consequence outcomes.
- Use with *Data Pipelines and Collection Patterns* when the system needs validated ingestion, time-aware records, and reliable movement of streaming data into downstream consumers.

# 13. Cyber-Physical Systems

## 13.1 Problem

AI is increasingly used to guide workflows that interact with real physical systems, but real-world data is expensive, noisy, delayed, and limited. The AI-specific problem is that learning must bridge two worlds that behave differently: a computational world where models can be trained quickly on simulated or historical data, and a physical world where interventions have cost, wear, risk, and partial observability. Teams need to know whether behavior learned from simulated, surrogate, or sparsely observed data remains reliable when deployed against the actual system, especially when sensing is imperfect and the operating envelope is broad.

Naive solutions fail because they ignore the simulation-to-reality gap or underestimate the importance of closed-loop effects. A model that predicts well on logged data can behave poorly once its outputs begin influencing the system that generates future data. Likewise, a surrogate or digital twin can accelerate development but still omit sensor artifacts, timing delays, coupled failure modes, or unmodeled physics. Another common failure mode is assuming that more autonomy is always better: in cyber-physical settings, poorly bounded exploration can damage equipment, waste scarce machine time, or produce scientifically invalid measurements. Finally, teams often validate components separately rather than validating the end-to-end loop from sensing through inference, decision, actuation, and recovery. This is why the pattern recurs for Fusion, FAMOUS, Axess, and MAIQMag: the underlying problem is learning and decision-making under physical consequence, limited data, and imperfect models of reality.

## 13.2 Characteristics of the Solution

The recurring solution is a closed loop between sensing, inference, decision, and physical action, often supported by surrogates, digital twins, simulation-in-the-loop, automated testbeds, and uncertainty-aware gating.

A first component is *sensing and state estimation*. Raw measurements are filtered,

fused, and converted into an estimated system state with uncertainty. This matters because physical systems are only partially observed, and decisions made on raw noisy measurements are often unstable.

A second component is *simulation, surrogates, or digital twins*. Faster approximations of the physical system are used to test candidate actions, generate training data, and bound plausible outcomes. This matters because direct experimentation is often too slow, expensive, or risky to serve as the only learning path.

A third component is *simulation-in-the-loop or hardware-in-the-loop validation*. Candidate policies or recommendations are exercised against increasingly realistic environments before touching the real system. This matters because many failures only emerge in the interaction between controller, sensor timing, and plant dynamics.

A fourth component is *uncertainty-aware gating*. The workflow checks whether the proposed action lies inside a validated envelope and whether model uncertainty is acceptable. This matters because the central question is not just "what action is best" but "is this action safe enough to try now?"

A fifth component is *fallback to trusted controllers or higher-fidelity methods*. When the AI system is uncertain or the system moves outside known regions, the workflow reverts to conservative baselines. This matters because cyber-physical safety depends on maintaining control authority during novelty and fault conditions.

A sixth component is *continuous recalibration with real measurements*. Observed outcomes are used to correct the digital twin, update surrogate error models, and refine operating envelopes. This matters because deployment is not the end of modeling; it is part of ongoing system identification.

## 13.3 Design Tradeoffs

*Fidelity vs. speed* – High-fidelity simulators and digital twins may better capture physical behavior, but they are often too slow for inner-loop use. Faster surrogates enable closed-loop operation but may miss subtle effects. A practical compromise is to use surrogates in the inner loop and reserve high-fidelity models for screening, calibration, and periodic challenge cases.

*Simulated coverage vs. real-world validity* – Simulation can cheaply cover broad parameter ranges, rare events, and hypothetical interventions. However, overreliance on synthetic coverage can hide the fact that key sensing artifacts or failure modes are absent. Scientists should therefore ask not only whether the simulator covers enough cases, but whether it covers the right failure mechanisms.

*Autonomy vs. oversight* – Autonomous orchestration accelerates experimentation and can react faster than operators in tight loops, but human oversight remains important for costly interventions, exceptional states, and scientific interpretation. The dividing line is often whether the intervention is reversible and whether the system remains inside a well-characterized operating envelope.

*Broad operating envelope vs. robustness* – Extending AI to more regimes increases utility, but pushes the model toward extrapolation where data are sparse and uncertainty estimation is hardest. It is often better to certify narrow envelopes incrementally than to claim universal coverage without enough evidence.

*Continuous adaptation vs. reproducibility* – Updating models with new measurements

improves realism and long-term performance, but complicates auditability and exact replay. Scientists should decide early whether the workflow is primarily an exploratory adaptive system or a reproducible validated instrument, because that choice affects release cadence and governance.

## 13.4 Implementation Advice

- Validate simulation and surrogate data against reality repeatedly.
  Do not treat the digital twin or surrogate as valid once and for all. Compare predictions against fresh physical measurements, maintain error summaries by regime, and identify where simulation is trustworthy versus where it drifts. This can be done through holdout physical experiments, periodic challenge cases, or online residual monitoring.
- Use uncertainty-aware gating before physical actions.
  Require candidate actions to pass checks on uncertainty, operating envelope, and predicted consequence before they reach the plant or instrument. Practical implementations include confidence thresholds, conformal intervals, out-of-distribution detectors, and explicit domain rules for forbidden regions. The goal is to prevent the system from exploring blindly in poorly understood regimes.
- Design explicit fallback paths.
  Keep a trusted controller, conservative baseline policy, or manual override path available whenever AI recommendations are uncertain, delayed, or inconsistent. Fallbacks should be integrated into the orchestration layer and tested under fault conditions. In cyber-physical systems, the existence of a fallback is often more important than squeezing out marginal extra performance from the AI model.
- Keep humans in the loop for high-impact decisions.
  Reserve expert review for interventions with high cost, irreversibility, or unclear validity. Useful patterns include approval checkpoints, ranked recommendation lists instead of direct actuation, and interfaces that expose model rationale, uncertainty, and recent system context. Human involvement works best when the workflow is designed to support rapid comprehension rather than post hoc guessing.
- Continuously incorporate real measurements to recalibrate models.
  Use deployment as a source of additional evidence. Update surrogate residual models, retrain state estimators, and refine safe operating envelopes using observed outcomes. The important practice is to separate exploratory data capture from validated release so that adaptation improves the system without silently changing its scientific meaning.
- Preserve provenance across sensing, modeling, and actuation.
  Record sensor versions, calibration status, model versions, policy settings, action commands, and observed outcomes in one lineage chain. This supports replay, root-cause analysis, and comparison between simulated and physical behavior. Without integrated provenance, teams cannot distinguish whether a failure came from the model, the sensor, the actuator, or the orchestration logic between them.

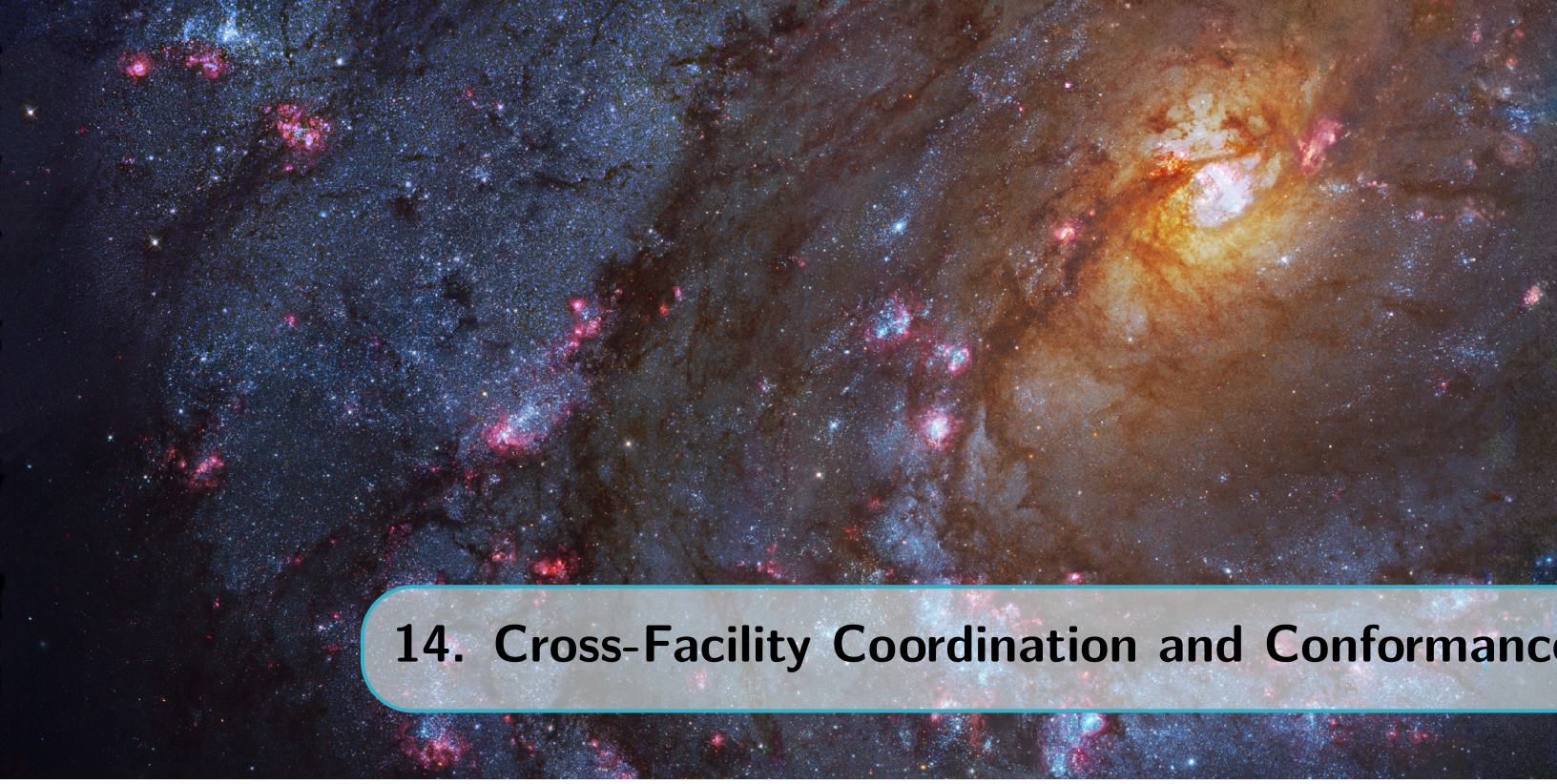
**Questions for Future Expansion**
Which of Fusion, FAMOUS, Axess, and MAIQMag already use digital twins, surrogates, or hardware-in-the-loop testbeds that can be described concretely?

What are the known simulation-to-reality gaps in these workflows: sensing artifacts, timing delays, unmodeled physics, or actuator saturation?
Which actions are reversible enough for autonomous execution, and which require mandatory expert approval?
Are there existing uncertainty-estimation or safe-envelope methods used by these teams that should be named explicitly?
What provenance systems currently connect sensing, modeling, and actuation, and where are the remaining audit gaps?

**Figure Suggestions**
Primary figure: Closed-loop cyber-physical workflow showing sensing, state estimation, AI or surrogate or digital twin, uncertainty gate, action, physical system response, and fallback controller.
Optional figure: Simulation-to-reality gap graphic comparing validated simulated operating regions with actual observed behavior and highlighting mismatch regions.

## 13.5 Relationships to Other Patterns

- Use with *Safety-Critical Systems* when model errors can create equipment, process, or mission harm.
- Use with *Near Real-Time / Streaming Systems* when decisions must respond to live physical dynamics.
- Use with *Physics-Informed AI Patterns* when physical processes provide both constraints and the domain data needed to keep predictions or control proposals meaningful.
- Use with *Surrogates for Optimization* when operating the real system is expensive enough that planning, tuning, or design exploration should happen first with faster approximations.



# 14. Cross-Facility Coordination and Conformanc

## 14.1 Problem

Many scientific workflows no longer live inside a single instrument, laboratory, or computing environment. They span beamlines, control systems, data services, workflow engines, facility-edge resources, leadership HPC systems, cloud services, and external user environments. Each site may expose different APIs, identity models, metadata conventions, scheduling practices, container policies, and tool interfaces. The AI-specific problem is that an agentic workflow that works at one facility can fail to transfer elsewhere, not because the scientific task changed, but because the surrounding infrastructure is inconsistent.

Naive solutions often rely on one-off adapters, local scripts, and team-specific conventions. That can produce a successful demonstration at a single site, but it does not scale across facilities. The same agent may need to learn different naming schemes for similar devices, different ways to launch jobs, different log formats, or different approval paths at each location. In the worst case, portability is only nominal: the workflow appears reusable, but hidden assumptions about authentication, metadata, scheduling, or tool behavior break reproducibility as soon as it crosses site boundaries. Scientific organizations then pay repeatedly for the same integration work and still cannot show that a workflow is trustworthy outside the original deployment context.

The deeper problem is not merely interoperability in the abstract. It is the absence of shared infrastructure that allows workflows, tools, and agents to be reused and evaluated consistently across heterogeneous facilities. This becomes especially important when DOE or similarly distributed scientific ecosystems want to coordinate experiments, computing, and data across multiple labs rather than optimizing each site in isolation.

MAIQMag is a direct fit because the workflow must connect multimodal experimental data from DOE laboratories and user facilities with synthetic-data generation, forward solvers, and inference on HPC resources.

## 14.2 Characteristics of the Solution

The recurring solution is to pair shared coordination infrastructure with explicit conformance mechanisms that test whether workflows and tools behave consistently across sites.

A first component is *standardized interfaces*. Facilities expose common API shapes, metadata fields, and invocation contracts for the parts of the workflow that must be shared. This matters because portability is impossible when every site expresses the same operation differently.

A second component is *shared registries*. Tool definitions, agent capabilities, service endpoints, and sometimes workflow templates are published in registries that multiple facilities can consume. This matters because scientists cannot reuse what they cannot reliably discover or identify.

A third component is *portable execution packaging*. Containers, reproducible environments, and versioned workflow definitions allow the same logic to run across different computing contexts. This matters because coordination across facilities often fails on environment drift before it fails on scientific logic.

A fourth component is *federated identity and access context*. Cross-site workflows need a way to carry authenticated identity, scoped authority, and audit information across organizational boundaries. This matters because a portable workflow is only useful if it can also execute legally and traceably at each site.

A fifth component is *conformance testing*. Facilities validate shared tools, APIs, and workflows against common tests, reference traces, or expected behaviors. This matters because nominal standards do not guarantee real interoperability; conformance tests reveal where implementations diverge in practice.

Finally, mature ecosystems support *cross-site demonstrations and reference workflows*. Shared workflows are exercised end to end across heterogeneous facilities, using facility-specific tools behind common interfaces. This matters because the real measure of coordination infrastructure is not whether a standard exists on paper, but whether it enables reproducible scientific execution across sites.

## 14.3 Design Tradeoffs

*Standardization vs. local optimization* – Common interfaces improve reuse, but they can constrain local systems that have domain-specific capabilities or legacy constraints. Scientists should standardize the shared surface area needed for portability while allowing local specialization behind the interface.

*Central registries vs. federated autonomy* – Central registries simplify discovery and consistency, while local control supports sovereignty and site-specific governance. A practical compromise is federated participation in a common registry model rather than full central control of every tool and service.

*Strict conformance vs. implementation flexibility* – Tight conformance requirements improve reproducibility, but may slow adoption for sites with older infrastructure. The right landing point is usually a minimal mandatory contract plus optional extensions, paired with tests that make divergence visible rather than hidden.

*Portability vs. performance tuning* – A workflow packaged for broad portability may leave some site-specific performance on the table. Scientists should decide when the value of

cross-site reuse outweighs the value of deep local optimization, and document where the workflow intentionally trades one for the other.

*Shared infrastructure vs. coordination overhead* – Building cross-facility systems requires governance, maintenance, and community agreement. That cost is justified most clearly when multiple sites repeatedly solve similar problems and when shared infrastructure can eliminate duplicated integration work.

## 14.4 Implementation Advice

- Standardize the interface, not necessarily the implementation.
  Define common request and response contracts, metadata fields, authentication hooks, and error semantics for shared workflow functions. Then allow facilities to keep local implementations as long as they satisfy the contract. This approach improves portability without forcing uniform internals.
- Publish reusable tools and workflow definitions in shared registries.
  Registries should include stable identifiers, versions, interface descriptions, capability metadata, and policy context. This makes it possible for agents and scientists to discover reusable components across sites without relying on informal personal knowledge.
- Use conformance tests, not only documentation.
  Written standards are insufficient on their own. Build executable tests, reference workflows, golden traces, and expected-behavior suites that facilities can run to show whether their implementations are actually compatible.
- Package workflows for reproducible execution across sites.
  Use containers, versioned environments, and explicit dependency capture so a workflow can be rerun on different computing infrastructure without hidden environment assumptions. Reproducible packaging is a prerequisite for meaningful cross-site coordination.
- Carry identity, authorization, and audit context across facility boundaries.
  A cross-facility workflow should preserve who initiated it, what scope of authority it has, what services it invoked, and where each action occurred. Coordination is incomplete if execution is portable but identity and accountability are not.
- Validate portability through cross-site demonstrations.
  Choose a small number of representative workflows and run them across heterogeneous facilities using site-specific tools behind common interfaces. This reveals where the real interoperability barriers are and produces stronger evidence than isolated local pilots.

**Questions for Future Expansion**
Which workflow surfaces are the best candidates for early standardization: job submission, metadata access, logbook access, tool invocation, or digital-twin interfaces?
What should a minimum shared registry schema contain for tools and agent capabilities?
Which conformance tests would best expose false portability across facilities?
How should optional local extensions be represented without breaking common workflows?

What reference cross-site demonstrations would provide the strongest evidence that coordination infrastructure is working in practice?

**Figure Suggestions**
Primary figure: Cross-facility reference architecture showing several facilities with different local tools behind common interfaces, shared registries, conformance testing, and federated identity.
Optional figure: Conformance testing ladder showing standard specification, reference workflow, executable test suite, and site-by-site pass or divergence results.

## 14.5 Relationships to Other Patterns

- Use with *Versioning and Change Management* when conformance depends on shared versions of APIs, models, schemas, procedures, or policies.
- Use with *Institutional Memory and Knowledge Capture* when coordination depends on preserving conventions, interface expectations, and operational decisions across sites.
- Use with *Agent-as-User Delegation* when delegated agents must call facility APIs or services in the same way across organizations.

# Part V

# Control and Assurance Patterns

Control and assurance patterns focus on making AI-enabled workflows governable, safe, and resilient under uncertainty. These chapters collect the recurring mechanisms used to constrain behavior, delegate authority safely, protect confidential information, and deliberately probe for failure before deployment or broader adoption. They belong together because each pattern is less about what AI can do in principle and more about how an organization keeps that capability inside acceptable technical, operational, and institutional boundaries.

# 15. Iterative Correction / Guardrails Patterns

## 15.1 Problem

AI systems in scientific workflows often produce outputs that are directionally useful but not consistently safe, correct, or in-bounds on the first pass. This is especially common when models generate code, suggest control parameters, summarize scientific evidence, rank options under uncertainty, or infer actions from partially observed state. The first answer may be plausible enough to pass a superficial review while still violating hidden constraints such as numerical stability, policy restrictions, physical feasibility, formatting requirements, or mission-specific operating envelopes. In scientific settings, that gap between plausibility and trustworthiness matters because the output may trigger expensive simulations, consume scarce instrument time, contaminate a dataset, or steer a human operator toward a bad decision.

Naive solutions fail in two common ways. One is to trust the model too early: teams accept fluent outputs as if confidence were equivalent to correctness, then discover failures only after they have propagated into downstream workflows. The other is to react by adding a single coarse approval step, which catches obvious problems but misses subtle ones and becomes a bottleneck as throughput rises. AI-specific failure modes also make ordinary software validation insufficient. Models can produce non-deterministic outputs, degrade under distribution shift, exploit ambiguities in prompts or interfaces, and fail silently in ways that are hard to distinguish from novel but valid solutions.

The recurring need is therefore not merely validation in the traditional sense, but a structured correction-and-containment loop. Teams such as MOAT, GRID, and Fusion are natural candidates where side effects and safety matter; AI4HPC and Microelectronics fit the pattern where generated code or workflow steps must pass trusted compilers, tests, and numerical checks before adoption. The broader pattern generalizes to any scientific workflow in which AI outputs are useful enough to keep in the loop, but not reliable enough to execute without mediation.

GridAI enforces guardrails based on solution feasibility and physical system constraints.

GridAI computes physics diagnostics directly from every surrogate output, including power-balance error, voltage-violation counts, line loadings, and per-scenario severity scores, so any constraint violation is visible at the point of use; this is reinforced by the augmented-Lagrangian physics loss the surrogate is trained with, and a piloted drift-detection harness monitors a physics-consistency residual to flag when operating conditions move outside the regime where the surrogate is reliable. GridAI evaluates model output against these feasibility metrics and against the rankings of a trusted physics-based solver, while workflow output is evaluated for orchestration correctness, confirming that the right stages rerun when an input changes, and for end-to-end throughput. For the fallback path, GridAI keeps the surrogate out of the safety-critical path, since it is used only to screen and rank, it is interchangeable with a classical physics-based solver behind the same interface, and the final consequence stage re-solves each selected scenario independently under its own physics-based model so results never inherit surrogate approximation error; rollback itself is handled by content-hashed artifacts, which let the workflow recover an exact earlier state without recomputation when a change is reverted.

## 15.2 Characteristics of the Solution

The common solution is to wrap AI outputs in an explicit control loop with several recurring components.

First, the workflow needs *layered, machine-checkable validation*. Each layer catches a different failure mode and provides feedback for correction or escalation. These layers can include syntax checks, schema checks, policy rules, compilation steps, unit and regression tests, numerical sanity bounds, domain-specific assertions, or simulation-based verification. They matter because they transform vague concerns about bad output into concrete pass/fail or scored criteria.

Second, the workflow benefits from *structured outputs and intermediate plans* rather than free-form text alone. When the AI emits code patches, parameter sets, action plans, or typed records, downstream tools can inspect them reliably. This reduces ambiguity and makes correction more targeted.

Third, robust systems include *bounded iterative correction*. Instead of one generation attempt, the workflow allows the model to revise its output using validator feedback, compiler errors, judge outputs, or discrepancy reports. The bound matters because unlimited retries can hide instability, burn compute, and create the illusion of convergence.

Fourth, teams typically need *risk-based routing*. Low-risk cases can proceed automatically, medium-risk cases may require additional checks, and high-risk cases should escalate to a human or fall back to a trusted method. This matters because not every output deserves the same latency or level of review.

Fifth, strong implementations maintain *fallback and rollback paths*. If the AI cannot produce an acceptable answer, the system should revert to a known-good state, a simpler heuristic, a prior approved artifact, or a human-operated path. Without fallback, guardrails can prevent action but not preserve workflow continuity.

Finally, effective guardrail systems capture *provenance and failure logs*. Recording prompts, models, validation outcomes, correction attempts, and decision paths makes it possible to diagnose recurrent failure modes, improve prompts and tests, and justify why an output was accepted or blocked.

## 15.3 Design Tradeoffs

*Speed vs. assurance* – Additional checks, retries, and approval points improve reliability, but each layer adds latency and compute cost. Also not all checks provide the same kind of assurance: syntax, compilation, and schema validation are useful for fast iteration, but they can create false confidence if treated as evidence of scientific correctness. Regression tests, numerical comparisons, simulation-based validation, and performance benchmarks provide stronger assurance, but they are slower and more expensive to run. Scientists should therefore place the heaviest guardrails around side-effectful or expensive actions, while using lighter-weight checks for exploratory drafting and low-cost iterations. The right landing point depends on the cost of a false accept compared with the cost of delay.

*Automation vs. human oversight* – Full autonomy scales well for repetitive cases, but ambiguous or high-consequence outputs still benefit from human judgment. A useful compromise is to reserve human review for classes of events: repeated correction failures, low-confidence decisions, novel operating regions, or outputs that would change an experiment or production workflow.

*Hard constraints vs. flexibility* – Hard guardrails prevent illegal actions and preserve safety, but they can also reject borderline outputs that are scientifically interesting. Soft scoring or staged review is often better for discovery-oriented tasks, while hard blocking is more appropriate when outputs affect facilities, control systems, or irreversible computations.

*Rich failure context vs. simplicity* – Detailed validator feedback helps the model self-correct, but assembling rich diagnostics takes engineering effort. Teams should prioritize the smallest feedback payload that is still actionable: compiler errors, failing test names, violated bounds, or schema mismatch locations are often more useful than verbose natural-language critique.

*Fallback robustness vs. maintenance burden* – Keeping a trusted non-AI path greatly improves resilience, but it must be maintained, tested, and kept compatible with the evolving workflow. This cost is justified most clearly where the workflow must continue even when the AI path is unavailable or untrusted, which is likely for teams operating near safety or schedule constraints.

## 15.4 Implementation Advice

- Put machine-checkable validators before high-impact actions.
  Do not rely on human review alone for outputs that can trigger compute, experimental, or operational consequences. Use compilers, schema validators, policy engines, unit tests, regression suites, rule-based bounds, and domain-specific assertions to convert trust decisions into reproducible checks. In practice this often means validating generated code in CI-style harnesses, validating JSON or tabular outputs against a schema, or running a fast simulator before any side-effectful action.
- Define explicit escalation paths for risky or ambiguous cases.
  Guardrails work best when failure handling is predetermined rather than improvised. Specify what happens after repeated correction failures, low-confidence outputs, conflicting validators, or out-of-distribution inputs. Common routes include human approval, queueing for expert review, switching to a safer heuristic, or halting the workflow. This keeps operators from having to invent policy under time pressure.

- Use bounded iterative correction loops.
  Allow the model to revise outputs after receiving concrete feedback, but cap the number of retries and record the reason for termination. A typical pattern is generate, validate, repair, revalidate, then either accept or escalate after a small fixed number of attempts. Bounded loops prevent silent thrashing and create cleaner data for later analysis of why the model failed.
- Preserve a known-good fallback.
  A trusted baseline may be slower or less capable, but it protects continuity when the AI path cannot satisfy the guardrails. This could be a prior approved code path, a deterministic rules engine, a simpler model, or a manual workflow. The fallback should be exercised periodically so it remains viable rather than existing only on paper.
- Log failure reasons and decision provenance.
  Store enough metadata to reconstruct why an output was accepted, corrected, blocked, or escalated. Useful fields include model version, prompt or task specification, validator outputs, risk scores, reviewer identity when applicable, and the final action taken. These logs support debugging, governance, and continuous improvement of both prompts and validators. Over time, collected logs and traces can also become regression cases, benchmark examples, and evidence for governance.
- Tie guardrails to confidence or risk signals.
  Not every case needs the same intervention. Use uncertainty estimates, anomaly scores, rule violations, novelty indicators, or downstream cost estimates to choose when to apply deeper checks or human review. Risk-based gating helps teams maintain throughput while still concentrating scrutiny on the cases most likely to matter.

**CM2US example: guardrails for surrogate confidence signals.**
In CM2US, one important guardrail was to avoid giving too much authority to inexpensive confidence signals produced by the surrogate model. In `matsim-agents`, the HydraGNN BranchWeightMLP signal was used as a practical warning indicator, not as a rigorous uncertainty estimate. When the model clearly routed a structure through one dominant output branch, the workflow treated the prediction as more routine. When the model spread the prediction across several branches, the workflow treated that case as less certain and could trigger additional validation, DFT labeling, or active learning.

The main solution was to use the signal only to decide when the workflow should be more cautious. A sharp routing signal did not mean that the prediction was guaranteed to be correct, and a diffuse routing signal did not automatically mean that the prediction was wrong. It simply indicated whether the surrogate was making a prediction in a clear or ambiguous routing regime. The lesson learned was that low-cost AI confidence signals can be useful guardrails, but only when the workflow explicitly defines what they can and cannot be used for.

**Questions for Future Expansion**
What concrete examples from Microelectronics, AI4HPC, or MOAT show iterative correction loops in production rather than prototypes?
Which validators are domain-generic versus team-specific, and which have proven most predictive of downstream failure?

Where are current workflows using hard blocking, and where are they using score-based escalation?
What evidence exists on acceptable retry limits, latency budgets, and reviewer workload for guarded AI systems?
Are there documented fallback paths and rollback procedures that could be cited directly?

**Figure Suggestions**
Primary figure: Guarded generation loop showing AI output, validator checks, repair attempt, revalidation, bounded retries, and final accept or escalate or fallback outcomes.
Optional figure: Risk-based routing tree showing low-risk auto-accept paths, medium-risk additional checks, and high-risk human review or fallback.

## 15.5 Relationships to Other Patterns

- Use with *Safety-Critical Systems* when validation and fallback become mandatory rather than optional.
- Use with *Red Teaming and Adversarial Patterns* when discovered failures should be converted into validators, escalation rules, and regression cases.
- Use with *AI-Augmented Software Development* when generated code or workflow changes need compilers, tests, and policy checks before acceptance.
- Use instead of *Red Teaming and Adversarial Patterns* when the immediate need is operational containment of outputs rather than proactive discovery of new failure modes.

# 16. Safety-Critical Systems

## 16.1 Problem

Some AI-enabled scientific workflows can influence physical systems, expensive facilities, or operational decisions with meaningful safety consequences. In these settings, the core problem is not merely prediction accuracy; it is whether an AI-assisted recommendation, generated plan, control signal, or approval decision can be trusted when the cost of error is high. A model may appear strong on benchmark data and still fail in operation because the actual deployment environment contains edge cases, missing context, coupled subsystems, stale inputs, or unmodeled constraints. Safety-critical use is therefore defined less by the presence of AI and more by the fact that downstream side effects can be hazardous, irreversible, or expensive before humans have time to intervene.

Naive solutions fail because they assume that a high-performing model should be allowed to act directly on the system it supports. That assumption breaks in several ways. First, confidence estimates are often poorly calibrated out of distribution, so the model sounds certain precisely when oversight is most needed. Second, raw natural-language or free-form outputs are difficult to validate automatically, which means unsafe intent can pass through unnoticed. Third, purely manual review does not scale well under time pressure and often becomes a rubber stamp if the workflow generates too many intermediate decisions. Fourth, teams tend to validate the model in isolation, while the actual risk lives in the full socio-technical loop: sensors, operators, orchestration logic, policies, actuators, and recovery behavior. This is why the pattern recurs across teams such as MOAT, GRID, Fusion, Prometheus, and Axess: even when the domain changes, the AI system must be prevented from becoming the final unchecked source of action.

In GridMind, the agentic workflow in GridAI, the surrogate model is used only to screen and rank scenarios, and its solutions are never passed into the safety-critical downstream stages. Every surrogate output carries explicit physics-feasibility diagnostics, including power-balance error and voltage-violation counts, so constraint compliance is visible at each step rather than assumed, and the same workflow runs interchangeably against a

classical physics-based solver that serves as the trusted reference. The cascade simulation that produces the final risk assessment re-solves each selected scenario independently under its own physics-based model, so the reported consequences do not inherit the surrogate's approximation error.

## 16.2 Characteristics of the Solution

The recurring solution is to insert an explicit mediation layer between the AI system and the side-effectful system. That layer exists to transform AI output from a direct action into a proposal that can be classified, checked, constrained, and either approved or blocked.

A first common component is *structured intent representation*. Instead of allowing free-form output to drive execution, the AI system emits typed plans, parameter sets, ranked options, or constrained action objects. This matters because structured outputs can be validated mechanically against schemas, operating envelopes, and policy rules.

A second component is *risk classification and policy gating*. Actions are assigned to risk tiers based on context, confidence, operating regime, and expected consequence. This matters because low-risk actions can often proceed automatically while high-risk actions receive stronger validation or require human approval, preserving both safety and throughput.

A third component is *independent validation*. Proposed actions are checked against trusted references such as rule engines, simulators, digital twins, static analyzers, test harnesses, or domain constraints. This matters because the model is not asked to certify its own behavior; a separate mechanism evaluates whether the proposal is physically, procedurally, or scientifically admissible.

A fourth component is *human approval and escalation paths*. Expert review is reserved for consequential, ambiguous, or novel cases rather than every event. This matters because human attention is scarce and must be concentrated where it changes outcomes.

A fifth component is *delegated authority within existing access-control systems*. Rather than treating agents as privileged automation outside normal governance, mature systems scope agent capabilities as delegated extensions of authenticated users, approved roles, or facility-managed services. This matters because safety-critical deployment depends not only on whether an action is allowed in principle, but also on who is allowed to request it, for how long, under what operating context, and with what audit trail.

A sixth component is *provenance, logging, and replay*. The workflow records model version, prompt or input context, policy version, validation outputs, approval decisions, delegation context, and execution results. This matters for post-incident analysis, certification, rollback, and learning from near misses.

A seventh component is *fallback and halt behavior*. When uncertainty is high or validation fails, the system reverts to a trusted controller, a safe default, or an explicit stop state. This matters because safety depends as much on graceful non-action as on correct action.

## 16.3 Design Tradeoffs

*Automation vs. expert oversight* – More automation reduces operator burden and can improve consistency, but high-consequence actions still benefit from human judgment, especially when the relevant risk is contextual or weakly represented in training data. Scientists should

bias toward automation for repetitive, bounded, easily reversible tasks, and toward expert review for novel configurations, expensive experiments, or actions with coupled consequences.

*Static policies vs. dynamic risk-based gating* – Fixed approval rules are easier to certify, explain, and audit. Dynamic gating can reduce unnecessary reviews by adapting to current state, model confidence, and environmental context. The right landing point depends on how mature the sensing and calibration stack is: if confidence estimates and state quality are not trustworthy, static policies are usually safer.

*Fine-grained control vs. operator usability* – Decomposing a workflow into many explicit checkpoints improves transparency and containment, but too many gates create alert fatigue and can push experts to approve mechanically. A good design places checkpoints at true hazard boundaries rather than after every computational step.

*Isolation and redundancy vs. latency* – Sandboxing, redundant validation, and staged execution improve safety but add runtime cost. In operational contexts with tighter timing constraints, that extra latency may matter. The practical question is whether the action deadline is hard real-time, near real-time, or operator-paced; the more time available, the more external validation can be inserted.

*Complete provenance vs. operational overhead* – Full capture of inputs, models, policies, decisions, and delegated authority strengthens accountability and replayability, but increases storage and engineering burden. The usual compromise is to record complete provenance for all high-risk actions and sampled provenance for low-risk high-volume events only if the system is otherwise too costly to operate.

*Fine-grained delegation vs. operational simplicity* – Narrow, time-bounded, role-scoped permissions reduce blast radius and make audits clearer, but they also increase policy complexity and can frustrate users if delegation is too hard to obtain or renew. A practical landing point is to reuse existing identity and role frameworks, then add the minimum extra delegation semantics needed for agent execution, rather than inventing a parallel security model from scratch.

## 16.4 Implementation Advice

- Place a mediation layer between AI output and side-effectful systems.
  Do not let a model write directly to actuators, schedulers, or approval systems when consequences are material. Put an orchestration service, policy engine, or workflow controller in between. In practice this often means API boundaries that accept only typed requests, admission checks implemented as rules, and execution wrappers that can deny or defer unsafe requests.
- Represent intended actions as structured, inspectable plans before execution.
  Use schemas, typed messages, constrained JSON, state-machine transitions, or domain-specific command objects instead of free text. This makes it possible to run static validation, unit tests, rule checks, and simulation-based screening before anything happens in the real system. If a scientist cannot inspect the proposed action in a compact structured form, the workflow is harder to trust.
- Require stronger checks and human approval for high-risk actions.
  Adopt a tiered review model. Low-risk, reversible actions can be auto-approved; medium-risk actions can require additional machine checks; high-risk actions should require named human approval with clear rationale. This is easier to sustain than

universal manual review and creates an auditable mapping between consequence and scrutiny.

- Validate risky outputs against trusted references such as simulators, rule engines, or test harnesses.
  Use the best independent reference available, even if it is slower than the AI model. For example, test candidate control settings in a simulator, verify generated code with compilers and regression suites, or check parameter choices against operating envelopes and interlock rules. The reference need not be perfect; it needs to be independent enough to catch classes of failure the model itself cannot detect.
- Scope agent actions through delegated authority, not privileged bypasses.
  Whenever possible, let agents act through existing authentication, authorization, and role frameworks rather than through special unrestricted service identities. Practical implementations may use time-bounded tokens, beamtime- or campaign-scoped permissions, approved delegation chains, and facility-managed services that enforce policy before execution. This sharply reduces ambiguity about what the agent is allowed to do and who is accountable for the action.
- Capture immutable provenance for prompts, models, policies, decisions, delegated permissions, and outputs.
  Record the full chain needed to explain why an action was proposed and why it was allowed. Useful tools include artifact registries, append-only logs, signed policy bundles, versioned model registries, delegation records, and workflow metadata stores. This is essential for incident response, compliance review, and reproducing the exact conditions under which a decision was made.
- Provide explicit fallback behavior when uncertainty or risk exceeds thresholds.
  Define in advance what the system does when confidence is low, validation fails, inputs are stale, or a dependent service is unavailable. Common choices are reverting to a conservative controller, requesting operator takeover, or halting the workflow in a safe state. The important point is that failure behavior should be engineered, tested, and rehearsed rather than improvised during an event.

**Questions for Future Expansion**

What concrete examples from MOAT, GRID, Fusion, Prometheus, or Axess should be cited for approval checkpoints, safety gates, or fallback controllers?

What trusted-reference mechanisms are actually used in these workflows: simulation, rule engines, digital twins, test harnesses, or human review boards?

Are there existing calibration or uncertainty-estimation methods that could support more precise guidance on dynamic gating?

How are delegated permissions represented today: service identities, user-scoped tokens, role-based access controls, or facility-managed broker services?

Which actions are reversible versus irreversible in each team context, and how should that distinction shape approval thresholds?

What evidence exists on operator burden or alert fatigue so the report can recommend practical checkpoint densities rather than abstract principles?

**Figure Suggestions**
Primary figure: AI mediation layer for safety-critical action showing structured intent, risk tiering, independent validation, human approval where needed, execution, fallback behavior, and provenance capture.
Optional figure: Action tiering visual or table showing low-risk, medium-risk, and high-risk actions with required checks, approvals, and allowable automation levels.

## 16.5 Relationships to Other Patterns

- Use with *Iterative Correction / Guardrails Patterns* when AI outputs can trigger consequential actions and need enforced validation, escalation, and fallback.
- Use with *Red Teaming and Adversarial Patterns* when known tests are not enough and the workflow must actively search for dangerous failure modes before deployment.
- Use with *Cyber-Physical Systems* when the risk is tied to physical processes, equipment, or control actions rather than only informational harm.
- Use with *Physics-Informed AI Patterns* when domain constraints can materially reduce unsafe or non-physical outputs.

# 17. Red Teaming and Adversarial Patterns

## 17.1 Problem

Static evaluation is usually the wrong mental model for AI systems that will be exposed to changing environments, unusual edge cases, strategic misuse, or combinations of conditions that were rare in training data. In scientific and operational settings, teams often begin with benchmark sets, unit tests, and a few expert-authored scenarios. Those are necessary, but they mostly measure whether the system performs on the cases the team already anticipated. They do not reliably reveal what happens when an input distribution drifts, when multiple harmless-looking conditions interact, when a model is asked to extrapolate beyond its support, or when an operator or external actor pushes the system into failure regions on purpose. The AI-specific problem is therefore not just accuracy under normal conditions, but systematic discovery of brittle behavior before that behavior reaches high-consequence use.

Naive approaches fail for predictable reasons. A larger static test set still reflects yesterday's understanding of risk. Random fuzzing produces scale but often lacks domain realism, so teams learn little about failures that matter operationally. Manual expert review can be high quality, but it does not scale to the combinatorial space of prompts, sensor states, environmental perturbations, workflow branches, or policy interactions. Purely synthetic attack generation has the opposite failure mode: it can discover weaknesses that are easy for an optimizer to exploit but irrelevant to the real system. For teams with safety-sensitive work such as MOAT or Prometheus, the challenge is to create a repeatable process that is adversarial enough to find unknown weaknesses, realistic enough to matter, and traceable enough to support mitigation and deployment decisions.

The underlying issue generalizes beyond explicitly safety-critical systems. Any team whose model output influences expensive experiments, operational prioritization, access decisions, or downstream automation eventually needs a structured way to ask, "How could this fail in ways we did not think to test directly?" Red teaming is the discipline that turns that question into a continuous engineering loop rather than an occasional manual exercise.

## 17.2 Characteristics of the Solution

The common pattern is an iterative loop for generating challenging scenarios, evaluating them against a trusted reference, computing multidimensional risk scores, collecting hard examples, and using those examples to drive retraining, gating, or deployment decisions.

A useful implementation usually includes a scenario generation layer. This may use simulation, perturbation rules, prompt mutation, search-based input optimization, historical near-misses, or domain-specific heuristics. This component matters because unknown failure modes are rarely found by waiting for them to appear naturally; teams need a mechanism that actively searches the failure surface.

It also needs a trusted evaluator. Depending on the domain, that may be a simulator, a rule engine, a reference model, a human review workflow, or a hybrid scoring stack. This matters because adversarial generation without trustworthy judgment produces noise instead of evidence.

A scoring and triage layer is equally important. Failures differ in severity, reproducibility, exploitability, and operational relevance. A multidimensional score lets scientists distinguish harmless oddities from blocking risks, and supports explicit thresholds for retraining, escalation, or release gating.

The pattern also benefits from a hard-example store or red-team corpus. Teams need to preserve the exact input, model version, environment, seed, policy state, and outcome. This matters because the value of a red-team finding is not just discovery, but replay, clustering, root-cause analysis, and regression prevention.

Finally, mature usage includes a feedback path into the development lifecycle. Findings should update test suites, data augmentation, training sets, prompts, policies, and deployment gates. Without this component, red teaming becomes reporting rather than risk reduction.

## 17.3 Design Tradeoffs

*Coverage vs. computational cost* – Broad stress testing finds more unknowns, but trusted-reference evaluation can be expensive. In practice, most teams should use a tiered funnel: cheap broad generation first, then increasingly expensive validation on the most concerning cases. This is often the only way to search widely without exhausting simulation or expert-review capacity.

*Adversarial realism vs. attack strength* – Unconstrained attacks expose weaknesses efficiently, while constrained attacks are more physically relevant. Scientists should decide whether the goal is to find any brittle behavior or to estimate operational risk under plausible conditions. Early development often benefits from stronger unconstrained attacks; pre-deployment decisions should usually emphasize attacks that respect domain constraints.

*Automation vs. expert oversight* – Fully automated loops scale well, but expert review can catch subtler domain-specific failure modes. A good landing point is to automate generation, first-pass scoring, and clustering, while reserving human attention for high-severity, novel, or ambiguous cases. This is especially important when the cost of misclassification is high.

*Synthetic failures vs. drift from reality* – Heavy reliance on generated scenarios can overfit the red-team process to artificial weaknesses. Teams should routinely anchor the corpus with real incidents, operator reports, historical edge cases, and naturally occurring difficult examples so the adversarial program does not become detached from field conditions.

*Reproducibility vs. storage burden* – Full retention improves auditability, while compressed storage or seed-only approaches reduce cost but make forensic analysis harder. For severe findings, full replay bundles are usually worth the storage. For low-risk exploratory failures, seed-based regeneration may be sufficient if the environment is stable enough to recreate the case.

## 17.4 Implementation Advice

- Use tiered stress testing to control cost.
  Start with inexpensive generators and fast approximate checks, then promote only suspicious cases to high-fidelity simulators, full pipeline runs, or expert review. In practice this often means combining broad perturbation sweeps, prompt variants, or sampled scenario search with a smaller second stage that uses trusted physics codes, safety rules, or panel review. The method keeps coverage high without treating every test case as equally expensive.
- Constrain adversarial generation with domain rules when operational realism matters.
  Encode physical bounds, allowed actuator ranges, instrument tolerances, legal parameter combinations, or workflow invariants directly into the generator. Common techniques include rule-based filters, constrained optimization, rejection sampling, grammar-constrained generation, and simulator-in-the-loop validation. This prevents the team from spending most of its effort on failures that cannot occur in the real system.
- Define multidimensional risk metrics.
  Do not collapse all failures into a single scalar too early. Track at least severity, confidence, reproducibility, novelty, exploitability, and expected operational frequency. A simple table or structured JSON schema for findings is often enough. Separate dimensions help teams make better decisions about whether a case should trigger retraining, a policy change, additional instrumentation, or a release block.
- Feed failures directly into retraining and data augmentation.
  A red-team finding should become a concrete artifact in the development loop: a new regression test, an additional training example, a prompt guardrail case, or a calibration sample. If the pipeline supports it, label failures by type and route them automatically to the right mitigation path. This is what converts red teaming from an evaluation exercise into a system-improvement mechanism.
- Maintain a versioned red-team ledger.
  Store the model version, prompt or policy version, scenario seed, generation method, evaluator version, environment details, and adjudicated outcome for every important case. A lightweight database or versioned artifact registry is sufficient if it supports filtering and replay. The ledger matters because teams otherwise lose the ability to prove that a weakness was fixed, to compare releases fairly, or to investigate regressions months later.
- Split fast checks from deeper scheduled evaluations.
  Some adversarial checks belong in every commit or nightly run; others are too expensive and should run weekly, before release, or before deployment to a new environment. This split keeps the feedback loop fast for developers while preserving a deeper assurance layer. A practical pattern is smoke red-team suite in CI, expanded suites on schedule,

and full campaign-style evaluation at major release gates.

**Questions for Future Expansion**
Which concrete GRID, MOAT, or Prometheus workflows should be cited as examples of adversarial scenario generation or safety gating?
What trusted evaluators are actually used in the repository context: simulators, rule engines, expert review boards, or reference datasets?
Are there domain-specific risk dimensions that should be standardized across teams rather than defined locally?
What minimal replay bundle is sufficient for reproducibility in this environment: full artifacts, seeds plus environment hashes, or both?
Which red-team checks are cheap enough for CI and which require scheduled or release-gate execution?

**Figure Suggestions**
Primary figure: Red-team feedback loop showing scenario generation, trusted evaluation, multidimensional risk scoring, hard-example ledger, and retraining or gating updates.
Optional figure: Risk matrix heatmap organizing findings by severity and reproducibility or exploitability, with representative failure classes annotated.

## 17.5 Relationships to Other Patterns

- Use with *Safety-Critical Systems* when the cost of unknown failure modes is high enough to justify active stress testing.
- Use with *Iterative Correction / Guardrails Patterns* when red-team findings should directly produce validators, escalation triggers, and release gates.
- Use with *Cyber-Physical Systems* when realistic stress cases must reflect process dynamics and side effects.
- Use with *Data Pipelines and Collection Patterns* when hard examples should be preserved for retraining and regression.
- Use instead of *Iterative Correction / Guardrails Patterns* when the main need is discovering unknown weaknesses rather than controlling already-understood ones in operation.

# 18. Agent-as-User Delegation

## 18.1 Problem

As AI agents move from advisory prototypes into real scientific workflows, they increasingly need to read data, invoke tools, submit jobs, stage analyses, and sometimes request actions that affect shared infrastructure. A common but dangerous shortcut is to give the agent a privileged service identity or broad automation permissions that sit outside the normal access-control model used by people. That approach is tempting because it simplifies integration, but it creates a governance gap: actions become hard to attribute, permissions become hard to scope, and the system can bypass the operational boundaries that protect facilities, data, and users.

Naive solutions fail in two main ways. One is to treat the agent as a superuser, with blanket access to tools and data that no single human would normally possess. This expands the blast radius of mistakes, compromises least-privilege principles, and makes it difficult to explain why a particular action was authorized. The other is to avoid agent execution entirely because authorization is too hard, leaving the workflow stuck in manual copy-and-paste mode. In that case, the organization preserves safety but loses much of the operational value of agentic systems. Scientific facilities and multi-user research environments need a middle ground: a way for agents to act usefully without becoming an ungoverned class of actors.

The core problem is therefore not only whether an agent is technically capable of performing a task, but whether it can do so through bounded, auditable authority that fits existing institutional rules. This pattern generalizes wherever AI agents interact with shared resources, controlled data, facility systems, or time-bounded user entitlements.

## 18.2 Characteristics of the Solution

The recurring solution is to treat the agent as a delegated extension of an authenticated user, approved role, or facility-managed service rather than as independent privileged automation.

A first component is *delegated identity*. The workflow binds agent actions to a user, group, campaign, or approved service identity whose authority is already meaningful within the

organization. This matters because every action should inherit a recognizable accountability context rather than originating from an opaque automation account.

A second component is *scoped permissions*. Delegation should be limited by role, time window, resource class, operation type, and workflow stage. This matters because an agent may need permission to query metadata, launch a job, or write to a logbook, but not to modify safety interlocks or access unrelated data.

A third component is *policy mediation*. Agent requests should pass through the same policy engines, approval logic, and audit layers that govern human access, sometimes with additional checks for autonomy level or workflow risk. This matters because agents are most trustworthy when they reuse institutional control points rather than bypass them.

A fourth component is *auditable delegation chains*. The system records who delegated authority, what scope was granted, when it expires, and which actions were taken under that grant. This matters because post-incident review depends on being able to reconstruct not only what happened, but under whose authority it happened.

A fifth component is *revocation and expiry*. Delegation should be easy to terminate, time-bound by default, and aligned with campaign, beamtime, or project boundaries. This matters because stale authority is a major source of operational and security risk.

Finally, mature systems support *facility-managed broker services*. Instead of every user configuring direct agent access to every tool, a governed intermediary service can enforce policy, shape requests, attach context, and simplify auditability. This matters because centralized mediation can reduce both duplication and accidental over-permissioning.

## 18.3 Design Tradeoffs

*Delegation precision vs. usability* – Narrow, time-bounded permissions improve safety and audit quality, but they can make setup cumbersome. Scientists should prefer the least additional complexity that still preserves clear authority boundaries, often by extending familiar role-based access controls rather than inventing fully new authorization concepts.

*User accountability vs. facility control* – Direct user delegation preserves clear ownership, while facility-managed service identities can be easier to standardize and secure. The right choice depends on whether the workflow is primarily personal, team-based, or facility-operated. In many cases, user-scoped requests mediated by a facility service provide the best balance.

*Static permissions vs. dynamic workflow scope* – Fixed permissions are easier to audit, while workflow-aware delegation can be more precise. For example, beamtime-bounded or campaign-bounded permissions may be safer than indefinitely broad lab access. The challenge is to keep dynamic scoping understandable enough that users and operators know what the agent can do at any moment.

*Central mediation vs. local flexibility* – Brokered access simplifies governance, but can slow experimentation if every new tool requires central integration. A practical compromise is to define a small, reusable delegation framework while allowing local tools to adopt it incrementally.

*Security rigor vs. automation value* – Stricter delegation controls reduce risk, but too much friction can force users back into unsafe manual workarounds or prevent useful automation entirely. The right landing point is one where agents remain productive while clearly staying inside existing institutional trust boundaries.

## 18.4 Implementation Advice

- Reuse existing identity and authorization systems whenever possible.
  Do not create a parallel trust model for agents unless there is a compelling reason. Extend existing authentication, role-based access control, project scoping, and approval frameworks so agent authority remains legible to operators, auditors, and users.
- Delegate least privilege with explicit scope and expiry.
  Grant only the capabilities needed for the workflow step at hand, and bind them to a clear time window, campaign, or operational context. Examples include read-only access by default, beamtime-bounded action scopes, and write permissions that require separate approval.
- Record the full delegation chain.
  Log who authorized the agent, what scope was granted, what tools were invoked, what data were accessed, and which actions were actually executed. These records should be first-class provenance artifacts rather than incidental debug logs.
- Prefer mediated execution for high-impact actions.
  For consequential actions, route execution through a facility-managed broker, policy engine, or workflow controller that can enforce additional checks, attach audit context, and deny unsafe requests. This is usually safer than giving agents direct credentials to critical systems.
- Make revocation and emergency stop straightforward.
  Delegated authority should be easy to suspend or terminate when a workflow changes, a user leaves, an incident occurs, or a model begins behaving unexpectedly. Revocation paths should be tested rather than assumed.
- Separate delegation from correctness.
  An agent having permission to do something does not mean it is correct to do it. Authorization should work alongside validation, safety checks, simulation, and human approval rather than replacing them.

**Questions for Future Expansion**
Which facilities already support agent-like delegation through scoped tokens, broker services, or role-based access extensions?
What should the minimum delegation record contain for scientific workflows: delegator, scope, expiry, tool set, workflow ID, and action log?
Where should delegation be user-scoped, team-scoped, or facility-scoped?
Which classes of actions should always require mediated execution even when delegation exists?
How can revocation and emergency stop be standardized across heterogeneous facilities and tools?

**Figure Suggestions**
Primary figure: Delegation chain and policy mediation diagram showing user, delegated scope or token, broker or policy engine, agent, and downstream tools or resources, with

expiry, audit logging, and revocation points marked.
Optional figure: Permission scope matrix comparing which actions are allowed for the user, delegated agent, brokered service, and administrators across reads, job submission, logbook writes, and high-impact control changes.

## 18.5 Relationships to Other Patterns

- Use with *Versioning and Change Management* when agent actions need to be tracked in addition to the user, especially for code changes, workflow execution, approvals, or configuration updates.
- Use with *Cross-Facility Coordination and Conformance* when delegated actions must work consistently across organizational APIs, facilities, or service boundaries.
- Use with *AI-Augmented Software Development* when delegation appears in software contexts such as preparing pull requests, updating build configuration, applying code fixes, or running repository workflows on a user's behalf.

# 19. Federated and Confidential Data Patterns

## 19.1 Problem

Many teams need to train useful AI models across high-value datasets that cannot be fully centralized because of privacy, policy, security, ownership, export-control, or institutional restrictions. The AI-specific problem is that the scientific value often lies in combining distributions that are fragmented across sites, instruments, or organizations. No single site has enough coverage, but the collaboration cannot simply pool raw data into one shared repository. The result is a tension between model quality and permissible data movement: the model needs joint learning signal, while the institutions need local control, access boundaries, and evidence that confidential material never leaves approved environments.

Naive solutions fail in predictable ways. Full centralization can violate policy or trust agreements, create a single attractive breach target, and trigger long approval cycles that stall science. The opposite naive solution, keeping each site fully separate, usually produces brittle local models that do not generalize well and do not benefit from cross-site diversity. A third failure mode is assuming that federated learning is automatically safe once raw records stop moving. In practice, gradients, embeddings, checkpoints, and evaluation outputs can still leak sensitive information, and sites often differ in hardware, bandwidth, label quality, modality availability, and governance maturity. These differences matter scientifically because they change who can participate, how often synchronization happens, and whether the resulting global model overfits the best connected or most powerful institutions. That is why the pattern is especially relevant for FAMOUS and partially relevant for Axess and Prometheus: the shared challenge is not merely distributed computation, but useful collaboration under confidentiality constraints. FAMOUS has several datasets that characterize unique samples or are collected under unique conditions and consequently are institutionally protected until they have been analyzed, published, and released to the public.

Real grid data are protected under a combination of privacy, institutional policy, proprietary ownership, and national security constraints, so we currently train on publicly available synthetic datasets (some of which are statistically modeled on real grid data). GRIDAI

team are working toward secure computing environments for utility data — for example, utilizing CITADEL at OLCF — both for training and for prototyping federated learning with utilities, and our federated learning package APPFL already implements differential privacy and secure aggregation techniques. The data from real utilities and synthetic grids can be highly non-IID across topologies, load conditions, and dataset formats, with schemas and problem types varying further as we expand the model to cover additional grid problems. To match this diversity, we envision a multi-head architecture and are actively developing in that direction. A key governance gap going forward is defining what privacy guarantees look like and what utilities require in order to participate with real-world data, and then developing the corresponding techniques, tools, and agreements to meet those expectations.

## 19.2 Characteristics of the Solution

The shared pattern keeps raw data local and moves model state, summaries, or carefully constrained intermediate representations rather than source data.

A first component is *local training or local adaptation at each site*. Each institution computes updates inside its own security boundary. This matters because it preserves policy compliance and allows each site to use local storage, approvals, and domain preprocessing without surrendering custody of the underlying data.

A second component is *coordinated aggregation*. A central or hierarchical coordinator combines local updates into a shared model, often using weighted averaging, secure aggregation, or asynchronous participation. This matters because the collaboration still needs a mechanism to learn from distributed evidence and produce a coherent global artifact.

A third component is *privacy and confidentiality controls*. Depending on sensitivity, these may include secure enclaves, secure multiparty aggregation, differential privacy, update clipping, encryption in transit and at rest, and strict metadata minimization. This matters because data protection is not achieved by locality alone; the update channel itself must be defended.

A fourth component is *site-aware validation*. Performance is measured globally and per site, often with stratification by modality, cohort, or instrument. This matters because a model that looks good on pooled metrics may fail at minority sites or on rarer modalities that matter scientifically.

A fifth component is *checkpointing and rejoin support*. Sites may miss rounds, reconnect later, or operate on uneven schedules. This matters because real federated collaborations are operationally messy, and the workflow must tolerate partial participation without corrupting lineage.

A sixth component is *modular architectures when modalities differ*. Separate encoders, adapters, or shared trunks with site-specific heads allow institutions with different sensing capabilities to contribute meaningfully. This matters because a single monolithic input schema often excludes weaker or specialized sites.

## 19.3 Design Tradeoffs

*Communication overhead vs. convergence speed* – Frequent synchronization reduces model drift across sites and can speed convergence in stable networks, but it can dominate runtime when links are slow or expensive. Scientists should synchronize more often when data are

highly non-IID and sites are reasonably balanced, and less often when bandwidth is the limiting factor.

*Privacy guarantees vs. model fidelity* – Stronger privacy protections, such as heavier noise injection or stricter clipping, reduce leakage risk but can degrade performance, especially on small, rare, or high-value classes. The right point depends on the true sensitivity of the task and whether the collaboration can tolerate some quality loss in exchange for broader participation.

*Synchronous fairness vs. asynchronous practicality* – Synchronous rounds can be easier to reason about and may feel fairer, but they let the slowest site determine progress. Asynchronous aggregation increases throughput and resilience, but may overweight fast sites unless participation is normalized carefully.

*Global uniformity vs. local specialization* – A single shared model is simple to manage and compare, but local fine-tuning may be necessary when instruments, populations, or operating regimes differ significantly. The key scientific question is whether differences across sites are nuisance variation to be averaged out or meaningful variation that should be modeled explicitly.

*Monolithic models vs. modular modality support* – Unified models reduce orchestration complexity when all sites look similar. Modular designs better accommodate missing modalities and uneven capability, but they increase alignment work, checkpoint complexity, and evaluation burden.

## 19.4 Implementation Advice

- Keep raw data local by default.
  Treat centralization as an exception requiring explicit justification, not the baseline. Build training and evaluation flows so that the default unit of movement is a model update, summary statistic, or approved intermediate artifact. This makes later governance reviews much easier because the system architecture already aligns with least-privilege principles.
- Use secure aggregation and privacy controls appropriate to data sensitivity.
  Match protection strength to actual risk. For moderately sensitive collaborations, encrypted transport, access-controlled coordinators, and update retention limits may be enough. For stronger confidentiality needs, add secure aggregation, clipping, differential privacy, or trusted execution environments. The important engineering habit is to document the threat model and choose controls against that model rather than applying privacy techniques as branding.
- Validate both globally and per site after aggregation rounds.
  Always inspect performance by institution, modality, and important subpopulation. A federated model can improve the global mean while materially harming one participating site. Use dashboards or reports that show round-by-round site metrics, calibration, and error types so scientists can decide whether the aggregated model is genuinely collaborative or merely dominated by a few participants.
- Plan explicitly for uneven bandwidth, compute capacity, and modality coverage.
  Do not assume all sites can contribute equally. Introduce configurable local epoch counts, update compression, asynchronous participation, and minimum-capability participation tiers. These details often decide whether smaller or more constrained

sites remain in the collaboration long enough to contribute scientific value.

- Version checkpoints and support rollback.
  Keep versioned global checkpoints, local checkpoints, aggregation policies, and participation logs so that failed rounds can be analyzed and reversed. Useful mechanisms include model registries, immutable object storage, signed manifests, and round metadata tables. Without explicit rollback, a bad aggregation or policy mistake can contaminate the collaboration and be difficult to reconstruct.
- Prefer modular model designs when sites hold different modalities.
  If one site has imaging data, another has time series, and another has only metadata, a single rigid model often either excludes contributors or forces weak imputation. Modular encoders, adapters, or mixture-style architectures give sites an entry point consistent with their actual holdings while preserving a shared learning objective.

**Questions for Future Expansion**

What concrete confidentiality constraints in FAMOUS, Axess, or Prometheus should be named explicitly: privacy, institutional policy, export control, clinical sensitivity, or proprietary ownership?

Are there repository examples of secure aggregation, enclave use, or differential privacy that could be referenced instead of describing them generically?

How non-IID are the participating datasets in practice, and are there observed fairness or representation issues across sites?

Which modalities differ by site, and would adapter-based or multi-head architectures better match the real workflows?

What governance artifacts already exist, such as data use agreements or audit logs, that could be tied to the technical recommendations here?

**Figure Suggestions**

Primary figure: Federated learning architecture showing several institutions training locally, protected update exchange, central or hierarchical aggregation, and privacy or confidentiality controls.

Optional figure: Per-site validation dashboard or chart showing global performance alongside site-specific metrics to illustrate why pooled averages are insufficient.

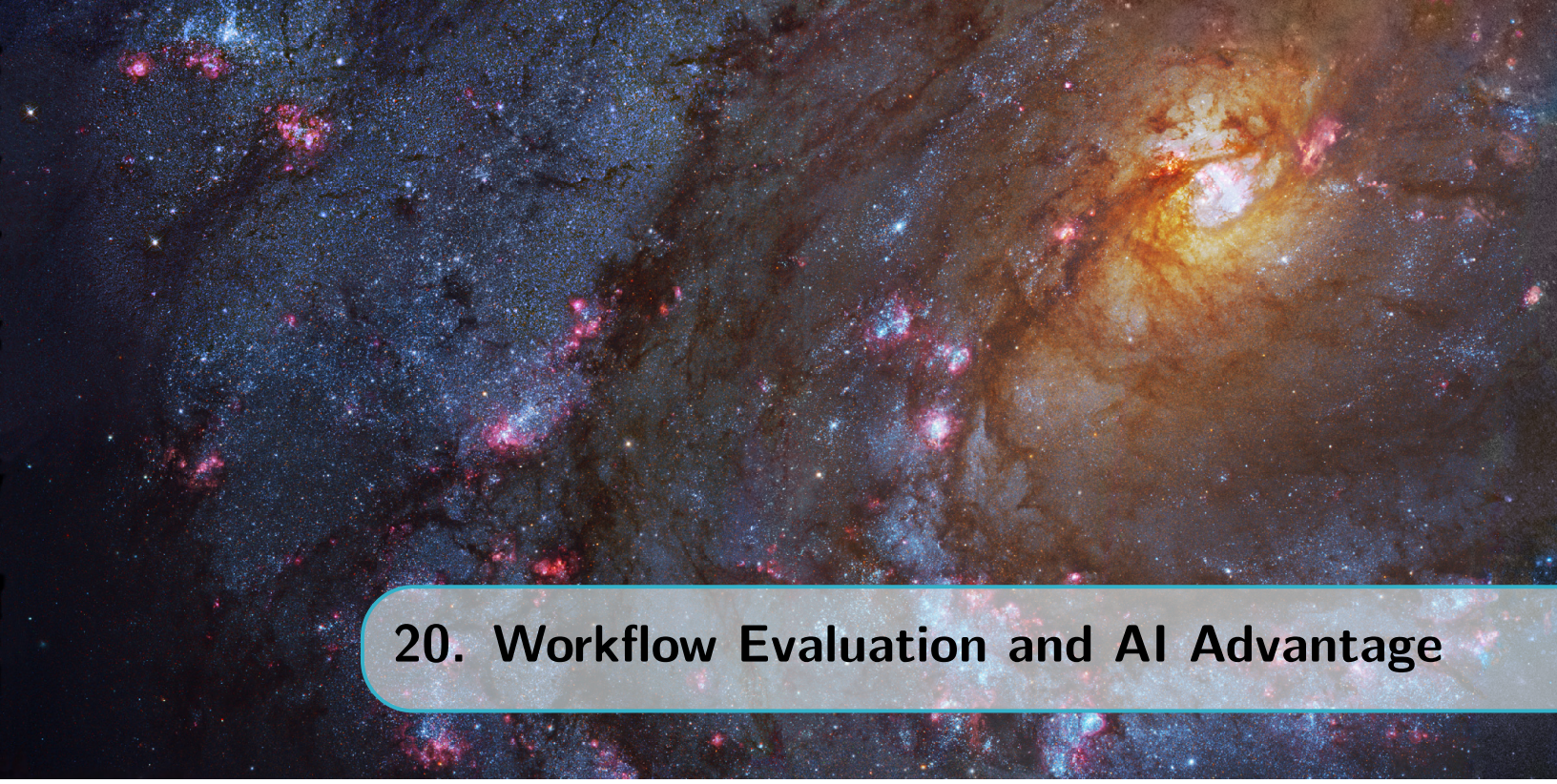
## 19.5 Relationships to Other Patterns

- Use with *Data Pipelines and Collection Patterns* when the shared challenge is protecting, packaging, and governing the data products and metadata emitted by pipelines across restricted environments.
- Use with *Physics-Informed AI Patterns* when shared scientific structure helps align learning across sites even though raw data cannot be pooled.
- Use instead of *At-Scale AI Training and Inference* as the primary framing when the hard problem is restricted data movement, privacy, or ownership rather than raw compute scale.

# Part VI

# Assessment Patterns

Assessment patterns sit at the level of the overall workflow rather than any single component. This grouping captures how teams determine whether AI is delivering real scientific or operational advantage once integrated into practice. Ending with assessment reinforces a central idea of the report: the point of these patterns is not only to build AI-enabled workflows, but to evaluate whether they improve outcomes in ways that are measurable, trustworthy, and worth sustaining.



# 20. Workflow Evaluation and AI Advantage

## 20.1 Problem

AI-enabled scientific workflows are often evaluated as if the only relevant question were whether a model is accurate on a benchmark. In practice, that is rarely enough. Scientific users care whether the workflow reaches a feasible answer faster, reduces the number of expensive experiments, improves decision quality, lowers operator burden, broadens access to expertise, or enables tasks that were previously too slow or labor-intensive to perform at all. The AI-specific problem is that model-level metrics such as accuracy, loss, or pass@k often capture only a small piece of what makes a workflow worthwhile.

Naive evaluation approaches fail in two common ways. One is to report only model metrics and then assume workflow value will follow automatically. This misses the fact that a more accurate component can still increase latency, reviewer burden, or integration overhead enough to make the whole workflow worse. The other is to describe success only anecdotally, without a reproducible way to compare the AI-assisted workflow against a credible non-AI or lower-AI baseline. In that case, teams may know that a system feels useful without being able to show where the gains come from, when they disappear, or whether the AI is actually responsible for the improvement.

The broader challenge is therefore to evaluate AI as part of a scientific workflow rather than in isolation. That includes identifying *AI advantage*: cases where AI provides a meaningful increase in productivity, coverage, or capability compared with a realistic alternative. In some workflows, the advantage is speed. In others it is reduced experiment count, improved triage quality, lower manpower, more consistent reasoning, or the ability to search a design space that would otherwise be impractical. A useful pattern must therefore ask not only "Is the model good?" but "Where does AI materially improve the workflow, and by how much?"

GridMind demonstrates AI advantage at the workflow level: it routes a five-stage cross-domain pipeline correctly from a single natural-language request, and when an analyst adjusts one input and reruns, it recomputes only the stages affected by that change and

reuses the rest, which avoids more than half of the computation across a typical multi-query session and makes follow-up studies about twice as fast as rerunning the whole pipeline each time, while the analyst specifies and revises the study in natural language instead of assembling the toolchain by hand. The minimum evaluation template pairs orchestration correctness, verified through a per-stage hit/run trace confirming that exactly the right stages rerun when an input changes, with throughput metrics such as end-to-end wall time, stages executed versus possible, and reuse speedup. Among the candidate human-centered measures, workload reduction and time to correct action are the most practical to collect now, since both follow directly from the authoring-effort and time-to-result evidence, whereas trust and preference are better captured through a structured user study against the manual workflow, noted as the next step.

## 20.2 Characteristics of the Solution

The recurring solution evaluates the workflow end to end, compares it against credible baselines, and explicitly identifies where AI provides measurable advantage.

A first component is *task-level decomposition*. The workflow is broken into major decision points or bottlenecks such as candidate generation, retrieval, diagnosis, simulation selection, code drafting, experiment planning, or result interpretation. This matters because AI advantage is usually concentrated in specific parts of the workflow rather than spread evenly across every step.

A second component is *baseline comparison*. The AI-assisted workflow is compared against a realistic alternative: manual review, heuristic rules, a prior non-AI process, a simpler model, or a lower-autonomy workflow. This matters because improvement claims are only meaningful relative to something the organization could actually have done instead.

A third component is *workflow-level outcome metrics*. These include time to first feasible result, number of experiments or simulations required, operator effort, recovery time, throughput, decision quality, reproducibility, and user trust. This matters because the workflow may improve even when component metrics move only modestly, or may fail even when component metrics look strong.

A fourth component is *AI-advantage attribution*. The team identifies which observed gains are truly due to AI rather than to surrounding engineering improvements such as automation, better interfaces, or more disciplined data collection. This matters because otherwise the contribution of AI can be overstated or left too ambiguous to guide future investment.

A fifth component is *scenario-based evaluation*. Systems are tested on representative tasks, operational cases, or design briefs rather than only on abstract benchmark instances. This matters because scientific workflows often derive value from context-sensitive usefulness, not from generic benchmark dominance.

A sixth component is *evidence artifacts*. Mature evaluations preserve replayable traces, source-linked justifications, verified citations where relevant, and explicit separation between retrieved facts and generated content. This matters because deployment decisions often depend on whether the workflow can be inspected and trusted after the fact, not just on whether it scored well during a single trial.

Finally, mature implementations include *human-centered measures*. These may include expert preference, trust, usability, cognitive load, or whether the system helps users reach

correct conclusions more quickly. This matters because many AI workflows are intended to augment human work rather than replace it outright.

## 20.3 Design Tradeoffs

*Simple metrics vs. meaningful metrics* – Accuracy, latency, and pass rates are easy to collect, while workflow value is harder to define and may require user studies or scenario testing. Scientists should still invest in the harder measures when they are closer to the real objective of the workflow.

*Controlled evaluation vs. real-world realism* – Highly controlled experiments make comparisons cleaner, but can miss the interruptions, mixed-quality inputs, and time pressures of actual operation. A useful compromise is to combine controlled benchmark tasks with scenario-based evaluations drawn from real workflows.

*Short-term productivity vs. long-term capability* – Some AI systems save time immediately, while others create value by enabling broader search, improved documentation, or better future reuse. Teams should avoid choosing metrics that capture only short-term convenience when the strategic value lies in expanded scientific capability.

*Automation gains vs. AI gains* – Workflow improvements often come from both AI and surrounding engineering. It is important to separate those effects where possible so decision makers understand whether future gains require better models, better integration, or both.

*User trust vs. raw throughput* – A system that maximizes throughput but is difficult to trust may be adopted slowly or used only superficially. For expert-facing workflows, measured trust and interpretability may be as important as nominal speedup.

*Incremental gains vs. adoption thresholds* – A workflow may show statistically real improvement while still failing to justify the training, integration, governance, and operational overhead of deployment. In many high-consequence settings, the relevant question is whether the gain is large enough to overcome organizational inertia. Scientists should therefore distinguish between measurable improvement and compelling improvement.

## 20.4 Implementation Advice

- Evaluate the workflow end to end, not just the model.
  Measure outcomes that reflect what the scientific team actually cares about: time to solution, number of experiments avoided, throughput, operator burden, decision quality, and reproducibility. Component metrics still matter, but they should support rather than replace workflow-level evaluation.
- Define a credible non-AI or lower-AI baseline.
  Compare against the process that users would realistically employ without the new AI capability, such as manual review, heuristics, simpler models, or an earlier workflow version. This makes claims of improvement interpretable and helps prevent inflated estimates of value.
- Identify where AI creates advantage.
  Decompose the workflow and ask which steps gain materially from AI. The most useful targets are often bottlenecks where AI yields a significant increase in productivity, such as reducing expensive simulation count, accelerating diagnosis, shortening documentation effort, or enabling broader exploration of candidate space. Naming these

points of advantage makes it easier to justify investment and focus future optimization.

- Use representative scenarios in addition to aggregate metrics.
  Build evaluation sets from realistic workflow cases, design briefs, troubleshooting episodes, or experiment-planning tasks. Aggregate metrics are useful, but scenario tests reveal whether the system helps on the kinds of tasks that actually matter to scientists and operators.
- Preserve evidence artifacts, not only scores.
  When a workflow makes claims about correctness, usefulness, or trust, keep the artifacts needed to inspect those claims: replayable traces, linked sources, verified citations where applicable, and explicit markers distinguishing retrieved content from generated synthesis. These artifacts support domain-scientist review and make evaluations more reusable across teams.
- Include human-centered usefulness measures.
  If the workflow is intended to augment experts, measure trust, preference, ease of use, reviewer effort, and time to correct decision. These factors often determine whether a technically strong system is adopted in practice.
- Evaluate whether the gain clears the adoption threshold.
  Do not stop at showing that AI is somewhat better. Ask whether the improvement is large enough to justify integration cost, governance burden, retraining effort, and organizational change. In many operational settings, only step-function gains in throughput, recovery time, experiment count, or capability will support durable adoption.
- Distinguish AI contribution from surrounding engineering improvements.
  When possible, separate gains from the AI model itself from gains produced by workflow automation, better interfaces, cleaner data, or stronger process discipline. This helps teams understand whether the next improvement should come from better models or from better system design.

**Questions for Future Expansion**

Which survey examples provide the clearest demonstrations of AI advantage, such as fewer experiments, lower simulation count, faster triage, or major reductions in authoring effort?

What minimum evaluation template should the report recommend for workflow-level AI claims?

Which human-centered measures are most practical to collect across DOE workflows: trust, preference, time to correct action, or workload reduction?

How should teams compare against baselines when no fully comparable non-AI workflow exists because AI enables a qualitatively new capability?

Which representative scenarios could become reusable benchmarks for workflow-level evaluation rather than model-only evaluation?

**Figure Suggestions**

Primary figure: Workflow-level AI advantage decomposition showing major workflow stages, baseline effort, AI-assisted effort, and measured gains at each stage.
Optional figure: Baseline versus AI comparison dashboard showing time to solution, experiments avoided, operator effort, trust, and throughput.

## 20.5 Relationships to Other Patterns

- Use with *Surrogates for Optimization* when the real benefit question is decision quality or acceleration of search, not just predictive fit.
- Use with *Hypothesis Generation* when the value of the pattern depends on whether it leads to better experiments or discoveries.
- Use with *Advanced AI Data Search* when retrieval quality should be judged by downstream study and decision outcomes.
- Use with *Safety-Critical Systems* when workflow performance must be evaluated together with assurance costs and constraints.

# 21. Scaling Law Analysis

## 21.1 Problem

Scientific AI programs often want to know which teams should receive scarce compute for larger training runs, data ablations, or scaling-law studies. In practice, that question is not answered well by enthusiasm, model size alone, or the presence of a promising demo. The AI-specific problem is that scaling studies require several preconditions to be true at the same time: the model or workflow must actually be scalable, the training regime must permit meaningful changes in model or data size, the team must have enough data to support larger runs, and the evaluation must capture the signals needed to interpret whether extra compute helped.

Naive allocation approaches fail in two common ways. One failure mode is to treat every AI effort as if it were a scaling candidate. That collapses important distinctions between de novo trainable models, fine-tuned systems with frozen backbones, workflow teams using external frontier APIs, and agent systems whose near-term bottleneck is orchestration rather than model scaling. In those cases, allocating large training budgets for Kaplan- or Chinchilla-style experiments can be wasteful because the workflow is not actually positioned to benefit from model scaling. The other failure mode is to classify teams by informal reputation rather than by documented evidence such as parameter counts, data volume, baselines, convergence behavior, and compute instrumentation. This leads organizations to fund larger runs before they can tell whether gains come from more parameters, more data, better engineering, or simply better documentation.

The recurring challenge is therefore to establish *scaling readiness*: whether a scientific AI effort is prepared for disciplined scaling experiments and can generate evidence useful enough to justify additional compute. Across the materials in ‘data/scaling/‘, the central blockers are not usually lack of scientific ambition, but missing FLOPs reporting, unclear convergence evidence, incomplete model documentation, and confusion between trainable model programs and workflow or agent programs. A useful pattern must therefore separate readiness tiers, require a minimum evidence bundle, and connect compute allocation to the

actual scaling mechanism available to the team.

The GridAI team estimates required compute hours from training runs on representative topology data, using measured throughput together with performance metrics to estimate the number of epochs needed for convergence. The lightest-weight cross-framework, cross-hardware tracking we use is our built-in throughput measurement, which can be translated to FLOPs via the standard 6ND approximation whenever the model architecture supports it. On workflow and agent-system efforts, we agree these should be governed separately: in many cases they are dominated by token usage rather than traditional node/GPU hours, and typically run as persistent services rather than the submit-job / fetch-results pattern that current allocation policies are designed around. We do not currently have component-level models such as circuit surrogates, but see value in developing them for specific applications. On teams with strong data assets but incomplete model metadata, we view strong data as highly competitive in its own right, though the value is most fully realized when those assets are translated into a trained model.

## 21.2 Characteristics of the Solution

The recurring solution classifies efforts by scaling mechanism, requires instrumentation before major allocation, and links compute decisions to explicit ablation designs.

A first component is *tiered readiness classification*. Teams are classified into categories such as scaling-ready, data-scaling only, and remediation required or not applicable. This matters because a portfolio usually contains a mix of de novo model teams, fine-tuning efforts, workflow teams, and agent systems, and they should not all be evaluated with the same scaling lens.

A second component is *training-regime separation*. The team documents whether the primary model is trained from scratch, fine-tuned from a frozen base, or mixed. This matters because the "pre-training wall" changes what kinds of scaling studies are valid. If the base model cannot be resized, model-parameter scaling is not the right experiment and the organization should focus on data scaling or workflow evaluation instead.

A third component is *minimum scaling evidence*. Before major compute allocation, the team records parameter counts, data volume, architecture flexibility, loss definition, baseline comparisons, and at least GPU-hours or preferably FLOPs. This matters because scaling claims are difficult to interpret if the organization cannot normalize runs by compute or even confirm that the architecture can be changed systematically.

A fourth component is *convergence and baseline visibility*. The workflow preserves loss curves, validation curves, and credible baseline comparisons against earlier models, classical methods, or simpler alternatives. This matters because without these traces the team cannot tell whether a larger run under-trained, saturated, diverged, or merely repeated existing performance at higher cost.

A fifth component is *component-level decomposition*. Multi-model programs are decomposed so that a trainable foundation model can be evaluated separately from workflow, agentic, or fine-tuned components that follow different scaling dynamics. This matters because portfolio-level labels often hide the fact that one component is scaling-ready while another is not.

A sixth component is *compute-linked recommendations*. Each tier maps to a concrete next step: Kaplan-style model and data ablations for scaling-ready teams, data-ablation studies

for frozen-base systems, and remediation steps for under-documented or non-applicable efforts. This matters because the goal is not only to label teams, but to decide what the next allocation should buy.

Finally, mature implementations include *documentation closure*. Missing model cards, undocumented base models, absent parameter counts, and vague architecture descriptions are treated as readiness blockers rather than as minor paperwork issues. This matters because portfolio decisions become unreliable when basic scaling metadata is missing.

## 21.3 Design Tradeoffs

*Broad portfolio coverage vs. precise per-component assessment* – Classifying every team quickly gives program-level visibility, but hides differences between individual components inside a single team. Decomposition takes more work, yet it often reveals the actual scaling candidate.

*Fast allocation vs. instrumented allocation* – It is tempting to fund large runs as soon as a team looks promising, while FLOPs tracking, convergence logging, and model-card cleanup feel slower. In practice, the missing instrumentation often makes expensive runs much harder to interpret, so a small delay upfront can save large wasted allocation later.

*Model scaling vs. data scaling* – Some systems can benefit from larger models and larger datasets, while others are constrained to data scaling because the base model is frozen or externally hosted. Teams should not force a model-scaling experiment when the real lever is data quantity, data quality, or retrieval quality.

*Uniform criteria vs. domain-specific exceptions* – A single portfolio-wide rubric improves comparability, but some scientific teams have unusual losses, hybrid architectures, or partial scaling paths. The framework should stay consistent while still allowing component-level exceptions when the evidence supports them.

*Documentation rigor vs. scientific pace* – Requiring detailed model cards, compute metrics, and convergence artifacts increases burden on teams moving quickly. However, once compute requests become large, that documentation burden is part of scientific governance rather than optional overhead.

*Compute optimism vs. adoption realism* – Large datasets and ambitious plans make scaling look attractive, but readiness also depends on whether the code is parametric, the baselines are credible, and the metrics allow comparison across runs. Organizations should fund demonstrated readiness rather than roadmap optimism alone.

## 21.4 Implementation Advice

- Classify the scaling mechanism before allocating large compute.
  Determine whether the effort is primarily a de novo trainable model, a fine-tuning workflow with frozen backbones, a workflow team using external APIs, or an agent system whose current value comes from orchestration rather than model growth. This prevents organizations from treating every AI project as if it were eligible for the same style of scaling study.
- Require a minimum scaling evidence bundle.
  Before major allocation, collect explicit parameter counts, architecture type, training regime, data volume, loss function, baseline metrics, and compute measures. FLOPs

are best, but GPU-hours are still better than nothing. The organization should be able to answer what is being scaled, with what data, under what loss, and at what cost.

- Treat missing FLOPs and convergence traces as real blockers.
  Across the scaling analyses, lack of FLOPs reporting and missing convergence plots were recurring systemic gaps. Teams do not need perfect theory before scaling, but they do need enough instrumentation to compare runs, estimate compute efficiency, and distinguish under-training from saturation.
- Separate trainable components from workflow and agent components.
  If a program mixes de novo models, frozen frontier APIs, and agentic orchestration, evaluate those pieces separately. A team may deserve compute for one trainable component while a different component should instead be evaluated with workflow-level or agent-level metrics.
- Match the experiment design to the tier.
  For scaling-ready efforts, prescribe multi-size model and data ablations tied to reported compute. For data-scaling-only efforts, hold model size fixed and vary training data or synthetic generation volume. For remediation cases, pause large allocations and first close the gaps in instrumentation, baselines, or documentation.
- Use documentation quality as a governance signal.
  Missing model cards, undocumented base models, and unclear parameter counts are not just editorial defects. They often indicate that the team cannot yet support reproducible scaling analysis or defend a major compute request.
- Reclassify workflow teams instead of penalizing them for not being trainable models.
  Some efforts are valuable precisely because they improve coding, orchestration, analysis, or operational workflows without bespoke model training. Those efforts should move to a workflow evaluation track rather than being judged as failed scaling-law candidates.
- Review readiness again after remediation or architecture changes.
  A Tier 2 or Tier 3 effort can move into scaling readiness when it begins de novo training, exposes parametric architectures, documents baselines, or adds compute instrumentation. Treat the tier as a current operating assessment, not as a permanent label.

**Questions for Future Expansion**

Which minimum evidence bundle should the report standardize for requesting large-scale training allocation: FLOPs, GPU-hours, convergence plots, baselines, model card, or all of the above?

Should the report define a formal ”workflow team” and ”agent systems” track so those efforts are not forced into a model-scaling rubric?

What is the lightest-weight way for teams to add FLOPs tracking across heterogeneous frameworks and hardware?

Which Tier 2 edge cases should be highlighted as component-level Tier 1 candidates, such as AXESS Circuit Surrogate or Prometheus Peregrine?

How should compute governance handle promising but poorly documented teams that have strong data assets but incomplete model metadata?

**Figure Suggestions**
Primary figure: Scaling-readiness decision tree showing de novo model teams, fine-tuned frozen-base systems, workflow teams, and agent systems, with the recommended evaluation or allocation path for each.
Optional figure: Readiness matrix showing teams by tier against FLOPs reporting, convergence plots, parameter documentation, baselines, data volume, and architecture flexibility.

## 21.5 Relationships to Other Patterns

- Use with *At-Scale AI Training and Inference* when a team is scaling-ready and the main challenge becomes efficient execution across large HPC systems.
- Use with *Workflow-Level Evaluation and AI Advantage* when a project should be judged by end-to-end workflow gains rather than by model scaling potential.
- Use with *Evaluating Agentic AI* when the effort is agentic or multi-step and evaluation bottlenecks lie in coverage, criteria, or delayed rewards rather than in parameter scaling.
- Use with *Versioning and Change Management* when scaling evidence depends on reproducible run metadata, model versions, and comparable experiment histories.

# 22. Evaluating Agentic AI

## 22.1 Problem

Scientific teams often recognize that an agentic AI system looks impressive in demos while still being hard to trust, hard to compare, and hard to improve in practice. A benchmark score, a small pilot, or a few expert anecdotes may suggest promise, but those signals do not answer the questions that matter for scientific deployment: does the system cover realistic cases, are we measuring the right notion of success, can it satisfy several legitimate goals at once, is the evaluation effort scalable, and can we learn anything useful when the real payoff appears only much later? The AI-specific problem is that agentic workflows amplify all of these evaluation weaknesses at once because they act over sequences of decisions rather than producing a single isolated prediction.

Naive evaluation approaches usually fail in one of two ways. One failure mode is to treat evaluation as a conventional static benchmark problem and assume that enough held-out examples, a few aggregate metrics, and average-case reporting are sufficient. That misses coverage gaps, poorly chosen criteria, objective conflicts, and delayed consequences that only appear in realistic workflows. The other failure mode is to make evaluation so bespoke and labor intensive that the team cannot keep up with the pace of iteration. In that case, experts become the bottleneck, findings arrive too late to guide development, and the organization gradually stops evaluating the system with the rigor it originally intended.

The recurring challenge is therefore broader than model validation. Teams need a structured way to inspect five recurring LIMITs in scientific evaluation: *limited coverage*, *inaccurate or inexact criteria*, *multiple objectives*, *implementation costs*, and *time-delayed rewards*. These LIMITs appear across diverse scientific AI efforts because they reflect practical weaknesses in how evidence is collected and interpreted rather than weaknesses of any one model family. A useful pattern should help teams expose these weaknesses early, choose techniques that match the bottleneck, and build evaluation programs that remain credible as the workflow grows more autonomous.

## 22.2 Characteristics of the Solution

The recurring solution is to organize evaluation around the five LIMITs and choose explicit techniques for each one rather than relying on a single benchmark or scoring rubric.

A first component is *coverage analysis*. The team asks whether the evaluation actually spans the operational states, data regimes, workflow branches, and difficult edge cases that matter. Useful evidence may include adversarial tests, state- or path-based coverage summaries, residual plots, uncertainty estimates, extrapolation-distance diagnostics, or distribution-shift measures such as KL divergence. This matters because average metrics can hide the specific regions where deployment confidence should collapse.

A second component is *criteria validation*. The team checks whether the metrics being optimized or reported actually match scientific requirements. In many workflows, the right criterion is not raw fidelity, benchmark accuracy, or exact reproduction of the input, but preservation of diagnostically important structure, satisfaction of downstream constraints, or performance on the final scientific task. This matters because evaluation can be perfectly reproducible and still be wrong if it measures the wrong thing.

A third component is *multi-objective alignment*. The evaluation makes objective conflicts explicit instead of burying them inside a single score. Teams may need to trade off quality, speed, robustness, cost, interpretability, energy use, or compliance with scientific constraints. Common tools include thresholding, weighted objectives, Pareto fronts, and proxy instruments for correlated variables. This matters because many scientific decisions are not about maximizing one number, but about finding acceptable operating points under several competing demands.

A fourth component is *evaluation scalability*. The team designs the evaluation process so it can grow without requiring linear growth in scarce expert time. Practical methods include bootstrapping with manually authored tasks, training automatic generation and validation components, validating the validators, and tracking saturation so human effort is focused where it still changes conclusions. This matters because implementation cost often determines whether a sound evaluation plan survives beyond the first prototype.

A fifth component is *reward refactoring*. When the true value of a decision is delayed, uncertain, or expensive to observe, the workflow introduces earlier intermediate signals that are still meaningfully related to success. Depending on the domain, this may involve prediction of future scores, hierarchical decomposition into subproblems, Bellman-style value backpropagation, branch-and-bound style pruning, or Monte Carlo tree search style rollouts. This matters because agentic systems are especially difficult to improve when every useful signal arrives only at the end of a long sequence.

Finally, mature implementations preserve *evidence traces* linking difficult cases, chosen criteria, human judgments, delayed outcomes, and model or workflow versions. This matters because evaluation is not only about producing a score; it is about making later debugging, governance, and scientific comparison possible.

## 22.3 Design Tradeoffs

*Coverage breadth vs. evaluation cost* – Broader coverage reveals more failure modes, but the most realistic cases are often the most expensive to simulate or review. Most teams need tiered evaluation that uses cheap broad probes first and reserves expensive analysis for suspicious regions.

*Strict criteria vs. useful criteria* – It is tempting to demand exact fidelity or very conservative thresholds, but overly strict criteria can reject scientifically acceptable solutions and block large practical gains. Teams should derive criteria from downstream scientific needs rather than from what is easiest to score.

*Single-score simplicity vs. multi-objective honesty* – A single metric makes comparison easier, while multi-objective reporting better reflects reality. Scientists should simplify only after confirming that the compressed score does not hide important tradeoffs or non-binding criteria.

*Manual rigor vs. scalable evaluation* – Expert-designed and expert-validated tasks can be high quality, but they do not scale. Automation increases throughput, but only if the generated tasks and validators are themselves checked carefully enough to remain credible.

*Immediate proxies vs. true delayed outcomes* – Intermediate rewards speed learning and evaluation, while final outcomes remain the real target. The practical goal is not to replace delayed outcomes entirely, but to use proxies that are predictive enough to guide iteration without severing the connection to scientific value.

*Model-focused diagnosis vs. workflow-focused diagnosis* – Some LIMITs arise from the model, while others come from task design, criteria selection, human bottlenecks, or process architecture. Teams should avoid assuming that every evaluation weakness requires a better model.

## 22.4 Implementation Advice

- Test for coverage gaps explicitly.
  Do not rely on average-case accuracy alone. Inspect hard regions with residual plots, adversarial sampling, uncertainty estimates, extrapolation-distance measures, or state-space coverage summaries. When a gap is found, categorize it so the next action is clear: collect more data, change the model, constrain deployment, or redesign the workflow.
- Work backward from downstream scientific requirements.
  Ask how the output will actually be consumed by downstream codes, instruments, or experts, and derive criteria from those needs. Useful checks often include preserving maxima, spectra, rankings, threshold decisions, or diagnostically relevant structure rather than merely minimizing generic reconstruction or prediction error.
- Make objective conflicts visible before optimizing them away.
  List the goals that are simultaneously in play, identify which are hard constraints and which are optimization targets, and record when a criterion is never binding in practice. Thresholding plus one primary objective is often easier to defend than an arbitrary weighted sum, but Pareto-front views can reveal when that simplification is hiding important options.
- Budget expert time as the scarcest evaluation resource.
  Assume that narrow scientific expertise, not compute, will often be the real bottleneck. Start with a manually curated seed set, then use automatic generation and validation to scale only after the manual process is well understood. Validate the validators and monitor saturation so experts are spent on novel or consequential cases rather than repetitive ones.
- Refactor delayed rewards into earlier, inspectable signals.

When final outcomes arrive late, define intermediate milestones that still correlate with success. Depending on the workflow, this may mean scoring subgoals, predicting final utility, pruning obviously poor branches, or simulating candidate futures. Keep validating that these early signals still align with the true outcome rather than drifting into convenience metrics.

- Preserve a reusable evaluation ledger.
  Store the task or scenario, model and workflow version, chosen criteria, human judgments, uncertainty signals, and final outcomes where possible. This turns evaluation into an accumulating asset that supports regression testing, governance review, and later refinement of the criteria themselves.
- Revisit the LIMITs after major workflow changes.
  A system that passed an earlier evaluation may still fail once autonomy increases, the data distribution shifts, the objectives change, or the organization tries to scale usage. Re-running the same benchmark is rarely enough; re-check the five LIMITs whenever the workflow meaningfully changes.

**Questions for Future Expansion**
Which examples from the survey should anchor each LIMIT most clearly, especially limited coverage and delayed rewards in real scientific workflows?
Should the report recommend a lightweight checklist so teams can score themselves against the five LIMITs before deployment?
Which techniques for validating automated evaluators are practical enough to recommend as a baseline across projects?
How should the report distinguish between inexact criteria that are acceptable abstractions and inaccurate criteria that systematically mislead decisions?
What minimum evidence bundle should teams preserve so later readers can audit how the evaluation handled each LIMIT?

**Figure Suggestions**
Primary figure: The five LIMITs as an evaluation framework, with each LIMIT mapped to representative failure modes, techniques, and example evidence artifacts.
Optional figure: Evaluation-scaling funnel showing manual task creation, automated generation, automated validation, validator checking, and saturation monitoring.

## 22.5 Relationships to Other Patterns

- Use with *Workflow-Level Evaluation and AI Advantage* when the question is not only whether evaluation is sound, but whether the AI-assisted workflow produces meaningful advantage over a credible baseline.
- Use with *Red Teaming and Adversarial Patterns* when limited coverage should be addressed through active search for hard or unknown failure cases.
- Use with *Iterative Correction / Guardrails Patterns* when delayed rewards or uncertain criteria require intermediate checks, validators, and staged acceptance.

- Use with *Hypothesis Generation* when multi-objective selection and delayed scientific feedback shape which candidates are worth testing next.
- Use with *Surrogates for Optimization* when evaluation must balance predictive fit, downstream decision quality, and the cost of collecting more ground truth.

# Part VII

# Backmatter

# 23. Changelog

- **0.1** Initial set of 14 patterns collected from the model seed teams
- **0.2** Additional 5 patterns identified from the broader DOE-wide workflow survey
- **0.3** Identified connections between patterns and reorganized them into thematic categories
- **0.4** Identified 2 additional evaluation patterns
- **0.5** Identified 1 additional reasoning and discovery pattern
- **0.6** Sent to model teams via ambassadors for initial review