Table of contents

"Talk is cheap. Show me the code." —Linus Torvalds

1 Introduction

Many valuable research artifacts are absent from modern libraries—not because they lack importance, but because current systems struggle to accommodate them. Designing a library is a forward-looking endeavor: it requires anticipating how information will be interpreted, reused, and built upon in the future.

This work is part of a broader agenda: identifying knowledge worth preserving—especially that which supports reuse and transparency—and rethinking how libraries must evolve to capture it. It builds on prior efforts advocating for the preservation of reasoning and experimentation, not just final outcomes (Oderinwale, 2025).

Procedural Knowledge Libraries (PKLs) represent a concrete step toward this vision. PKLs are structured, versioned representations of procedural knowledge. They capture not only what was done but also how and why—organized into modular units that can be queried and reused across domain-specific workflows.

Capturing a *process*—a sequence of intentional actions as they unfold—requires real-time documentation that remains aligned with execution. Because steps evolve, goals shift, and edits are made, the most faithful archives treat the process as a dynamic artifact. Effective documentation, then, must go beyond recording outcomes to include the rationale behind them.

To illustrate the distinction between *process* and *procedure*, and between *information* and *knowledge* (Aamodt & Nygård, 1995), consider the contrast between a lab protocol and a researcher's notebook. A protocol describes fixed, validated steps; the notebook captures informal observations, failed attempts, and evolving reasoning—the researcher's working understanding of how to adapt those steps. Put simply: a *process* is to a *protocol* as a *procedure* is to a *notebook*.

Similarly, a recipe lists ingredients and instructions, but a chef draws on procedural knowledge to adjust for taste, freshness, and timing under uncertain conditions (Parkinson et al., 2012). Or compare a software tutorial with learning an experienced developer's workflow through pair programming: the tutorial offers a static path, while the workflow reveals how to navigate uncertainty, debug creatively, and optimize using tacit heuristics (Gregory & Lindsay, 2016).

These omissions—of intent, trial-and-error, and tacit judgment—stymie collaboration, reproduce redundant effort, and obscure provenance. PKLs address these gaps by enabling earlier-stage collaboration, structured reuse, and clearer methodological accountability. The goal is to construct documentation that captures procedural knowledge: how next steps were inferred, why paths were abandoned, and what informed decisions beyond the trace—where *traces* refer to recorded actions.

2 Defining Procedural Knowledge Libraries

In considering the impact of PKLs, a question arises: what is documentation? Especially within the context of Procedural Knowledge Libraries, it's essential to recognize that documentation serves multiple purposes. It can describe the features of an object, provide instructions for its use, or detail its functions. However, in the realm of procedural knowledge, documentation should aim to capture not only the steps involved in a process but also the underlying reasoning, decision points, and adaptations made during its execution.

Procedural knowledge refers to an agent's understanding of how to accomplish specific tasks through structured actions. Unlike conceptual knowledge, which concerns abstract principles and generalizable ideas (McCormick, 1997), procedural knowledge consists of executable steps tied to concrete outcomes (Byrnes & Wasik, 1991b). I distinguish between procedural information—raw sequences of actions—and true procedural knowledge, which integrates both the steps themselves and their contextual purpose in a domain.

Thus, a key characteristic of procedural knowledge is its goal-oriented organization. For instance, directions become meaningful procedural knowledge only when connected to a destination. This contextual grounding enables domain-specific problem solving while making the knowledge challenging to transfer—procedures are often internalized through unique experience rather than explicit documentation. Capturing such knowledge would represent a significant advance in preserving tacit expertise.

A complete understanding of procedural knowledge requires examining both successful and failed processes. While metrics can evaluate a procedure's effectiveness, even unsuccessful attempts contain valuable insights for those who can interpret them. This perspective informs our approach to procedural knowledge libraries, where documenting the relationship between methods and outcomes–including dead ends—building more than just repositories, but libraries that can be used to construct maps of problem-solving instances.

Procedural knowledge forms the backbone of all goal-directed systems, yet capturing and reusing it remains both challenging and often overlooked. The core difficulty lies in developing encodings that preserve not only the steps executed but also their contextual rationale—transforming raw execution traces into reproducible, interpretable workflows with their underlying reasoning intact (Elsaka, 2017). While some processes follow clearly articulated hypotheses, others may emerge from intuition or ad-hoc judgment. PKLs cannot directly capture tacit knowledge, but the data they contain and the clarity they bring to a process can make it easier to discern between intentional, iterative decisions and outcomes shaped by intuition, serendipity, or accident.

This challenge is particularly important because, as psychological research demonstrates, procedural learning is inherently more difficult than conceptual learning (Byrnes & Wasik, 1991a). The implications are universal: whether

in scientific research, technical workflows, or organizational operations, the ability to encode, retrieve, and compare procedural knowledge could unlock new paradigms in how one documents, teaches, and builds upon processes.

To address this, I propose infrastructure that captures procedural knowledge in its full context—the how and why behind methodological choices—transforming isolated expertise into reusable, composable units.

3 Contributions

PKLs serve as containers for domain-specific expertise, designed for transfer and reuse—not just as version control trees, but as semantically enriched structures with context-aware annotations on each commit.

This paper makes three contributions: (1) it defines Procedural Knowledge Libraries (PKLs) as a conceptual framework for capturing and reusing process-level knowledge in computational workflows; (2) it introduces a lens-based abstraction for selective encoding and transformation of procedural units; and (3) it proposes a practical system architecture for PKLs, including a prototype storage schema, patch format, and a sketch of the querying system. PKLs are presented as a conceptual schema, a working prototype, and a candidate standard for representing procedural knowledge—each contribution reflecting a different level of maturity in the framework's development.

This paper targets practitioners, tool builders, and infrastructure designers seeking to expand procedural documentation capabilities. It provides both conceptual foundations and a system sketch to inform future tooling and research.

3.1 Completeness & Reconstruction

Completeness in PKLs is relational, not absolute. A PKL artifact is considered complete if a domain-qualified interpreter can reconstruct the procedure in a way that aligns with its original intent. Failures in reconstruction may stem from the artifact itself or from gaps in the interpreter's contextual knowledge (Cohen et al., 2024).

To clarify this, we define a *qualified interpreter* as one with access to domain knowledge equivalent to that of the original author. For example, two interpreters executing the same plotting script with different datasets will produce different outputs—not due to an incomplete procedure, but due to divergent contexts.

Because interpreter knowledge is hard to measure directly, we compare the PKL artifacts they generate. Discrepancies in outputs may reveal missing assumptions or dependencies. PKLs mitigate this by being referential by design—each procedural step is linked to its semantic and temporal context, enabling debugging of where understanding or fidelity breaks down.

4 Relevant Work

Research on episodic memory (EM) offers insight into how experiences are stored and retrieved in the brain. Zeng et al. show that one function of EM is using past experiences to prepare for future events (Zeng et al., 2023), directly informing our understanding of cognitive replay—an ability that PKLs aim to support. This connects to the concept of *episodic control* (Blundell et al., 2016; Ciatto et al., 2025), where discrete experiences can be retrieved and reused as structured, standalone units.

In distributed systems, Kleppmann's framework for event classification establishes valuable structural principles (Kleppmann, 2021). The analysis of event ordering—distinguishing between partial and total order—along with an examination of time-boundedness and persistence characteristics, provides concrete parameters for modeling procedural flows.

In ML, complementary approaches have been developed using Procedural Knowledge Ontologies (PKOs). Carriero et al. (Carriero et al., 2025) demonstrate how standardized representations make interoperability easier, extending the Linked Terms Methodology for ontology development (Poveda-Villalón et al., 2022). These efforts help create common formats for describing procedural knowledge across different fields.

Further, work by Fan et al. outlines the use of "chunked sub-routines" to supplement an agent's library of "primitive concepts" in a process termed "structured library learning." They find that such examples establish a foundation for more efficient learning in novel systems—enabling agents to adapt to new environments more effectively by building on initial library of learned processes. Over time, agents develop "procedural abstractions," referring to increasingly larger fragments of steps, thereby making communication more efficient (McCarthy et al., 2021).

Collections of codified processes have also been studied as a means to support compositional learning (Felice, 2022), as studied in representation learning, by decomposing workflows into modular, reusable units—allowing models or agents to generalize across tasks by recombining known procedures in novel ways (Chang et al., 2019).

Furthermore, ML model architectures PRISM (PRocedure Identification with a Segmental Mixture Model) leverage hierarchial Bayesian reasoning to recover procedural abstractions from time-series data in an unsupervised manner (Goel & Brunskill, 2019). By contrast, this work emphasizes determinism and explainability, prioritizing explicit encodings and reproducible transformations over probabilistic inference.

4.0.1 PKLs for Post Hoc Planning

PKLs can be understood as a structured analogue to the reverse process of dynamic plan synthesis (Abe et al., 2025; Acharya et al., 2024)—the real-time

construction or adjustment of a plan in response to unfolding conditions. In contrast, PKLs focus on generating post hoc representations of procedural logic across a wider range of loosely specified situations. Where dynamic plan reconstruction seeks to infer intent and structure from observed agent behavior, PKLs aim to capture the procedural logic behind complex human tasks.

Relatedly, "model diffing" is an interpretability technique where researchers compare two versions of a model (e.g., a base model and a fine-tuned variant) to identify meaningful differences in behavior or representation (Ameisen et al., 2025; Shah et al., 2022). What sets interpretability through model diffing and PKLs apart is the stage of model processing they target. Model diffing focuses on changes in weights, features, and internal representations, whereas PKLs operate at a higher level of abstraction, capturing human-readable procedural structures like method, parameter, and design choices.

Despite these differences, the two probing methods are compliments of each other. Model diffing reveals internal changes such as activation shifts or architectural rewiring, while PKL-style diffs capture shifts in intent, strategy, or framing. Both can support 'interpretability,' but their units of analysis differ: numerical changes in internal representations versus compositional changes in decision logic. While model diffing traces how a model evolves, PKLs clarify how procedural knowledge develops—and how it can be reused, adapted, or audited. Together, they offer a stronger foundation for model oversight.

5 PKL System Architecture

The core functions of the PKL can be broken down into three (encoding, storage, and retrieval) and give exposition via case-studies and reflection on them.

5.1 Encoding

What makes a procedural representation meaningful? For PKLs, *encoding* refers to transforming raw procedural traces—such as code cells or pipeline steps—into structured representations that preserve context and support reusability. This section defines what makes an encoding meaningful and how PKLs differentiate between intent and outcome.

5.2 Definitions

- **Lenses:** Bidirectional transformations that isolate, extract, or reinterpret (procedural) components "between a database and a view of it." (Fong & Johnson, 2019; Gottlob et al., 1986) In PKLs, lenses allow for chosen views, representations, or rewrites of workflows while preserving semantic consistency and enabling composition.
- **Procedure:** A versioned, structured representation of a process. In PKLs, a procedure captures the steps, context, and rationale behind a task or

workflow, including its evolution over time and potential for reuse or reinterpretation.

- **Patches:** In Git, a patch is the delta between two repository states—typically the difference between two commits or working tree snapshots. These are usually expressed as line-based text diffs showing which lines were added ('+'), removed ('-'), or modified (Joshy & Le, 2020). While commonly associated with version control, the concept generalizes to any structured set of differences between states. A collection of patches represents the transformations needed to move a repository from one state to another.
- **Unit:** The smallest traceable and meaningful element of a procedure. In PKLs, a unit corresponds to an atomic step—such as a code cell, function call, or task—that can be versioned, transformed, or composed within a larger workflow.
- **File:** A named unit of data (often a collection of subunits) in a storage system. In PKLs, a file represents a procedural element—such as code, configuration, or output—that forms part of a reproducible workflow.
- **Directory:** A container that holds files and/or other directories in a hierarchical structure (for organizational purposes). In PKLs, directories organize procedural artifacts, collections of structured steps or shareable workflows.

5.3 Granularity & Interoperability

Processes can exist in a number of environments, and ideally this framework is generalizable. Therefore, for any given development environment, a PKL system must first identify the procedural primitives, the smallest meaningful units of work, such as cells in a notebook, functions in a script, or tasks in a pipeline. Next, it is necessary to extract the execution context, including the order of operations, dependencies, and any relevant environment configuration required to interpret or reproduce the procedure. These procedural elements must be captured in a structured way, such as JSON objects, abstract syntax trees (ASTs), or a node in a dependency graph, making them standardized and queryable.

Crucially, one must define patch semantics for the environment: that is, specify how changes are represented and what constitutes a transformation, such as a modified code cell, reordered steps, or updated parameters. Finally, the system must implement lens mappings that associate user-defined transformations with the underlying procedural structures, enabling selective extraction, abstraction, or reinterpretation of the process according to task-specific goals.

Procedural Unit Granularity

What is the smallest traceable and meaningful unit of work in the environment?

Table 1: Procedural unit examples across environments

Development Environment	Procedural Unit
Jupyter Notebooks	Code cell
Python Scripts	Function or block
RStudio/RMarkdown	Code chunk
Ib IDEs (e.g., Codespaces)	File change or commit
Data Pipelines (e.g., Airflow, Luigi)	Task or DAG node
Domain-specific (Modelling) Language	Construct
Interactive Notebooks (e.g., Observable)	Cell or reactive block

5.3.1 Application to Jupyter Workflows

Encoding a Jupyter notebook into a PKL requires identifying and transforming its procedural elements into structured, queryable units. A notebook consists of code cells, outputs, markdown cells, metadata, and execution order—each of which contributes to the procedural context.

I define the encoding process as follows:

1. **Cell Extraction:** Parse the notebook's JSON to extract code cells, markdown cells, and their metadata.
2. **Execution Trace Capture:** Record execution order (e.g., `execution_count`) and timestamped outputs to reconstruct the temporal flow.
3. **Dependency Analysis:** Infer dependencies between cells using variable usage and definition analysis (e.g., via AST parsing or tools like nbdime).
4. **Lens Application:** Apply lenses such as:
   - **Extraction Lens:** Isolate all cells using a particular library (e.g., matplotlib) or performing a specific task (e.g., data loading).
   - **Abstraction Lens:** Collapse multiple exploratory steps into a generalized data-cleaning block.
   - **Temporal Lens:** Compare notebook state between two commits or execution runs.
5. **Patch Generation:** Represent each edit (cell added, removed, modified) as a diff in unified format, enriched with semantic tags (e.g., type: plotting, intent: exploratory).
6. **Semantic Indexing:** Store the transformed cells and their relationships in the metadata database, allowing retrieval by task, variable, or semantic intent.

Example Lens Template for Notebook Cell Extraction:

”TargetSchema”: ”cell-snippet-set”, ”PatchTemplate”: ”extractPattern”: ”.(read_csv | read_excel | read_parquet).”, ”cellType”: ”code”, ”matchField”: ”source” , ”InversePatchTemplate”: ”mergeStrategy”: ”append-to-top”, ”targetNotebook”: ”inferred”

5.4 Towards Meaningful Encodings

Encoding in a PKL refers to transforming raw procedural information—such as code cells, function calls, or pipeline steps—into structured, reusable representations of procedural knowledge. This section outlines what makes an encoding meaningful and how PKLs differentiate between intent and outcome.

5.5 Goals & Challenges

A central challenge in encoding procedural knowledge is distinguishing between intent and execution. In many workflows—especially exploratory ones, such as computational notebooks—goals often evolve and remain implicit (Rule et al., 2018).

PKLs address this by treating the end state—such as a result, figure, or model checkpoint—as the source of a linked procedural trace defined as a *procedural unit*.

Each procedural unit (e.g., a notebook cell) may include annotations indicating its purpose or expected outcome (e.g., `load data`, `train model`, `plot results`). These semantic tags, stored in the metadata database, allow us to reconstruct the chain of reasoning and compare what was attempted with what was achieved. For example, a failed training step and its correction can both be stored, annotated with `intent: model training` and `outcome: failed convergence`, enabling retrospective understanding.

Encoding the goal explicitly can involve metadata fields like `TargetMetric`, `ExpectedOutput`, or `DesiredState`, while the final artifacts—plots, metrics, exports—can be automatically captured and indexed. This setup supports querying not only by actions taken, but also by the original intention behind them.

5.6 Semantic & Temporal Structuring

To support meaningful encoding, our PKL system integrates semantic tagging and temporal reasoning through structured metadata. A classification engine parses each unit—such as a Jupyter cell, Python function, or DAG node—using (Python's) abstract syntax trees (ASTs), import resolution, and pattern matching. It labels each unit with semantic tags (e.g., `data loading`, `model training`), stored as JSON-LD (*JSON-LD - JSON for Linked Data — Json-Ld.org*).

5.6.1 Jupyter Semantic Classification Engine

Luckily, for Jupyter notebooks there is a common language across notebooks—Python. Furthermore, all libraries are imported from a shared library of "modules." Thus, one can programmatically extract the primitive functions and create a repository of functions to parse from a notebook. Each method can be considered an "operation" as a heuristic for a "step" or "unit" in a PKL.

The following is a code snippet intended to extract all `import` statements from a `.ipynb` file and the function definitions for each.

import nbformat import re

Load the notebook with open("notebook.ipynb") as f: nb = nbformat.read(f, as_version=4)

imports = set() functions = set()

Regex patterns for imports and function defs import_pattern = re.compile(r'^(import|from)+[^]+') def_pattern = re.compile(r'^def++$(')

for cell in nb.cells: if cell.cell_type == "code": for line in cell.source.splitlines(): if import_pattern.match(line): imports.add(line.strip()) if def_pattern.match(line): functions.add(line.strip())

print("Imports found:") for imp in sorted(imports): print(imp)

print("definitions found:") for func in sorted(functions): print(func)

Next, module-level abstract syntax trees (ASTs), provided for all Python modules, and accessible through the `ast` Python module can then be used to produce more granular, expressive annotations for a PKL unit (Arts, 2022). The process consists of parsing Python code in ordered, strcutured fashion producing one hierarchical AST for a whole notebook. ASTs for `.ipynb` are not inherently cell-order aware. Therefore, the metadata that makes up PKL provenance serves as enrichment to ASTs for procedure extraction.

6 Order & Time

In a notebook, there's different layers of temporality that is valuable to track. First, there's cell-execution order. In a notebook, the order of the cells when editing does not matter. However, someone running a notebook, must execute cells in the appropriate order for it to run correctly. Execution order is already captured in `.ipynb` file metadata with the `execution_count` field which records the order that cells were run in a kernel session. The execution order is represented as an ordered list.

In PKLs, cell order alone is too coarse-grained, as each cell may contain multiple distinct operations or steps. To address this, the PKL database uses a nested

structure, embedding finer-grained PKL units within each notebook cell. Procedural order is tracked using Lamport timestamps—logical clocks that capture causal relationships across operations (Lamport, 1978). Each unit is assigned a Lamport value, which is updated with edits, reordering, or forks.

The Lamport timestamp algorithm can be shown as follows:

Initialization time = 0

On local event or before sending a message time = time + 1 send(message, time)

On receiving a message (message, timestamp) = receive() time = max(time, timestamp) + 1

By combining semantic tags with logical timestamps, workflows gain both meaning and a clear sense of order. Logical clocks, such as Lamport timestamps, help track the sequence of events and their causal relationships. Semantic tags add context and make it easier to interpret and organize actions. Together, they allow systems to rebuild the history of a process, handle overlapping edits, and reuse past work in new settings. This makes workflows more reliable, easier to review, and better suited for collaboration.

6.1 Lenses as Procedural Abstractions

We propose a patch-based encoding approach in which lenses—bidirectional transformations—is the means for isolating and modifying procedural elements.

Lens Types:

1. **Extraction Lenses:** Isolate specific elements (e.g., hyperparameters)
2. **Abstraction Lenses:** Summarize or simplify detailed procedures
3. **Transformation Lenses:** Convert formats or representations
4. **Temporal Lenses:** Compare or filter by time/version

Each lens is implemented as a parameterized patch template, enabling programmable views that preserve reversibility.

”LensID”: ”hyperparameter-focus”, ”Description”: ”Isolates model hyperparameters from full procedure”, ”SourceSchema”: ”ml-training-procedure”, ”TargetSchema”: ”hyperparameter-set”, ”PatchTemplate”: ”contextLines”: 0, ”pathPattern”: ”**/model_config.json,yaml”, ”extractPattern”: ”`$\.training.hyperparameters.*`” ” , ”InversePatchTemplate”: ”mergeStrategy”: ”deep-merge”, ”targetPath”: ”`$\.training.hyperparameters.*`”

Lens Composition:

To support complex operations, lenses can be composed sequentially. Composition is valid when the output schema of one lens matches the input schema of the next.

def can_compose(lens1, lens2): return lens1.targetSchema.isCompatibleWith(lens2.sourceSchema)

def compose_lenses(lens1, lens2): if not can_compose(lens1, lens2): raise IncompatibleLensError() return Lens(sourceSchema=lens1.sourceSchema, targetSchema=lens2.targetSchema, transform=lambda x: lens2.transform(lens1.transform(x)), inverse=lambda y: lens1.inverse(lens2.inverse(y)))

This modular encoding system allows for scalable transformation and precise interpretation of procedural traces.

6.2 Storage

Once encoded, procedural knowledge must be persistently stored in a form that is accessible, queryable, and robust to evolution over time.

6.2.1 File Format and Storage Considerations

I leverage established version control concepts and formats:

- **Base Format:** JSON or YAML for structured data, with additional formats for domain-specific data
- **Patch Format:** Standard unified diff format (compatible with git, diff, patch utilities)
- **Directory Structure:**

  procedures/ procedure-id/ base/ files representing base procedure versions/ v1/ v2/ ... views/ lens-id-params-hash/ ... ...

6.2.2 Storage Proposal

I propose an architecture based on:

1. **Git-Compatible Storage:** Leveraging mature version control for procedure history
2. **Patch Files:** Standard unified diff format for representing transformations
3. **Metadata Database:** Lightweight index for lens information and relationships

Database Schema

Procedures ProcedureID (PK) BasePath Description Tags CreatedAt Lenses LensID (PK) Type (Extraction|Abstraction|Transformation|Temporal) SourceSchema TargetSchema PatchTemplate InversePatchTemplate Views ViewID (PK) ProcedureID (FK) LensID (FK) Parameters Path

CreatedAt Compositions CompositionID (PK) Name LensSequence [LensID] ValidationRules

This schema provides a lightweight index on top of a filesystem-based storage system.

6.2.3 Key Features

- **Filesystem-Based Storage:** Procedures and their versions are stored as files and directories, making them compatible with existing tools
- **Standard Patch Format:** Changes are represented using the unified diff format, making them human-readable and compatible with existing patch tools
- **Metadata Database:** A database that indexes procedures, lenses, and views for efficient querying
- **Composition Rules:** Explicit rules for lens composition ensure transformations maintain integrity

6.3 Retrieval

The retrieval layer enables users to locate, interpret, and reuse procedural knowledge.

6.3.1 Query Language

Find procedures by tag FIND PROCEDURES WHERE tags CONTAINS "image-classification"

View a procedure through a lens VIEW procedure-19 THROUGH lens:high_level_summary

Apply a sequence of lenses VIEW procedure-19 THROUGH lens:extract_hyperparams THEN lens:visualize_as_table

Compare versions DIFF procedure-7:v1 AGAINST procedure-7:v3

Extract specific steps FROM procedure-42 GET STEPS 3 TO 5

6.3.2 Implementing Retrieval

The retrieval system is implemented as:

1. **Command-Line Interface:** Git-like CLI for managing procedures
2. **API Layer:** RESTful and GraphQL APIs for programmatic access
3. **Ib Interface:** Visual exploration of procedures, versions, and transformations

Example CLI commands:

Create a new procedure from existing files pkl create-procedure—name ”image-classification”–base-path ./training_code/

Apply a lens to create a view pkl apply-lens hyperparameter-focus–to procedure-42

Compare two versions pkl diff procedure-7:v1 procedure-7:v3

Export a view to a specific format pkl export procedure-19 –lens high_level_summary–format markdown

6.4 Practical Implementation

6.4.1 File Formats

1. **Procedure Base:** JSON, YAML, Markdown, code files in native formats
2. **Patches:** Standard unified diff format
3. **Lens Definitions:** JSON schema
4. **Metadata:** SQLite or a fitting alternative

6.4.2 Operation Flow

-1.5cm-1.5cm

PKL Operation Flow Diagram

6.4.3 Example Implementation

Creating a procedure def create_procedure(name, base_path, tags=None):
Generate a unique ID procedure_id = generate_id()

# Create directory structure os.makedirs(f”procedures/procedure_id/base”)
os.makedirs(f”procedures/procedure_id/versions”) os.makedirs(f”procedures/procedure_id/views”)

Copy base files for file in glob.glob(f”base_path/**/*”, recursive=True):
if os.path.isfile(file): rel_path = os.path.relpath(file, base_path) target_path =
f”procedures/procedure_id/base/rel_path” os.makedirs(os.path.dirname(target_path),
exist_ok=True) shutil.copy2(file, target_path)

Add to index db.execute(”INSERT INTO Procedures (ProcedureID, Name,
BasePath, Tags, CreatedAt) VALUES (?, ?, ?, ?, ?)”, (procedure_id, name,
f”procedures/procedure_id/base”, json.dumps(tags or []), datetime.now()))

return procedure_id

Applying a lens def apply_lens(procedure_id, lens_id, parameters=None):
Get procedure and lens info procedure = db.query(”SELECT * FROM
Procedures WHERE ProcedureID = ?”, (procedure_id,)).fetchone()

lens = db.query(”SELECT * FROM Lenses WHERE LensID = ?”, (lens_id,)).fetchone()

Generate a view ID param_hash = hashlib.md5(json.dumps(parameters or).encode()).hexdigest() view_id = f”procedure_id-lens_id-param_hash” view_path = f”procedures/procedure_id/views/lens_id-param_hash”

Create view directory os.makedirs(view_path, exist_ok=True)

Apply lens template to generate patch patch = generate_patch_from_template(procedure[”BasePath”], json.loads(lens[”PatchTemplate”]), parameters)

Apply patch to create view apply_patch(patch, procedure[”BasePath”], view_path)

Add to index db.execute(”INSERT INTO Views (ViewID, ProcedureID, LensID, Parameters, Path, CreatedAt) VALUES (?, ?, ?, ?, ?, ?)”, (view_id, procedure_id, lens_id, json.dumps(parameters or), view_path, datetime.now()))

return view_id

This implementation provides a concrete and practical approach to building a Procedural Knowledge Library system that preserves the essential capabilities for encoding, storing, and retrieving procedural knowledge.

6.5 Search & Navigation

Search is one of the most important features of a Procedural Knowledge Library. It allows users to find and reuse past work without having to manually dig through files or notebooks. Whether someone is looking for a specific step in a project, a general method for solving a problem, or a comparison between different approaches, the search layer makes it possible to retrieve that information quickly and clearly.

Under the hood, search is powered by a structured metadata system. Each piece of a procedure—such as a code cell, function, or data-processing step—is stored as a unit with attached information: what it does, where it came from, when it was created, and how it fits into the larger process. This information is stored in a lightweight SQL database, which support fast indexing and querying.

The core database tables are presented as follows:

– Table of procedures CREATE TABLE Procedures (ProcedureID TEXT PRIMARY KEY, Name TEXT, CreatedAt TIMESTAMP);

– Units of procedural work (e.g., cells, functions) CREATE TABLE Units (UnitID TEXT PRIMARY KEY, ProcedureID TEXT, Content TEXT, StartLine INT, EndLine INT, SemanticTag TEXT, – e.g., ”data loading”, ”plotting” LamportClock INT, FOREIGN KEY (ProcedureID) REFERENCES Procedures(ProcedureID));

– Views created through lens application CREATE TABLE Views (ViewID TEXT PRIMARY KEY, ProcedureID TEXT, LensID TEXT, Parameters TEXT, Path TEXT, CreatedAt TIMESTAMP);

Using this database, a range of queries become possible. Here are a few examples:

- Search for procedures by task: `FIND procedures WHERE tag="data cleaning"`
- Retrieve steps with a specific role: `GET all units WHERE role="visualization"`
- View a procedure through a different lens: `VIEW procedure-12 THROUGH lens:abstraction`
- Compare versions over time: `DIFF procedure-19:v1 AGAINST v3`

6.5.1 Search Capabilities

Search in PKLs is designed not just for finding information, but for helping people understand, reuse, and build on prior work more effectively.

Each query can return a filtered version of the procedure, a summary of what each part is doing, or a list of changes between two versions. This helps users understand not just what was done, but how it was done and why certain steps were taken.

To make this easy to use, the system would support both a command-line interface (CLI) and a visual interface. The visual interface allows users to explore procedures as trees or timelines and apply lenses interactively. There's also support for fuzzy search, so users can type natural language queries like "load CSV and plot scripts" and get relevant results.

6.5.2 Debugging Example

Consider a researcher maintaining a model that suddenly performs worse on a key benchmark after recent changes. With a Procedural Knowledge Library, the researcher can query the last known "good" version and compare it to the current version using a temporal lens:

DIFF procedure-17:v4 AGAINST procedure-17:v7

This returns not just a code-level diff, but a semantic view showing that a hyperparameter tuning step was removed, and a data augmentation function was modified. Because each change is tagged with its role (e.g., training config, data preprocessing), the system surfaces meaningful differences rather than low-level line edits.

The researcher can then replay the procedural path leading to version 4, verify intermediate results, and selectively roll back or adjust specific units. This

enables faster root-cause analysis and avoids re-running full experiments from scratch—saving both time and compute.

7 Existing Systems & Their Limitations

7.1 Experimental Reproducibility

Reproducing machine learning results often means guessing how experiments were actually run. Initiatives like the NeurIPS Reproducibility Challenge show the desire for transparency and explainability of the research process. The difficulties faced by the 6-year initiative also show whynumerical outputs and shared code are rarely enough (Pineau et al., 2020). Tools such as MLflow and the Sacred track parameters, but do not fully capture the steps the researchers took or why (Chen et al., 2020; Greff et al., 2017). Procedural Knowledge Libraries (PKLs) aim to fill this gap by recording not just outputs but also the decisions that led to them.

7.2 Case Study: Procedural Archaeology in the o1 Model Family

The OpenAI o1 model family illustrates the need for better process-level documentation. While OpenAI shared results suggesting that inference-time compute—rather than training scale—drove performance gains, no code or method was released to replicate their scaling law graphs (OpenAI et al., 2024).

In response, external researchers reverse-engineered the evaluation pipeline using the o1-mini API (Zhang & Chen, 2024). They simulated computation time via prompt design, inferred token counts from billing, and approximated majority vote logic. Yet their reproduction diverged from OpenAI's results at high compute levels, raising questions about hidden evaluation steps, architectural changes, or prompt tuning.

This reconstruction effort—essentially procedural archaeology—highlights why PKLs matter. Even with visible outputs, the absence of procedural detail leads to ambiguity and loss of scientific rigor. PKLs are positioned to fill this gap by recording not just outputs but also the decisions that led to them.

8 Evaluating PKLs

The evaluation of a Procedural Knowledge Library system must extend beyond theoretical assessments to include practical implementation concerns. I propose a multidimensional evaluation framework that addresses both the system's theoretical foundations and its utility in real-world research environments.

Another important dimension for evaluating PKL systems is the overhead they introduce to existing workflows. PKLs add computational cost through con-

tinuous tracking, versioning, and indexing of procedural traces. Without optimization, storage requirements scale with $O(n \log v)$, where n is the size of procedural artifacts and v is the number of tracked versions. This is more efficient than naive approaches that store full copies of each version ($O(nv)$), but increases retrieval overhead, as procedures must be reconstructed from base versions and patch sequences. More broadly, the abstraction layers introduced by PKLs contribute to complexity in three areas: encoding overhead during capture, storage overhead for managing metadata, and retrieval overhead when rendering or transforming procedures.

9 File Formats & Interoperability

The choice of file formats dramatically impacts a PKL's interoperability with existing tools and workflows. Ideally, a PKL should leverage established formats where possible, introducing custom extensions only when necessary for representing procedural metadata.

Our implementation utilizes a layered approach to file formats:

- Base Layer: Native formats for domain-specific artifacts (e.g., Python files, notebooks, configuration files)
- Differential Layer: Standard unified diff format for tracking changes
- Lens Layer: JSON Schema-based lens definitions for transformations
- Index Layer: Lightweight database schema for efficient queries

This approach minimizes the "format tax" imposed on users while enabling the essential features of procedural knowledge management. Integration with existing version control systems like Git allows PKLs to leverage mature infrastructure while extending it with procedural semantics. By implementing PKL operations as extension commands to Git, researchers can continue using familiar workflows while gaining access to procedural knowledge capabilities.

10 Integration into Existing Systems

For PKLs to achieve adoption, they must integrate seamlessly with existing research environments. I outline three paths to do so:

Computational notebooks represent a natural starting point, as they already combine code, documentation, and results in a semi-structured format. Our prototype integrates with Jupyter through a custom extension that automatically captures execution traces and enables lens-based views. This approach requires minimal changes to existing workflows.

Integration with development environments presents additional challenges due to their focus on textual manipulation rather than execution traces. Our imple-

mentation addresses this through a combination of language server extensions and procedure inferencing techniques that reconstruct procedural knowledge from code repositories and execution logs.

Scientific workflows systems (e.g., Apache Airflow, Luigi) offer more structured representations of procedures but typically lack the semantic richness needed for knowledge transfer. By extending these systems with lens-based views and semantic annotations, PKLs can preserve both execution details and the reasoning behind workflow design decisions.

11 Discussion

11.1 Considerations for Adoption

The complexity of any new system introduces a learning curve that can impede adoption. For PKLs, this presents as three distinct challenges: (1) Conceptual overhead: Understanding the lens-based manipulation model (2) Technical overhead: Learning new commands and interfaces (3) Process overhead: Adapting existing workflows to support procedural knowledge capture

The learning investment is non-trivial but justified by longitudinal benefits, particularly for collaborative teams or research programs spanning multiple years.

Adopting a Domain-Specific Language (DSL) for procedural knowledge presents additional trade-offs. While a DSL enables precise expression of procedural semantics, it introduces a steeper learning curve. Our evaluation suggests that DSL complexity should be minimized for initial adoption, with more sophisticated features introduced progressively as users become familiar with the system.

11.1.1 Long-term Viability and Preservation

The value of procedural knowledge increases over time, making preservation a critical concern. PKLs must be evaluated not just for their immediate utility but for their longevity in the face of changing technologies and research practices.

Our approach to preservation includes: format migration strategies that can adapt as file formats evolve, emulation capabilities that preserve execution environments, semantic annotations that capture intent independent of specific implementations, and dependency management that tracks external resources and their versions.

Thus, PKLs help conserve procedural knowledge so it stays useful—even as tools, formats, and technologies change over time.

12 Future Work

A natural direction for future work is to generalize the foundation of Procedural Knowledge Libraries to support a wider range of development environments. Developers frequently move between tools and interfaces—from notebooks and scripts to CI pipelines, configuration files, and visual editors. This raises an open problem: how can PKL artifacts be composed across diverse systems while still enabling procedural capture?

To address this, future PKL systems must support contributions from different tools and environments, and encode transformations that span across modalities and domains. This requires representations that can unify procedures originating from distinct formats and workflows while maintaining structure and interpretability. One promising path lies in research on lenses and bidirectional transformations. Foster et al. explore a relevant framework in their work on "Combinators for Bi-Directional Tree Transformations," which addresses the View Update problem in structured data contexts (Foster et al., 2007). Extending such approaches to PKLs may provide a foundation for composing, diffing, and synchronizing procedural artifacts across tools in a way that remains both interpretable and reversible.

Another avenue for future work involves systematically extracting procedural knowledge—or implicit documentation—from existing outputs such as notebooks, code repositories, logs, and reports. These artifacts often contain rich procedural information that, if harnessed effectively, can enhance transparency and reproducibility. Techniques from information extraction (IE) and natural language processing (NLP) have shown promise in structuring such content. For instance, Agarwal et al. propose methods to extract procedural knowledge from technical documents using structural and linguistic patterns (Agarwal et al., 2020).

Schemas for extracted procedural data can help ensure consistency in PKL artifacts and help automate the process (DistillerSR, 2024). Integrating these approaches could automate PKL artifact generation from legacy materials. Aligning extraction with standardized metadata schemas (Akhtar et al., 2024).

Looking ahead, several directions could extend the utility of Procedural Knowledge Libraries (PKLs). One avenue is their integration with machine learning agents (*AlphaEvolve*), where PKLs could serve as long-term procedural memory—supporting grounded, revisable, multi-step behavior and enabling agents to reflect, adapt, or reuse prior workflows.

Another line of work involves enhancing PKLs with temporal semantics. Embedding structured representations of time—such as causal ordering or change intervals—could support time-aware queries, comparisons over workflow evolution, and replay of development histories. For example, this may include capturing when specific steps were added, modified, or deprecated (e.g., timestamped edits, version intervals, or dependency-aware sequences), enabling users to trace

how workflows evolve, identify when key decisions were made, and reason about changes over time.

13 Next Steps

In the context of Jupyter notebooks, PKLs offer a path toward semantically-aware versioning. Traditional line-based diffs are ill-suited to notebooks, which blend code, outputs, markdown, and execution metadata. PKLs can represent these elements as traceable units, supporting structured diffs, procedural patching, and more verifiable forms of citation. Future work includes developing extensions that reconstruct execution graphs from non-linear cell runs (Prenner & Robbes, 2025), improving provenance tracking and reproducibility.

Additional work is needed to explore access control and privacy-preserving mechanisms for procedural traces—particularly when they contain sensitive information such as API keys, credentials, or unpublished results. Techniques from differential privacy and secure provenance tracking could be adapted to this setting.

Finally, future efforts should assess the usability of different procedural representations. Comparative studies could examine how formats like notebooks, scripts, or structured workflows affect the ability of users—or models—to understand, modify, or transfer procedures. This research would inform design decisions for PKLs, ensuring they remain not only expressive but learnable and reusable across contexts.

14 Conclusion

By capturing both the steps and context of research processes, PKLs allow for more efficient, collaborative, and reproducible science. The architecture I have proposed—based on lens transformations, patch-based storage, and semantic retrieval—provides a practical blueprint for implementing PKLs across diverse research domains. As computational methods continue to dominate scientific discovery, the ability to effectively manage procedural knowledge will become increasingly central to research progress, turning tacit knowledge into explicit, shareable resources that improve how science is done and disseminated.

14.1 References

Aamodt, A., & Nygård, M. (1995). Different roles and mutual dependencies of data, information, and knowledge — an AI perspective on their integration. *Data Knowl. Eng.*, *16*(3), 191–222.

Abe, R., Ito, A., Takayasu, K., & Kurihara, S. (2025). *LLM-mediated dynamic plan generation with a multi-agent approach*. https://arxiv.org/abs/2504.01637

Acharya, K., Raza, W., Dourado, C., Velasquez, A., & Song, H. H. (2024). Neurosymbolic reinforcement learning and planning: A survey. *IEEE Transactions on Artificial Intelligence*, *5*(5), 1939–1953. https://doi.org/10.1109/tai.2023.3311428

Agarwal, S., Atreja, S., & Agarwal, V. (2020). Extracting procedural knowledge from technical documents. *arXiv Preprint arXiv:2010.10156*.

Akhtar, M., Benjelloun, O., Conforti, C., et al. (2024). Croissant: A metadata format for ML-ready datasets. *arXiv Preprint arXiv:2403.19546*.

AlphaEvolve: A Gemini-powered coding agent for designing advanced algorithms — deepmind.google. https://deepmind.google/discover/blog/alphaevolve-a-gemini-powered-coding-agent-for-designing-advanced-algorithms/.

Ameisen, E., Lindsey, J., Pearce, A., Gurnee, W., Turner, N. L., Chen, B., Citro, C., Abrahams, D., Carter, S., Hosmer, B., Marcus, J., Sklar, M., Templeton, A., Bricken, T., McDougall, C., Cunningham, H., Henighan, T., Jermyn, A., Jones, A., … Batson, J. (2025). Circuit tracing: Revealing computational graphs in language models. *Transformer Circuits Thread*. https://transformer-circuits.pub/2025/attribution-graphs/methods.html

Arts, E. (2022). *Towards querying abstract syntax trees for python programs* [PhD thesis]. Master Thesis, Department of Computer Science, Eindhoven University of ….

Blundell, C., Uria, B., Pritzel, A., Li, Y., Ruderman, A., Leibo, J. Z., Rae, J., Wierstra, D., & Hassabis, D. (2016). *Model-free episodic control*. https://arxiv.org/abs/1606.04460

Byrnes, J. P., & Wasik, B. A. (1991a). Role of conceptual knowledge in mathematical procedural learning. *Dev. Psychol.*, *27*(5), 777–786.

Byrnes, J. P., & Wasik, B. A. (1991b). Role of conceptual knowledge in mathematical procedural learning. *Developmental Psychology*, *27*(5), 777.

Carriero, V. A., Scrocca, M., Baroni, I., Azzini, A., & Celino, I. (2025). *Procedural knowledge ontology (PKO)*. https://arxiv.org/abs/2503.20634

Chang, M. B., Gupta, A., Levine, S., & Griffiths, T. L. (2019). *Automatically composing representation transformations as a means for generalization*. https://arxiv.org/abs/1807.04640

Chen, A., Chow, A., Davidson, A., DCunha, A., Ghodsi, A., Hong, S. A., Konwinski, A., Mewald, C., Murching, S., Nykodym, T., Ogilvie, P., Parkhe, M., Singh, A., Xie, F., Zaharia, M., Zang, R., Zheng, J., & Zumar, C. (2020). Developments in MLflow: A system to accelerate the machine learning lifecycle. *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning*. https://doi.org/10.1145/3399579.3399867

Ciatto, G., Agiollo, A., Magnini, M., & Omicini, A. (2025). Large language models as oracles for instantiating ontologies with domain-specific knowledge. *Knowledge-Based Systems*, *310*, 112940. https://doi.org/10.1016/j.knosys.2024.112940

Cohen, E., Kaplan, H., Mansour, Y., Moran, S., Nissim, K., Stemmer, U., & Tsfadia, E. (2024). *Data reconstruction: When you see it and when you don't*. https://arxiv.org/abs/2405.15753

DistillerSR. (2024). *Data extraction template for systematic review*.

https://www.distillersr.com/resources/systematic-literature-reviews/data-extraction-template-for-systematic-review.

Elsaka, E. (2017). Fault localization using hybrid static/dynamic analysis. In *Advances in computers* (pp. 79–114). Elsevier.

Felice, G. de. (2022). *Categorical tools for natural language processing*. https://arxiv.org/abs/2212.06636

Fong, B., & Johnson, M. (2019). Lenses and learners. *CoRR*, *abs/1903.03671*. http://arxiv.org/abs/1903.03671

Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., & Schmitt, A. (2007). Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, *29*(3), 17–es. https://doi.org/10.1145/1232420.1232424

Goel, K., & Brunskill, E. (2019). Learning procedural abstractions and evaluating discrete latent temporal structure. *International Conference on Learning Representations*. https://openreview.net/forum?id=ByleB2CcKm

Gottlob, G., Paolini, P., & Zicari, R. (1986). *Properties and update semantics of consistent views* (UCB/CSD-86-258). http://www2.eecs.berkeley.edu/Pubs/TechRpts/1986/6078.html

Greff, K., Klein, A., Chovanec, M., Hutter, F., & Schmidhuber, J. (2017). The sacred infrastructure for computational research. *Proceedings of the 16th Python in Science Conference*, 49–56.

Gregory, P., & Lindsay, A. (2016). Domain model acquisition in domains with action costs. *Proc. Int. Conf. Autom. Plan. Sched.*, *26*, 149–157.

Joshy, A. K., & Le, W. (2020). *Invariant diffs*. https://arxiv.org/abs/1911.07988

JSON-LD - JSON for Linked Data — json-ld.org. https://json-ld.org/.

Kleppmann, M. (2021). Thinking in events: From databases to distributed collaboration software. *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems*, 15–24.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, *21*(7), 558–565. https://doi.org/10.1145/359545.359563

McCarthy, W. P., Hawkins, R. D., Wang, H., Holdaway, C., & Fan, J. E. (2021). *Learning to communicate about shared procedural abstractions*. https://arxiv.org/abs/2107.00077

McCormick, R. (1997). Conceptual and procedural knowledge. *Int. J. Technol. Des. Educ.*, *7*(1-2), 141–159.

Oderinwale, H. (2025). Towards codified context, durable documentation, and process preservation. In *Topos Institute*. https://topos.institute/blog/2025-02-27-codified-context/

OpenAI, :, Jaech, A., Kalai, A., Lerer, A., Richardson, A., El-Kishky, A., Low, A., Helyar, A., Madry, A., Beutel, A., Carney, A., Iftimie, A., Karpenko, A., Passos, A. T., Neitz, A., Prokofiev, A., Wei, A., Tam, A., … Li, Z. (2024). *OpenAI o1 system card*. https://arxiv.org/abs/2412.16720

Parkinson, S., Longstaff, A., Crampton, A., & Gregory, P. (2012). The application of automated planning to machine tool calibration. *Proc. Int. Conf.*

Autom. Plan. Sched., *22*, 216–224.

Pineau, J., Vincent-Lamarre, P., Sinha, K., Larivière, V., Beygelzimer, A., d'Alché-Buc, F., Fox, E., & Larochelle, H. (2020). *Improving reproducibility in machine learning research (a report from the NeurIPS 2019 reproducibility program)*. https://arxiv.org/abs/2003.12206

Poveda-Villalón, M., Fernández-Izquierdo, A., Fernández-López, M., & García-Castro, R. (2022). LOT: An industrial oriented ontology engineering framework. *Eng. Appl. Artif. Intell.*, *111*(104755), 104755.

Prenner, J. A., & Robbes, R. (2025). *Simple fault localization using execution traces*. https://arxiv.org/abs/2503.04301

Rule, A., Tabard, A., & Hollan, J. D. (2018). Exploration and explanation in computational notebooks. *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 1–12. https://doi.org/10.1145/3173574.3173606

Shah, H., Park, S. M., Ilyas, A., & Madry, A. (2022). *ModelDiff: A framework for comparing learning algorithms*. https://arxiv.org/abs/2211.12491

Zeng, X., Diekmann, N., Wiskott, L., & Cheng, S. (2023). Modeling the function of episodic memory in spatial learning. *Frontiers in Psychology*, *14*, 1160648.

Zhang, H., & Chen, C. (2024). *Test-time compute scaling laws*. https://github.com/hughbzhang/o1_inference_scaling_laws.