`mandala`: Compositional Memoization for Simple & Powerful Scientific Data Management

Abstract

We present `mandala`[1], a Python library that largely eliminates the accidental complexity of scientific data management and incremental computing. While most traditional and/or popular data management solutions are based on *logging*, `mandala` takes a fundamentally different approach, using *memoization* of function calls as the fundamental unit of saving, loading, querying and deleting computational artifacts.

It does so by implementing a *compositional* form of memoization, which keeps track of how memoized functions compose with one another. In this way: (1) complex computations are effectively memoized end-to-end, and become 'interfaces' to their own intermediate results by retracing the memoized calls; (2) all computations in a project form a single computational graph, which can be explored, queried and manipulated in high-level ways through a *computation frame*, which is a natural generalization of a dataframe that replaces columns by a computation graph, and rows by (partial) executions of this graph.

Several features implemented on top of the core memoization data structures — such as natively and transparently handling Python collections, in-memory caching of intermediate results, and a flexible versioning system with dynamic dependency tracking — turn `mandala` into a practical and simple tool for managing and interacting with computational data.

1 Introduction

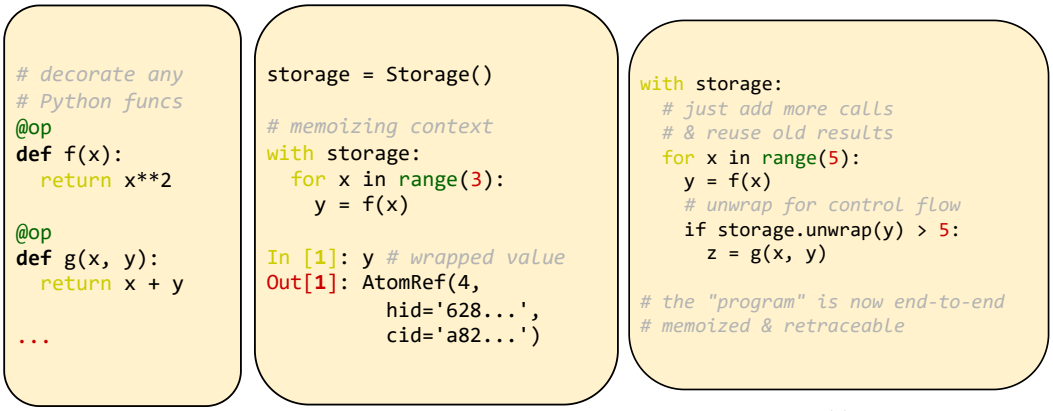
Numerical experiments and simulations are growing into a central part of many areas of science and engineering (Hey et al., 2009). Recent trends in computation-intensive fields, such as machine learning, point towards (1) ever-increasing complexity of computational pipelines, and (2) adoption in more safety-critical domains, such as autonomous driving (Bojarski et al., 2016) and healthcare (Ravì et al., 2016; Abramson et al., 2024).

These developments impose opposing constraints on the tools used to manage the resulting computational artifacts. On the one hand, they should be simple and easy to use by researchers, with a minimal learning curve and unobtrusive syntax and semantics. On the other hand, they should deliver a lot of added functionality, such as high-level operations (Wickham, 2014), full data & code provenance auditing (Davidson & Freire, 2008) and reproducibility (Ivie & Thain, 2018) in complex projects. Rules and best practices that help with these requirements exist and are well-known (Sandve et al., 2013; Wilkinson et al., 2016), but still require manual effort, attention to extraneous details, and discipline to follow. Researchers often operate under time pressure and/or the need to quickly iterate on code, which makes these best 'practices' a rather *impractical* time investment.

Thus, ideally we would like a system that (1) does not get in the way by imposing a complex new language/semantics/syntax, (2) provides powerful high-level data management operations over complex computational projects, and (3) incorporates best practices by design and without cognitive overhead.

In this paper we present `mandala`, our proposal for such a system. It **integrates data management logic and best practices** such as

[1] `https://github.com/amakelov/mandala`




```python
# decorate any
# Python funcs
@op
def f(x):
  return x**2

@op
def g(x, y):
  return x + y

...
```

```python
storage = Storage()

# memoizing context
with storage:
  for x in range(3):
    y = f(x)

In [1]: y # wrapped value
Out[1]: AtomRef(4,
          hid='628...',
          cid='a82...')
```

```python
with storage:
  # just add more calls
  # & reuse old results
  for x in range(5):
    y = f(x)
    # unwrap for control flow
    if storage.unwrap(y) > 5:
      z = g(x, y)

# the "program" is now end-to-end
# memoized & retraceable
```

Figure 1: Basic imperative usage of `mandala`. **(a)**: add the `@op` decorator to any Python functions to make them memoizable. **(b)**: create a `Storage`, and use it as a context manager to automatically memoize any calls to `@op`-decorated functions in the block. Memoized functions return `Ref` objects, which wrap a value with two pieces of metadata: a *content ID*, which is a hash of the value of the object, and a *history ID*, which is a hash of the identity of the `@op` that produced the `Ref` (if any), and the history IDs of the `@op`'s inputs. **(c)**: the storage context allows simple incremental computation and recovery from failures. Here, we add more computations while automatically reusing already computed values.

- Full data provenance tracking
- Idempotent & reproducible computation
- Content-addressable versioning of code and its dependencies
- Declarative high-level manipulation of persisted computational graphs

into Python's already familiar syntax and semantics (Figures 1 and 2). The integration aims to be maximally transparent and unobtrusive, so that the user can focus on the *essential complexity* (the scientific problem at hand), rather than on the *accidental complexity* (the data management tools necessary to implement the solution) (Brooks, 1987).

The rest of this paper presents the design and main functionalities of `mandala`, and is organized as follows:

- In Section 2, we describe how memoization is designed, how this allows memoized calls to be composed and memoized results to be reused without storage duplication, and how this enables the *retracing* pattern of interacting with computational artifacts.
- In Section 3, we introduce the concept of a *computation frame*, which generalizes a dataframe by replacing columns with a computational graph, and rows with individual computations that (partially) follow this graph. Computation frames allow high-level exploration and manipulation of the stored computation graph, such as adding the calls that produced/used given values to the graph, deleting all computations that depend on the calls captured in the frame, and restricting the frame to a particular subgraph or subset of values with given properties.
- In Section 4, we describe some other features of `mandala` necessary to make it a practical tool, such as:
  - Representing Python collections in a way transparent to the storage, so that the membership relationships between a collection and its items are propagated through the saved computational graph;
  - Caching of intermediate results to speed up retracing and memoization;
  - A flexible versioning system with automatic dynamic dependency tracking.

Finally, we give an overview of related work in Section 5.

2 Core Concepts

2.1 Memoization and the Computational Graph

Memoization is a technique that stores the results of expensive function calls to avoid redundant computation. `mandala` uses *automatic* memoization (Norvig, 1991) which is applied via the combination of a decorator (`@op`) and a context manager which specifies the `Storage` object to use (Figure 1). The memoization can optionally be made persistent to disk, which is what you would typically want in a long-running project. Any Python function can be memoized (as long as its inputs and outputs are serializable by the `joblib` library; see the limitations section 6 for caveats); there is no restriction on the type, number or naming scheme (positional, keyword, variadic, or variable keyword) of the arguments or return values.

`Call` and `Ref` objects, and content/history IDs. `Ref`s and `Call`s are the two atomic data structures in `mandala`'s model of computations. When a call to an `@op`-decorated function `f` is executed inside a storage context, this results in the creation of

- A `Ref` object for each input to the call. These wrap the 'raw' values passed as inputs together with content IDs (hashes of the Python objects) and history IDs (hashes of the memoized calls that produced these values, if any).
  - If an input to the call is already a `Ref` object, it is passed through as is;
  - If it is a 'raw' (i.e., non-`Ref`) value, a new `Ref` object is created with a 'empty' history ID that is simply a hash of the content ID itself.
- A `Call` object, which has pointers to the input and output (defined below) `Ref`s of the call to `f`, as well as a content ID for the call (a hash of the identity of `f` and the content IDs of the input `Ref`s) and a history ID (by analogy, a hash of the identity of `f` and the history IDs of the inputs). The version of `f` is also part of the identity of `f`; see the versioning section 4.2 for details.
- A `Ref` object for each return value of the call. These are again assigned content IDs by value, and history IDs by hashing the tuple (history ID of the call, corresponding output name[2]).

The `Ref`s and the `Call` are then stored in the storage backend, and the next time `f` is called on inputs that have the same *content* IDs, the stored `Call` is looked up to find the output `Ref`s, which are then returned (possibly with properly updated history IDs, if the call exists in storage by content ID only). The combination of all stored `Call`s and `Ref`s across memoized functions form the **computational graph** represented by the storage. Importantly, the 'interesting' structure of this graph is built up automatically by the way the user composes memoized calls.

2.2 Motivation for the Design of Memoization

Why content and history IDs? The simultaneous use of content and history IDs has a few subtle advantages. First, it allows for the *de-duplication* of storage, as the same content ID can be used to store the same value produced by different computations. For instance, there may be many computaitions all producing the value 42 (or a large all-zero array), but only one copy of 42 is stored in the backend. At the same time, the history IDs allow us to distinguish between computations that produced the same value, but in different ways. This avoids 'parasitic' results in declarative queries. For example, a call to `f` may result in 42, and we may be interested in all calls that were ran on this particular 42 returned by `f` and not on any other `Ref` whose value happens to be 42. Without history IDs, it would be impossible to make this distinction in the stored computational graph.

[2]Since Python functions don't have designated output names, we instead generate output names automatically using the order in the tuple of return values.

Why memoization? Memoization is an unusual choice for data management systems, most of which are based on *logging*, i.e. explicitly pointing to the value to be saved and the address where it should be saved (whether this is some kind of name or a file path). Basing data management on memoization means that the 'address' of a value is now implicit in the code that produced it, and the value itself is stored in a shared storage backend. This has several advantages:

- It **eliminates the need to manually name artifacts**. This eliminates a major source of accidental complexity: names are arbitrary, ambiguous, and can drift away from the actual content of the value they point to over time. On the other hand, names are not strictly necessary, because the composition of memoized functions that produced a given value — which must be specified anyway for the computation to take place — is already an unambiguous pointer to it.
- Since the `@op` decorator encourages (and in a sense enforces) composition of `@op`s, it **automatically builds up a computational graph of the project**. Most data management tasks — e.g., a frequent use case is getting a table of relationships between some variables — are naturally expressed as queries over this graph, as we will see in Section 3.
- It **organizes storage functionality around a familiar and flexible interface: the function call**. This automatically enforces the good practice of partitioning code into functions, and eliminates extra 'accidental' code to save and load values explicitly. Furthermore, it synchronizes failures between computation and storage, as the memoized calls are the natural points to recover from.

On the other hand, memoization suffers from the following limitations:

- Referring to values without reference to the code that produced or used them becomes difficult, because from the point of view of storage the 'identity' of a value is its place in the computational graph. We discuss practical ways to overcome this in Section 3.
- Modifying `@op` functions requires care, as changes may invalidate the stored computational graph. We discuss a versioning system that automates this process in Section 4.2.

2.3 Retracing as a Versatile Imperative Interface to the Stored Computation Graph

The compositional nature of memoization makes it possible to build complex computations out of calls to memoized functions, turning the entire computation into an end-to-end-memoized interface to its own intermediate results. The main way to interact with such a persisted computation is through **retracing**, which means stepping through memoized code with the purpose of resuming from a failure, loading intermediate values, or continuing from a particular point with new computations. A small example of retracing is shown in Figure 1 (c).

This pattern is simple yet powerful, as it allows the user to interact with the stored computation graph in a way that is adapted to their use case, and to explore the graph in a way that is natural and familiar to them. It also simplifies the management of state in an interactive environment such as a Jupyter notebook, because it makes it very cheap to re-run cells in order to recreate the intended state of local variables.

3 Computation Frames

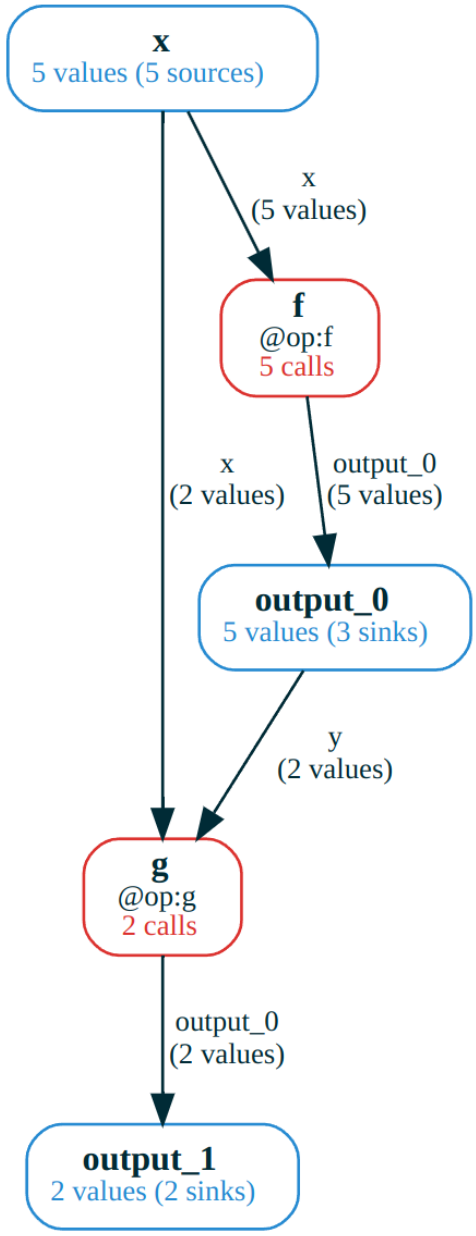
In order to be able to explore and manipulate the full stored computation graph, patterns like retracing are insufficient, because they require the complete code producing part of the graph to be available. To complement retracing, we introduce the `ComputationFrame` data structure, which is a high-level declarative interface to the stored computation graph.

```
In [1]:
  # get the computation frame for f
  storage.cf(f).\
    # add all computations reachable
    # from calls to f
    expand().\
      # extract as a dataframe
      eval()
Out [1]: Extracting tuples from the
computation graph:
  output_0 = f(x=x)
  output_1 = g(y=output_0, x=x)
```

(a) Continuing from Figure 1, we first create a computation frame from a single function `f`, then expand it to include all calls that can be reached from the memoized calls to `f` via their inputs/outputs, and finally convert the computation frame into a dataframe. We see that this automatically produces a computation graph corresponding to the computations found.

	x	f	output_0	g	output_1
0	4	Call(f, cid='05e...', hid='f87...')	16	Call(g, cid='4c0...', hid='c95...')	20.0
1	1	Call(f, cid='845...', hid='a55...')	1	None	NaN
2	3	Call(f, cid='f41...', hid='dab...')	9	Call(g, cid='00e...', hid='0ba...')	12.0
3	2	Call(f, cid='77b...', hid='1cb...')	4	None	NaN
4	0	Call(f, cid='82f...', hid='b98...')	0	None	NaN

(b) The output of the call to `.eval()` from the left subfigure used to turn the computaiton frame into a dataframe. The resulting table has columns for all variables and functions appearing in the captured computation graph, and each row correspond to a partial computation following this graph. The variable columns contain values these variables take, whereas function columns contain call objects representing the memoized calls to the respective functions. We see that, because we call g conditional on the output of `f`, some rows have nulls in the g column.



(c) A visualization of the computation frame from the previous two subfigures. The red nodes indicate functions, and the blue nodes indicate variables in the computation graph. Each edge is labeled with the input/output name of the adjacent function. Nodes and edges also show the number of `Refs` and `Calls` they represent.

Figure 2: Basic declarative usage of `mandala` and an example of computation frames.

3.1 Motivation and Intuition

Intuitively, a computation frame is a way to organize a collection of saved `@op` calls into groups, where the calls in each group have an analogous role in the computation, and the groups form a high-level computational graph of variables (which represent groups of `Refs`) and functions (groups of `Calls`). The illustration in Figure 2 (c) shows a visualization of a computation frame extracted from the computations in Figure 1.

This kind of organization is useful because it reflects how the user thinks about the computation, and allows them to tailor the exploration of the computation graph to a particular use case, much like a database view. For instance, sometimes it makes sense to group the outputs of several different `@ops` into a single variable because they are treated the same way by downstream computations.

From another point of view, computation frames are 'views' of the stored computation graph, analogous to database views. In particular, they may contain multiple references to the same `Ref` or `Call` object from different nodes of the graph, and do not necessarily contain all calls to a given `@op`.

3.2 Formal Definition

A computation frame (Figure 2) consists of the following data:

- **Computation graph**: a directed graph $G = (V, F, E)$ where V are named variables and F are named instances of `@op`-decorated functions. The edges E are labeled with the input/output names of the adjacent functions. An example is shown in Figure 2 (c);
- **Groups of `Refs` and `Calls`**: for each variable $v \in V$, a set of (history IDs of) `Refs` R_v, and for each function $f \in F$ with underlying `@op` o_f, a set of (history IDs of) `Calls` C_f;

subject to the constraint that: for every call $c \in C_f$, if there's an input/output edge labeled l connecting f to some variable v, then if c has a `Ref` r_l corresponding to input/output name l, we have $r_l \in R_v$.

In other words, when we look at all calls in $f \in F$, their inputs/outputs must be present in the variables connected to f under the respective input/output name.

3.3 Basic Usage

The main advantage of computation frames is that they allow iterative exploration of the computation graph, and high-level grouped operations over computations with some shared structure. For example, we can use them for:

- **Iteratively expanding the frame with functions that generated or used existing variables**: this is useful for exploring the computation graph in a particular direction, or for adding more context to a particular computation. For example, in Figure 2 (a), we start with a computation frame containing only the calls to `f`, and then expand it to include all calls that can be reached from the memoized calls to `f` via their inputs/outputs, which adds the calls to `g` to the frame.
- **Converting the frame into a dataframe**: this is useful at the end of an exploration, when we want to get a convenient tabular representation of the captured computation graph. The table is obtained by collecting all terminal `Refs` in the frame's computational graph (i.e., those that are not inputs to any function in the frame), computing their computational history in the frame (grouped by variable), and joining the resulting tables over the variables. This is shown in Figure 2 (right). In particular, as shown in the example, this step may produce nulls, as the computation frame can contain computations that only partially follow the graph.

- **Performing high-level storage manipulations**: such as deleting all calls captured in the frame as well as all calls that depend on them, available using the `.delete_calls()` method on the frame.

Computation frames are a powerful tool for exploring and manipulating the stored computation graph, and we're excited to explore their full potential in future work.

4 Main Extra Features

4.1 Data Structures

Python's native collections — lists, dicts, sets — can be memoized transparently by `mandala`, using customized type annotations, e.g. `MList[int]` inheriting from `List[int]`, ... By applying this type annotation, individual elements as well as the collection itself are memoized as `Ref`s (with the collection merely pointing to the `Ref`s of its elements to avoid duplication).

```
@op
def avg_items(xs: MList[int]) -> float:
  return sum(xs) / len(xs)

@op
def get_xs(n) -> MList[int]:
  return list(range(n))

with storage:
  xs = get_xs(10)
  for i in range(2, 10, 2):
    avg = avg_items(xs[:i])
```

Figure 3: Illustration of native collection memoization in `mandala`. The custom type annotation `MList[int]` is used to memoize a list of integers as a list of pointers to element `Ref`s.

This is implemented fully on top of the core memoization machinery, using 'internal' `@op`s like e.g. `__make_list__` which, given the elements of a list as variadic inputs, generates a `ListRef` (subclass of `Ref`) that points to the `Ref`s of the elements. In this way, collections are naturally incorporated in the computation graph. These internal `@op`s are applied automatically when a collection is passed as an argument to a memoized function, or when a collection is returned from a memoized function (Figure 3).

4.2 Versioning

It is crucial to have a flexible and powerful code versioning system in a data management tool, as it allows the user to keep track of the evolution of their computations, and to easily recover from mistakes. `mandala` provides the option to use a versioning system with three main features:

- **Per-function content-addressed versioning** (Torvalds et al., 2005; Lozano, 2017), where the version of a function is a hash of its source code and the hashes of the functions it calls. The storage can determine, based only on the state of the codebase, whether a given call is up-to-date or not.
- **Dynamic dependency tracking**, where each function call traces the dependencies it calls. This avoids the need for static analysis of the code to find dependencies, which can result in many false positives and negatives, especially in a dynamic language like Python. Moreover, it provides a stronger notion of reusability, as certain changes of the codebase may invalidate only a part of all memoized calls to an `@op`.
- **The flexibility to mark changes as backward-compatible or not**, which allows the user to maintain a stable interface to computations when performing routine refactoring or adding logging/debugging code.

5 Related Work

`mandala` combines ideas from several existing projects, but is unique in the Pythonic way it makes complex memoized computations easy to query, manipulate and version.

Memoization. There are several memoization solutions for Python that lack the compositional nature of `mandala`, as well as the versioning and querying tools: the builtin `functools`

module provides decorators such as `lru_cache` for memoization; the `incpy` project (Guo & Engler, 2011) enables automatic persistent memoization of Python functions directly on the interpreter level; the `funsies` project (Lavigne & Aspuru-Guzik, 2021) is a memoization-based distributed workflow executor that uses a similar hashing approach to keep track of which computations have already been done; `koji` (Maymounkov, 2018) is a design for an incremental computation data processing framework that unifies over different resource types (files or services), and uses an analogous notion of hashing to keep track of computations.

Computation Frames. Computation frames are closely related to the relational model (Codd, 1970), to graph databases, and to certain versatile in-memory data structures based on functors $\mathcal{F} : \mathscr{C} \to \underline{\mathbf{Set}}$ where $\mathscr{C}$ is a finite category (Patterson et al., 2022).

Versioning. The `unison` programming language (Lozano, 2017) uses a content-addressed system for code storage, where a function is identified by the hash of its syntax tree. The language shares many other features and goals with `mandala`, such as use of serializable values and pure functions to ensure reproducibility.

The dynamic tracing mechanism used to capture dependencies is somewhat similar to the `@tf.function` decorator in the TensorFlow library (Abadi et al., 2016), which traces the function calls to other `@tf.function`-decorated functions made during execution and builds a computation graph out of them. Unlike `@tf.function`, `mandala`'s versioner uses content (code) hashes to automatically detect changes in dependencies, and does not build a fine-grained model of a function's execution, but rather only tracks the set of its dependencies.

The revision history of each function in the codebase is organized in a bare-bones `git` repository (Torvalds et al., 2005): it is a content-addressed tree, where each edge tracks a diff from the content at one endpoint to that at the other. Additional metadata indicates equivalence classes of semantically equivalent contents.

6 Limitations

Computing deterministic content IDs of any Python object is difficult. `mandala` uses the `joblib` library to serialize Python objects into byte strings, and then hashes these strings to get the content ID. This approach is not perfect, as it is not always possible to serialize Python objects, and even when it is, the serialization may not be unique. For example, two Python objects `x, y` which satisfy `x == y` may not have the same content ID (e.g., `True` and `1`). Furthermore, hashing is sensitive to small changes in the input, such as numerical precision in floating point numbers. Finally, complex Python objects may contain state that is not intrinsically part of the object's identity, such as resource utilization data (e.g., memory addresses). This can lead to different content IDs before and after a round trip through the storage backend. These issues don't come up often as long as all initial `Ref`s are created from simple Python objects: complex objects are hashed and saved once when returned from an `@op`, and then referred to by their content ID.

Non-breaking versioning is difficult. The ability to mark code changes as backward-compatible or not may lead to situations where the storage 'believes' that a call is up-to-date, but in reality it is not. For example, a function `f` can be changed by extracting a subroutine `g` out of it. The semantics of `f` is unchanged, so past calls are still valid, but `g` is now an invisible (to the storage) dependency of `f`. Care should be taken to avoid such situations until an automatic solution is implemented.

7 Conclusion

`mandala` is being actively developed, and has the potential to considerably simplify the way scientific data is managed and interacted with in Python. The author has already used it extensively to manage several multi-month machine learning projects, and has found it to be a very powerful tool for managing complex computations. We hope that this paper has given a good overview of the core concepts of `mandala`, and that the reader will be interested in exploring the library further.

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pp. 265–283, 2016. doi: 10.5281/zenodo.4724125.

Josh Abramson, Jonas Adler, Jack Dunger, Richard Evans, Tim Green, Alexander Pritzel, Olaf Ronneberger, Lindsay Willmore, Andrew J Ballard, Joshua Bambrick, et al. Accurate structure prediction of biomolecular interactions with alphafold 3. *Nature*, pp. 1–3, 2024. doi: 10.1038/s41586-024-07487-w.

Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. 1987. doi: 10.1109/MC.1987.1663532. URL `https://api.semanticscholar.org/CorpusID:372277`.

Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970. doi: 10.1145/362384.362685.

Susan B Davidson and Juliana Freire. Provenance and scientific workflows: challenges and opportunities. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1345–1350, 2008. doi: 10.1145/1376616.1376772.

Philip J Guo and Dawson Engler. Using automatic persistent memoization to facilitate data analysis scripting. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pp. 287–297, 2011. doi: 10.1145/2001420.2001455.

Tony Hey, Stewart Tansley, Kristin Michele Tolle, et al. *The fourth paradigm: data-intensive scientific discovery*, volume 1. Microsoft research Redmond, WA, 2009. doi: 10.1007/978-3-642-33299-9_1.

Peter Ivie and Douglas Thain. Reproducibility in scientific computing. *ACM Computing Surveys (CSUR)*, 51(3):1–36, 2018. doi: 10.1145/3186266.

Cyrille Lavigne and Alán Aspuru-Guzik. funsies: A minimalist, distributed and dynamic workflow engine. *Journal of Open Source Software*, 6(66):3274, 2021. doi: 10.21105/joss.03274.

Roberto Castañeda Lozano. The unison manual, 2017.

Petar Maymounkov. Koji: Automating pipelines with mixed-semantics data sources. *arXiv preprint arXiv:1901.01908*, 2018.

Peter Norvig. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17(1):91–98, 1991.

Evan Patterson, Owen Lynch, and James Fairbanks. Categorical data structures for technical computing. *Compositionality*, 4, 2022. doi: 10.32408/compositionality-4-5.

Daniele Ravì, Charence Wong, Fani Deligianni, Melissa Berthelot, Javier Andreu-Perez, Benny Lo, and Guang-Zhong Yang. Deep learning for health informatics. *IEEE journal of biomedical and health informatics*, 21(1):4–21, 2016. doi: 10.1109/JBHI.2016.2636665.

Geir Kjetil Sandve, Anton Nekrutenko, James Taylor, and Eivind Hovig. Ten simple rules for reproducible computational research. *PLoS computational biology*, 9(10):e1003285, 2013. doi: 10.1371/journal.pcbi.1003285.

Linus Torvalds et al. *Git*, 2005. URL `https://git-scm.com/`. Version control system.

Hadley Wickham. Tidy data. *Journal of statistical software*, 59:1–23, 2014. doi: 10.18637/jss.v059.i10.

Mark D Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E Bourne, et al. The fair guiding principles for scientific data management and stewardship. *Scientific data*, 3(1):1–9, 2016. doi: 10.1038/sdata.2016.18.