The Management of Context in the Machine Learning Lifecycle

Rolando Garcia

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-142
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-142.html

June 25, 2024

Copyright © 2024, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

The Management of Context in the Machine Learning Lifecycle

by

Rolando Garcia

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Joseph M. Hellerstein, Chair
Professor Joseph Gonzalez
Professor Koushik Sen
Associate Professor Fernando Perez

Summer 2024

The Management of Context in the Machine Learning Lifecycle

Copyright 2024
by
Rolando Garcia

Abstract

The Management of Context in the Machine Learning Lifecycle

by

Rolando Garcia

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Joseph M. Hellerstein, Chair

We present novel techniques and systems for managing data context within the machine learning (ML) lifecycle. Drawing from a vision laid out in 2018, we present Flor and its evolutions, FlorDB and FlorDB with Build extensions, designed for comprehensive metadata capture and version control in the ML lifecycle. A cornerstone of our approach is the use of an interview study to understand what the ML lifecycle is, and how engineers operationalize machine learning, focusing on MLOps and the iterative model development process. Through the implementation of these systems and their use in real-world applications for lawyers and journalists, we demonstrate the tangible benefits of rich data context in agile model development. In sum, we show how the integration of Application, Build, and Change contexts—The ABCs of Context—enables MLEs to close the loop in the ML lifecycle.

To my grandfather, who worked as a citrus and cherry farmworker in California's Central Valley despite being an academic at heart, and to countless talented others who toil and are toiling without prospects, so that some distant day their children and grandchildren may enjoy better opportunities. To my mother and father, who invested all that they could into my education, and taught me to prioritize school over work. To my wife, Victoria, and son, Joshua, who remind me every day what matters most, and show me the way forward.

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I extend my deepest gratitude to my advisor, Professor Joe Hellerstein, for his exceptional mentorship and unwavering support throughout my doctoral journey. Professor Hellerstein's guidance was pivotal not only in shaping my academic and professional pursuits but also in inspiring me with his passion for distributed data-intensive systems and education. His ability to distill complex concepts into understandable insights has profoundly influenced my own approach to research and mentorship. I am especially thankful for his patience, encouragement, and the countless hours he invested in ensuring my work was both rigorous and meaningful. His exemplary role as a scholar and educator has left an indelible mark on my career and personal journey. Thank you, Professor Hellerstein, for being an extraordinary mentor and an even greater role model.

I would also like to thank my committee members Joey Gonzalez, Koushik Sen, and Fernando Perez for all their encouragement, guidance, and support, especially in the early years of graduate school. Their innovative research has significantly influenced my own research interests, and I continually look to their work for inspiration. Their guidance has been invaluable in navigating the challenges of graduate studies and in helping me develop a robust foundation for my academic career.

I would like to thank professors Aditya Parameswaran and Sarah Chasins for their mentorship and support. Their expertise in user-focused design and qualitative research methods was instrumental in the shaping of our interview study, and played a substantial role in helping me grow professional connections with lawyers and journalists, increasingly more important stakeholders in the AI race. I would like to thank my peers, collaborators, and co-authors, who made this work possible: Shreya Shankar, Gabriel Matute, Yifan Wu, Daniel Crankshaw, Neeraja Yadwadkar, Vikram Sreekanti, Doris Xin, Eric Liu, Anusha Dandamudi, Bobby Yan, Lehan Wan, Isabella Mendoza, Sona Jeswani, Malhar Patel, Pragya Kallanagoudar, and Chithra Anand.

Finally, I would like to thank the National Science Foundation for a Graduate Research Fellowship which provided three years of funding, the UC Berkeley Chancellor's office for an additional two-year fellowship, and Dell Inc for one year of funding. I would like to thank RISE Lab sponsors: Intel, Nvidia, VMWare, Google, Microsoft, Meta, Amazon, and Splunk; as well as EPIC lab sponsors: G-Research, Adobe, Microsoft, Google, and Sigma Computing.

Chapter 1

Introduction

In this dissertation[1], we explore the management of context in the machine learning lifecycle. We begin by defining "the machine learning lifecycle", akin to a conveyor belt for ML models, and then go on to define "context" and talk about some of the pain points that arise when it is lost. We discuss reasons why context is so varied and easy to lose in the ML lifecycle, and then go on to present a solution: `flor`, a context management system for the machine learning lifecycle. Flor and its evolutions, FlorDB and FlorDB with Build extensions, were developed incrementally over the years; the system is available in open source[2], and is pip-installable as `flordb`. In what follows, we lay out the argument for why context is so important for managing and streamlining the machine learning lifecycle, and discuss the technical contributions that have made its management possible.

1.1 Build, Train, Deploy: What is the Machine Learning Lifecycle?

The machine learning lifecycle encompasses a three-phase process: (i) building out a pipeline, (ii) training a model using that pipeline, and (iii) deploying the model into production. Variants of this lifecycle include AWS SageMaker's monitoring step, MLFlow's detailed breakdown of each phase, and in Chapter 2 the ML lifecycle is denoted in four steps, consisting of (i) data preparation on a schedule, (ii) experimentation, (iii) evaluation & deployment, and (iv) monitoring & response (Figure 2.1).

Model development, within the *build* phase, is iterative and empirical. During this phase, an ML Engineer typically works with an offline snapshot of data, performing tasks such as data cleaning, exploratory data analysis, and model training and validation. The process is inherently iterative, as engineers continuously refine their models. A participant from the interview study presented in Chapter 2 remarked:

[1]Talk available at `https://youtu.be/yjIH9l-jTz0`

[2]`github.com/ucbrise/flor`

"People have principled stances or intuitions for why things should work, but the number one thing is to achieve scary high experimentation velocity."

In the training phase, data processing pipelines run automatically on a schedule, ingesting fresh data periodically to train updated versions of the model. The MLE evaluates new models against historical averages and decides whether to deploy them to incrementally larger fractions of the population.

In the final deployment phase, on-call ML Engineers monitor data and model predictions to ensure quality at scale. The models, deployed as part of an intelligent application or service, are a small part of a large ecosystem. Model predictions may be piped to downstream models to build hierarchical systems, or "pipeline jungles," where an end-user application queries the prediction service and receives a response. Another participant from the interview study in Chapter 2 noted:

"We don't have a good idea of how models will behave in production until production."

In the machine learning lifecycle, feedback plays a crucial role by circling back to earlier stages, effectively "closing the loop." This feedback is integrated with evaluation datasets and unit tests to ensure consistent training behavior. However, our interview study, detailed in Chapter 2, uncovered several significant challenges in this process:

- **This data looks wrong!** Practitioners often encounter unexpected discrepancies in their datasets. As one interviewee (P16) noted, "This data is supposed to have 50 states, but there's only 40. What happened to the other 10?"

- **The feedback is late!** Timely feedback is crucial, yet often elusive. P7 expressed frustration with this delay: "Feedback is always delayed by at least 2 weeks [...] so when we realize maybe something went wrong, it could have been 2 weeks ago."

- **It worked better yesterday!** Maintaining consistent model performance over time is challenging. P19 shared a common strategy for dealing with unexpected performance drops: "If a new model's performance drops and an older model is still performing within a valid range, we rollback and just cut the losses."

- **I fixed it, who needs to know?** The disconnect between different teams involved in the ML process can lead to information silos. P10 highlighted this issue: "You have to bridge the gap between what's often a subject matter expert in one room annotating and then handing things over the wire to a data scientist — a scene where you have no communication." Improving cross-functional collaboration is essential for effective feedback integration.

- **They don't work here anymore...** As team members change over time, valuable knowledge about system components can be lost. P14 described a common scenario:

“We have some filters we keep around so we don’t break something but we’re not so sure what they do. Filters can accumulate over time into pipeline jungles.” This accumulation of poorly understood elements can lead to increasingly complex and opaque systems.

These challenges underscore the need for robust processes and tools to effectively manage feedback in the machine learning lifecycle, ensuring that insights are promptly and accurately incorporated into ongoing development efforts.

1.2 Context: The Missing Piece in the Machine Learning Lifecycle

The ML lifecycle is characterized by numerous fast-changing components, where it is easy to lose the thread of essential metadata — what we term *context*. Context is metadata broadly conceived: it represents a comprehensive framework that captures the nature, origins, evolution, and functional significance of data and digital artifacts within an organization. Our conceptualization of context is drawn from the work of Hellerstein et al. (2017) [62], who proposed it as an extension of traditional database metadata. They introduced the “ABCs of Context” mnemonic, which we find particularly useful for understanding and navigating the complex landscape of ML metadata. This framework provides a structured approach to capturing and managing the rich tapestry of information that underpins real-world ML applications. The ABCs of Context are as follows:

A. **Application Context (What)**: This is core information that describes **what** raw bits an application sees and interprets for its use. In its most comprehensive definition (which we will adopt here), this involves any information that *could be* logged; i.e., the value of arbitrary expressions at runtime.

B. **Build Context (How)**: This is information about **how** data is created and used: e.g., dependency management for distribution and building across different machines and by different people; provenance and lineage; routes, pathways, or branches in pipelines; and flow of control and data.

C. **Change Context (When)**: This is information about the version **history** of data, code, configuration parameters, and associated information, including changes over time to both structure and content.

An unfortunate consequence of the disaggregated nature of contemporary ML applications is the lack of a standard mechanism to assemble a collective understanding of the origin, scope, and usage of the data they manage. What is needed is a common service layer to support the capture, publishing and sharing of metadata information in a flexible way: Flor aims to fill this gap in a lightweight fashion, as envisioned in Chapter 3. By rethinking metadata in a far more comprehensive and open sense, Flor is built from the ground up to capture the full context of data in the machine learning lifecycle.

In a decoupled architecture of multiple applications and backend services, context serves as a narrow gateway — a single point of reference for the basic information about data and its usage. In our vision laid out in Chapter 3, Flor streamlines ML engineers' workflows by integrating fragmented metadata from scattered systems within a single, unified, lightweight solution. This ensures MLEs work quickly and efficiently while maintaining long-term organization and ensuring essential context is always captured.

1.3 Flor: The Management of Context in the Machine Learning Lifecycle

Flor represents a significant advancement in the field of machine learning, offering a comprehensive system designed to capture, manage, and utilize context throughout the entire machine learning lifecycle. Flor draws its name from the Spanish word for "flower," aptly symbolizing its organic growth and evolution from its predecessor, Ground, a pioneering data context service [62]. Just as a flower blossoms from the ground, Flor builds upon and expands the foundational concepts established by Ground, offering a more robust and versatile approach to context management in machine learning workflows.

Managing Application Context with Hindsight Logging

To understand application context, consider the following example: a model training script `train.py` that computes a loss, back-propagates gradients, updates weights, and so on. Application context involves the core information that describes how raw data is interpreted for use — essentially, the value and meaning of expressions at runtime. The application context of `train.py` includes:

- The hyper-parameters, seeds, and other configuration settings.
- The batch of training, validation, and test data.
- The loss, activations, tensor histograms, and other model data.
- The accuracy, recall, F1-score, and other metrics.

This list could continue indefinitely. Application context is anything that *could be* logged. When considering derived data from added expressions, application context becomes potentially infinite. How do you manage something infinite?

The approach presented in Chapter 4 is to manage application context *virtually*, and allow for its structure and content to be defined post-hoc, at query time. In a Flor Dataframe, each logging statement executed at runtime maps to a column. A Flor Dataframe is *virtually* infinite because log statements (and hence columns) can be added or defined post-hoc, using hindsight logging. Hindsight logging is a record-replay technique for materializing application context on-demand. In Chapter 4, we introduce the first iteration of Flor and describe the technical contributions that enable hindsight logging for model training, including: i) adaptive checkpointing in the background, and ii) Low latency and auto-parallel replay from checkpoint-resume.

Managing Change over Time with Multiversion Hindsight Logging

In Chapter 5 we consider extensions to our management of application context to allow us to track change over time. The way we motivate the management of temporal or change context, is through the following continuous training scenario:

Alice, an MLE at a self-driving car company, aims to improve self-driving performance in countries with many roundabouts. She tracks standard metrics such as the F1-score and loss, ensuring they remain within valid parameters while optimizing performance on roundabouts. Over time, Bob informs Alice that her models have lost the ability to detect pedestrians, an incident of "catastrophic forgetting." Alice needs to know when this issue began and roll back to an earlier version when the model performed well on roundabouts without losing pedestrian detection capabilities.

Unfortunately, Alice was not tracking metrics on pedestrians. The solution presented in Chapter 5 allows Alice to add the necessary logging statements to the latest version of her `train.py` script and replay from the beginning of history. This approach involves automatically versioning the code on every run and using software patching to propagate logging statements back in time, calling flor replay on each modified version of `train.py`. These extensions, combined with the relational model described in Section 5.4 (Figure 5.6), enable Alice to extend hindsight logging to multiple versions and track the evolution of application context over time.

Context is All You Need: Closing the Loop in the Machine Learning Lifecycle

In Chapter 6, we introduce the final extension to Flor, FlorDB with Build extensions, which integrates support for managing *build* context. This allows Flor to handle the full ABCs of context comprehensively. Build context encompasses information on the creation and utilization of data, including: dependency management across various machines and users, provenance and lineage of data, various routes or branches in computational pipelines, the flow of control and data.

By default, FlorDB employs Make to manage *build* context. However, it is designed to be flexible and can integrate with the preferred workflow or build management system of a machine learning engineer (MLE), including but not limited to Airflow and MLFlow. In Chapter 6, we showcase FlorDB's applicability across document intelligence scenarios, illustrating how it effectively closes the loop in the machine learning lifecycle.

FlorDB operates through a user-friendly logging interface (`flor.log`) and accesses stored data seamlessly via the `flor.dataframe` interface, regardless of the file or version where the value originated. This approach channels the previously disparate and disorganized metadata from the machine learning lifecycle through a narrow gateway, organizing it into a structured and coherent format.

1.4 Flow with FlorDB

Context is all you need because context is *so much*. Application context alone is effectively unbounded, and once you can manage build and change context alongside it, you harness the full spectrum of data and process insights. This comprehensive grasp enables MLEs to iterate on the ML lifecycle with velocity, adopting an agile but still strong "metadata later" approach. FlorDB not only documents and preserves every step of the model's development and deployment but also provides a dynamic interface to review, rewind, and fast-forward through the model's lifecycle. This capability ensures that every component of the ML system is transparent and traceable.

- **Enhanced Transparency**: By maintaining detailed logs of each phase of the ML lifecycle, Flor provides an exhaustive audit trail that helps in compliance and debugging.
- **Improved Collaboration**: Flor's logging mechanisms enable team members to understand changes made by others deeply, fostering better collaboration and reducing the risks of errors or redundant work.
- **Faster Iteration Cycles**: With the ability to rapidly access and review past states, MLEs can quickly iterate on models, experimenting with new ideas while confidently managing the risk of negative changes.

FlorDB's comprehensive management of context effectively "closes the loop" in the ML lifecycle by ensuring that feedback from deployed models can be seamlessly integrated back into model training and development phases. Through FlorDB, we provide not just the tools to manage context, but an agile hindsight logging framework to understand and utilize it post-hoc, making strides in our vision of a fully integrated, transparent, and efficient ML lifecycle.

Chapter 2

Operationalizing Machine Learning: An Interview Study

Organizations rely on machine learning engineers (MLEs) to deploy models and maintain ML pipelines in production. Due to models' extensive reliance on fresh data, the operationalization of machine learning, or MLOps, requires MLEs to have proficiency in data science and engineering. When considered holistically, the job seems staggering—how do MLEs do MLOps, and what are their unaddressed challenges? To address these questions, we conducted semi-structured ethnographic interviews with 18 MLEs working on various applications, including chatbots, autonomous vehicles, and finance. We find that MLEs engage in a workflow of (i) data preparation, (ii) experimentation, (iii) evaluation throughout a multi-staged deployment, and (iv) continual monitoring and response. Throughout this workflow, MLEs collaborate extensively with data scientists, product stakeholders, and one another, supplementing routine verbal exchanges with communication tools ranging from Slack to organization-wide ticketing and reporting systems. Later in this chapter[1], we introduce the 3Vs of MLOps: *velocity*, *visibility*, and *versioning*—three virtues of successful ML deployments that MLEs learn to balance and grow as they mature. Finally, we discuss design implications and opportunities for future work.

2.1 Introduction

As machine learning (ML) models are increasingly incorporated into software, a nascent subfield called *MLOps* (short for ML Operations) has emerged to organize the "set of practices that aim to deploy and maintain ML models in production reliably and efficiently" [203, 4]. It is widely recognized that MLOps issues pose challenges to organizations. Anecdotal reports claim that 90% of ML models don't make it to production [202]; others claim that 85% of ML projects fail to deliver value [181]—signaling the fact that translating ML models

[1]Chapter published as a CSCW conference paper by Shankar & Garcia et al. [177]. Shankar & Garcia contributed equally to this research, and are listed as co-first authors.

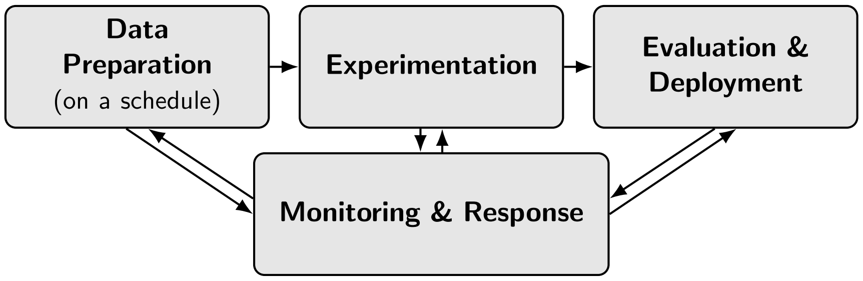


Figure 2.1: Core tasks in the MLOps workflow. Prior work discusses a production data science workflow of preparation, modeling, and deployment [200]. Our work exposes (i) the scheduled and recurring nature of **data preparation** (including automated ML tasks, such as model retraining), identifies (ii) a broader **experimentation** step (which could include modeling or adding new features), and provides more insight into human-centered (iii) **evaluation & deployment**, and (iv) **monitoring & response**.

to production is difficult. At the same time, it is unclear *why* MLOps issues are difficult to deal with. Our present-day understanding of MLOps is limited to a fragmented landscape of white papers, anecdotes, and thought pieces [42, 125, 46, 52], as well as a cottage industry of startups aiming to address MLOps issues [75]. Early work by Sculley et al. (2015) attributes MLOps challenges to *technical debt*, analogous to that in software engineering but exacerbated in ML [169]. Prior work has studied general practices of data scientists working on ML [165, 81, 131, 213], but successful ML deployments seem to further involve a "team of engineers who spend a significant portion of their time on the less glamorous aspects of ML like maintaining and monitoring ML pipelines" —that is, ML engineers (MLEs) [147]. It is well-known that MLEs typically need to have strong data science and engineering skills [6], but it is unclear how those skills are used in their day-to-day workflows.

There is thus a pressing need to bring clarity to MLOps—specifically in identifying what MLOps typically involves—across organizations and ML applications. While papers on MLOps have described specific case studies, prescribed best practices, and surveyed tools to help automate the ML lifecycle, there is a pressing need to understand the *human-centered* workflow required to support and sustain the deployment of ML models in practice. A richer understanding of common practices and challenges in MLOps can surface gaps in present-day processes and better inform the development of next-generation ML engineering tools. To address this need, we conducted a semi-structured interview study of ML engineers (MLEs), each of whom has been responsible for a production ML model. With the intent of identifying common themes across organizations and industries, we sourced 18 ML engineers from different companies and applications, and asked them open-ended questions to understand their workflow and day-to-day challenges—both on an individual and organizational level.

Prior work focusing on the earlier stages of data science has shown that it is a largely iterative and manual process, requiring humans to perform several stages of data cleaning, exploration, model building, and visualization [141, 91, 165, 66]. Before embarking on our study, we expected that the subsequent deployment of ML models in production would instead be more amenable to automation, with less need for human intervention and supervision. Our interviews, in fact, revealed the opposite—much like the earlier stages of data science, deploying and maintaining models in production is highly iterative, manually-intensive, and team-oriented. Our interviewees emphasized organizational and collaborative strategies to sustain ML pipeline performance and minimize pipeline downtime, mentioning on-call rotations, manual rules and guardrails, and teams of practitioners inspecting data quality alerts.

In this chapter, we provide insight into human-centered aspects of MLOps practices and identify opportunities for future MLOps tools. We conduct a semi-structured interview study solely focused on ML engineers, an increasingly important persona in the broader software development ecosystem as more applications leverage ML. Our focus on MLEs, and uncovering their workflows and challenges as part of the MLOps process, addresses a gap in the literature. Through our interviews, we characterize an ML engineer's typical workflow (on top of automated processes)into four stages (Figure 2.1): (i) data preparation, (ii) experimentation, (iii) evaluation and deployment, and (iv) monitoring and response, all centered around team-based, collaborative practices. Key takeaways for each stage are as follows:

Data ingestion often runs automatically, but MLEs drive data preparation through data selection, analysis, labeling, and validation (Section 2.4) We find that organizations typically leverage teams of data engineers to manage recurring end-to-end executions of data pipelines, allowing MLEs to focus on ML-specific steps such as defining features, a retraining cadence, and a labeling cadence. If a problem can be automated away, engineers prefer to do so—e.g., retraining models on a regular cadence to protect against changes in the distribution of features over time. Thus, they can spend energy on tasks that require human input, such as supervising crowd workers who provide input labels or resolve inconcistencies in these labels.

Even in production, experimentation is highly iterative and collaborative, despite the use of model training tools and infrastructure (Section 2.4) As mentioned earlier, various articles claim that it is a problem for 90% of models to never make it to production [202], but we find that this statistic is misguided. The nature of constant experimentation is bound to create many versions of models, a small fraction of which (i.e. "the best of the best") will make it to production. MLEs discussed exercising judgment when choosing next experiments to run, and expressed reservations about AutoML tools, or "keeping GPUs warm," given the vast search space. MLEs consult domain experts and stakeholders in group meetings, and prefer to iterate on the data (e.g., to identify new feature ideas) over innovating on model architectures.

Organizations employ a multi-stage model evaluation and deployment process, so MLEs manually review and authorize deployment to subsequent stages (Section 2.4) Textbook model evaluation "best practices" do not do justice to the rigor with which organizations think about deployments: they generally focus on using one typically-static held-out dataset in an offline manner to evaluate the model on [107] and a single ML metric choice (e.g., precision, recall) [135]. We find that many MLEs carefully deploy changes to increasing fractions of the population in stages. At each stage, MLEs seek feedback from other stakeholders (e.g., product managers and domain experts) and invest significant resources in maintaining multiple up-to-date evaluation datasets and metrics over time—especially to ensure that data sub-populations of interest are adequately covered.

MLEs closely monitor deployed models and stand by, ready to respond to failures in production (Section 2.4) MLEs ensured that deployments were reliable via strategies such as on-call rotations, data monitoring, and elaborate rule-based guardrails to avoid incorrect outputs. MLEs discussed pain points such as alert fatigue from alerting systems and the headache of managing pipeline jungles [169], or amalgamations of various filters, checks, and data transformations added to ML pipelines over time.

The rest of this chapter is organized as follows: we cover background and work related to MLOps from the CSCW, HCI, ML, software engineering, and data science communities (Section 2.2). Next, we describe the methods used in our interview study (Section 2.3). Then, we present our results and discuss our findings, including opportunities for new tooling (Section 2.4 and Section 2.5). Finally, we conclude with possible areas for future work.

2.2 Related Work

Our work builds on previous studies of data and ML practitioners in industry. We begin with the goal of characterizing the role of an ML Engineer, starting with related data science roles in the literature and drawing distinctions that make MLEs unique. We then review work that discusses data science and ML workflows, not specific to MLOps. Third, we cover challenges that arise from productionizing ML. Fourth, we survey software engineering practices in the literature that tackle such challenges. Finally, we review recent work that explicitly attempts to define and discuss MLOps practices.

Characterizing the ML Engineer

Data science roles span various engineering and research tasks [86], and many data-related activities are performed by people without "data" or "ML" in their job title [85], so it can be hard to clearly define job descriptions [131]. Nonetheless, since we focus on production ML pipelines, we discuss personas related to data science, ML, and engineering—culminating in the description of the persona we study.

The Data Scientist: Multiple studies have identified subtypes of data scientists, some of whom are more engineering-focused than others [213, 86]. Zhang et al. (2021) describe the many roles data scientists can take—communicator, manager/executive, domain expert, researcher/scientist, and engineer/analyst/programmer [213]. They found considerable overlap in skills and tasks performed between the (i) engineer/analyst/programmer and (ii) researcher/scientist roles: both are highly technical and collaborate extensively. Separately, Kim et al. taxonomized data scientists as: insight providers, modeling specialists, platform builders, polymaths, and team leaders [86]. Modeling specialists build predictive models using ML, and platform builders balance both engineering and science as they produce reusable software across products.

The Data Engineer: While data scientists engage in activities like exploratory data analysis (EDA), data wrangling, and insight generation [81, 205], *data engineers* are responsible for building robust pipelines that regularly transform and prepare data [100]. Data engineers often have a software engineering and data systems background [86]. In contrast, data scientists typically have modeling, storytelling, and mathematics backgrounds [86, 100]. Since production ML systems involve data pipelines and ML models in larger software services, they require a combination of data engineering, data science, and software engineering skills.

The ML Engineer (MLE): Our interview study focuses on the distinct *ML Engineer* (MLE) persona. MLEs have a multifaceted skill set: they know how to transform data as inputs to ML pipelines, train ML models, serve models, and wrap these pipelines in software repositories [89, 80, 157]. MLEs need to regularly process data at scale (much like data engineers [89]), employing statistics and ML techniques as do data scientists [145], and are responsible for production artifacts as are software engineers [103]. Unlike typical data scientists and data engineers, MLEs are responsible for deploying ML models and maintaining them in production.

We classify production ML into two modes. One, which we call *single-use* ML, is more client-oriented, where the focus is to generate predictions once to make a specific data-informed business decision [91]. Typically, this involves producing reports, primarily performed by data scientists [70, 85]. In the other mode, which we call *repeated-use* ML, predictions are repeatedly generated, often as intermediate steps in data pipelines or as part of ML-powered products, such as voice assistants and recommender systems [86, 6]. Continuously generating ML predictions over time requires more data and software engineering expertise [85, 190]. In our study, we focus on MLEs who work on the latter mode of production ML.

Machine Learning Workflows

Here, we cover literature on ML practitioners' workflows. We discuss both technical and collaborative workflows, and then we describe the workflow our study seeks to uncover.

Several studies have investigated aspects of the broader ML workflow, mostly in single-use production ML applications. Early studies on data science workflows point to industry-originated software development process models, such as the Agile framework [29] and the CRoss Industry Standard Process for Data Mining (CRISP-DM) [24]. More recently, Studer et al. (2021) introduce CRISP-ML, a new process model that augments CRISP-DM with a final "monitoring and maintenance" phase to support ML workflows [186]. Muller et al.

(2019) interview practitioners and focus on the data practices of data science workflows, breaking them down into discovery, capture, design, and curation [131]. Wongsuphasawat et al. (2019) workflow includes some ML: it consists of data acquisition, wrangling, exploration, modeling, and reporting [205]. Wang et al. (2019) takes another step back and includes productionization; they identify three high-level phases of preparation, modeling, and deployment [200]. Preparation includes activities ranging from wrangling [82] to feature engineering [145]. Modeling includes selection, hyperparameter optimization, ensembling, and validation, and deployment includes monitoring and improvement [200, 213]. Of the three large stages, several studies have identified preparation as the most time-intensive stage of the workflow [131, 60], where data scientists commonly iterate on rules to help generate features [142, 66].

While the above stages of the data science workflow comprise a loop of technical tasks, Kross and Guo (2021) identify an *outer loop* of data science, centered around collaborative practices [91]. The outer loop consists of groundwork (i.e., building trust), orienting, problem framing, magic (i.e., technical loop), and counseling. While Kross and Guo's loop focuses on data science work that directly interacts with clients, mostly in the form of single-use ML, similar themes emerge when performing repeated-use ML engineering work, e.g., repeatedly generating ML predictions in an automated fashion. In production settings, predictions must yield value for the business [145], requiring some groundwork, orienting, and problem framing. In their paper on tensions around collaborative, applied data science work, Passi and Jackson (20198) discuss that it's important to align different stakeholders on system performance metrics: for example, one of their interviewees mentioned that accuracy is a "problematic" metric because different users interpret it differently [141]. In another example, Holstein et al. (2020) say that a single global metric doesn't capture performance for certain groups of users (e.g., accuracy for a subgroup might decrease when overall accuracy increases) [68].

In our study, we characterize the workflow from a repeated-use ML engineering perspective, focusing on specific practices within deployment stages. Some related work defines steps in the ML workflow, such as model training and model monitoring, through both short papers [119] and extensive literature reviews [94]. We take a different but complementary approach: like Muller et al (2019) who focus on data scientists [131], we conduct an interview study of MLEs, using grounded theory to analyze our findings [184]. Further, our study seeks to uncover collaborative practices and challenges, focusing on the ML engineering perspective, and how MLEs align all stakeholders such that ML systems continually generate value.

Production ML Challenges

Sculley et al. (2015) were early proponents that production ML systems raise special challenges and can be hard to maintain over time, based on their experience at Google [169]. They coined the "Changing Anything Changes Everything" (CACE) principle: if one makes a seemingly innocuous change to an ML system, such as changing the default value for a feature from 0 to -1, the entire system's behavior can drastically change. CACE easily creates technical debt and is often exacerbated as errors "cascade," or compound, throughout an end-to-end pipeline [169, 165, 140, 146, 147]. We cover three well-studied challenges in production ML: data quality, reproducibility, and specificity.

First, ML predictions are only as good as their input data [146, 147], requiring active efforts from practitioners to ensure good data quality [20]. Xin et al. (2021) observe that production ML pipelines consist of models that are automatically retrained, and we find that this retraining procedure is a pain point for practitioners because it requires constant monitoring of data [208]. If a model is retrained on bad data, all future predictions will be unreliable. Data distribution shift is another known data-related challenge for production ML systems [151, 138, 128, 204, 187], and our work builds on top of the literature by reporting on how practitioners tackle shift issues.

Next, reproducibility in data science workflows is a well-understood challenge, with attempts to partially address it [22, 74, 88, 38]. Recent work also indicates that reproducibility is an ongoing issue in data science and ML pipelines [10, 163, 84, 92]. Kross and Guo (2019) mention that data science educators who come from industry specifically want students to learn how to write "robust and reproducible scientific code." [92] In interview studies, Xin et al. (2021) observe the importance of reproducibility in AutoML workflows [209], and Sambasivan et al. (2021) mention that practitioners who create reproducible data assets avoid some errors [165].

Finally, other related work has identified that production ML challenges can be specific to the ML application at hand. For example, Sambasivan et al. (2021) discusses how, in high-stakes domains like autonomous vehicles, data quality is extra important and explicitly requires collaboration with domain experts. They explain how data errors compound and have disastrous impacts, especially in resource-constrained settings. Unlike the present study, their focus is on data quality issues as opposed to understanding typical MLE workflows and challenges. Paleyes et al. (2022) review published reports of individual ML deployments and mention that not all ML applications can be easily tested prior to deployment [140]. While ad recommender systems might be easily tested online with a small fraction of users, other applications require significant simulation testing depending on safety, security, and scale issues [139, 95]. Common applications of ML, such as medicine [148], customer service [44], and interview processing [18] , have their own studies. Our work expands on the literature by identifying common challenges across various applications and reporting on how MLEs handle them.

Software Engineering for ML

Through interviews and practitioner surveys, some papers explore, at a high level, how ML engineering practices differ from traditional software engineering practices. Hill et al. (2016) interview ML application developers and report challenges related to building first versions of ML models, especially around the early stages of exploration and experimentation (e.g., feature engineering, model training) [64]. They describe the process of building models as "magic"—similarly echoed by Lee et al. (2020) when analyzing ML projects from Github—with unique practices of debugging data in addition to code [98]. Serban et al. (2020) conduct a survey of practitioners and list 29 software engineering practices for ML, such as "Use Continuous Integration" and "Peer Review Training Scripts" [172]. Muiruri et al. (2022) interview Finnish engineers and investigate technical challenges and ML-specific tools in the ML lifecycle [129]. Amershi et al. (2019) identify challenges such as hidden feedback loops and component entanglement through their interviews with scientists, engineers, and managers at Microsoft [6]. They broadly discuss strategies to integrate support for ML development into traditional software infrastructure, such as end-to-end pipeline support from data engineers and educational conferences for employees. Our work expands on the software engineering for ML ecosystem by considering human-centered, operational requirements for ML deployments, e.g., over time, as MLEs are introduced to ML pipelines that are unfamiliar to them, or as customer or product requirements change. Unlike Amershi et al., we focus on MLEs, who are responsible for maintaining ML pipeline performance. We also interview practitioners across companies and applications: we provide new and specific examples of ML engineering practices to sustain ML pipelines as software and categorize these practices around a broader human-centered workflow.

The data management, software engineering, and CSCW communities have proposed various software tools for ML workflows. For example, some tools manage data provenance and training context for model debugging purposes [133, 19, 62, 47]. Others help ensure reproducibility while iterating on different ideas [178, 84, 66]. With regards to validating changes in *production* systems, some researchers have studied CI (Continuous Integration) for ML and proposed preliminary solutions—for example, `ease.ml/ci` streamlines data management and proposes unit tests for overfitting [3], and some papers introduce tools to perform validation and monitoring in production ML pipelines [20, 166, 83]. Our work is complementary to existing literature on this tooling; we do not explicitly ask interviewees questions about tools, nor do we propose any tools. We focus on behavioral practices of MLEs.

MLOps Practices and Challenges

The traditional software engineering literature describes the need for DevOps, a combination of software *dev*elopers and *op*erations teams, to streamline the process of delivering software in organizations [103, 41, 114, 111]. Similarly, the field of *MLOps*, or DevOps principles applied to machine learning, has emerged from the rise of machine learning (ML) application development in software organizations. MLOps is a nascent field, where most existing papers give definitions and overviews of MLOps, as well as its relation to ML, software engineering, DevOps, and data engineering [89, 189, 50, 118, 190, 80, 191, 157, 175, 173]. Some work in MLOps attempts to characterize a production ML lifecycle; however, there is little consensus. Symeonidis et al. (2022) discuss a lifecycle of data preparation, model selection, and model productionization [189], but other literature reviews [106, 50] and guides on best practices drawing from authors' experiences [119] conclude that, compared to software engineering, there is not yet a standard ML lifecycle, with consensus from researchers and industry professionals. While standardizing an ML lifecycle across different roles (e.g., scientists, researchers, business leaders, engineers) might be challenging, characterizing a workflow specific to a certain role could be more tractable.

Several MLOps papers present case studies of productionizing ML within specific organizations and the resulting challenges. For example, adhering to data governance standards and regulation is difficult, as model training is data-hungry by nature [14, 56]. Garg et al (2021) discuss issues in continuous end-to-end testing (i.e., continuous integration) because ML development involves changes to datasets and model parameters in addition to code [50]. To address such challenges, other MLOps papers have surveyed the proliferating number of industry-originated tools in MLOps [156, 189, 162, 106]. MLOps tools can help with general pipeline management, data management, and model management [156]. The surveys on tools motivate understanding how MLEs use such tools, to see if there are any gaps or opportunities for improvement.

Prior work in this area—primarily limited to literature reviews, surveys, case studies, and vision papers—motivates research in understanding the *human-centered* workflows and pain points in MLOps. Some MLOps work has interviewed people involved in the production ML lifecycle: for example, Kreuzberger et al. (2022) conduct semi-structured interviews with 8 experts from different industries spanning different roles, such as AI architect and Senior Data Platform Engineer, and uncover a list of MLOps principles such as CI/CD automation, workflow orchestration, and reproducibility, as well as an organizational workflow of product initiation, feature engineering, experimentation, and automated ML workflow pipeline [89]. While Kreuzberger et al. (2022) explicitly interview professionals from different roles to understand shared patterns between their workflows—in fact, only two of the eight interviewees have “machine learning engineer” or “deep learning engineer” in their job titles, our work complements their findings by focusing only on self-declared ML engineers responsible for repeated-use models in production and uncovering strategies they use to sustain model performance. As such, we uncover and present a different workflow—one centered around ML engineering. To the best of our knowledge, we are the first to study the human-centered MLOps workflow from ML engineers’ perspectives.

2.3 Methods

Upon securing approval from our institution’s review board, we conducted an interview study of 18 ML Engineers (MLEs) across various sectors. Our approach mirrored a zigzag model common to Grounded Theory, with alternating phases of fieldwork and in-depth coding and analysis, directing further rounds of interviews [34]. The constant comparative method helped iterate and refine our categories and overarching theory. Consistent with qualitative research standards, theoretical saturation is generally recognized between 12 to 30 participants, particularly in more uniform populations [58]. By our 16th interview, prevalent themes emerged, signaling that saturation was attained. Later interviews confirmed these

RR	Id	Role	Org Size	Application	Yrs Xp	Site	Highlights
1	Lg1	MLE Mgr.	Large	Autonomous vehicles	5-10	US-West	high velocity experimentation; scenario testing
1	Md1	MLE	Medium	Autonomous vehicles	5-10	US-West	pipeline-on-a-schedule; copy-paste anomalies
1	Sm1	MLE	Small	Computer hardware	10-15	US-West	exploratory data analysis; AB Testing; SLOs
1	Md2	MLE	Medium	Retail	5-10	US-East	retraining cadence; adaptive test data; feedback delay
1	Lg2	MLE Mgr.	Large	Ads	5-10	US-West	ad click count; model ownership; keeping GPUs warm
1	Lg3	MLE	Large	Cloud computing	10-15	US-West	bucketing / binning; SLOs; hourly batched predictions
2	Sm2	MLE	Small	Finance	5-10	US-West	F1-score ; retraining cadence; progressive validation
2	Sm3	MLE	Small	NLP	10-15	Intl	triage queue; fallback models; false-positive rate
2	Sm4	MLE	Small	OCR + NLP	5-10	Intl	human annotators; word2vec; airflow
3	Md3	MLE Mgr.	Medium	Banking	10-15	US-West	human annotators; institutional knowledge; revenue
3	Lg4	MLE	Large	Cloud computing	10-15	US-West	online inference; pipeline-on-a-schedule; fallback models
3	Sm5	MLE	Small	Bioinformatics	5-10	US-West	model fine-tuning; someone else's features
4	Md4	MLE	Medium	Cybersecurity	10-15	US-East	model-per-customer; join predictions w/ ground truth
4	Md5	MLE	Medium	Fintech	10-15	US-West	retraining cadence; dropped special characters
5	Sm6	MLE	Small	Marketing and analytics	5-10	US-East	human annotators; label quality; adaptive test data
5	Md6	MLE	Medium	Website builder	5-10	US-East	SLOs; poor documentation; data validation
6	Lg5	MLE	Large	Recommender systems	10-15	US-West	pipeline-on-a-schedule; SLOs; progressive validation
6	Lg6	MLE Mgr.	Large	Ads	10-15	US-West	fallback models; on-call rotations; scale

Table 2.1: The table provides an anonymized description of interviewees from different sizes of companies, roles, years of experience, application areas, and their code attributions. The interviewees hail from a diverse set of backgrounds. Small companies have fewer than 100 employees; medium-sized companies have 100-1000 employees, and large companies have 1000 or more employees. **RR** denotes recruitment round. **Highlights** refers to key codes (i.e. from the code system) attached to that participant's transcript.

themes.

Our goal in conducting this study was to develop better tools for ML deployment, broadly targeting monitoring, debugging, and observability issues. Our study was an attempt at identifying key challenges and open opportunities in the MLOps space. This study therefore stems from our collective desire to enrich our understanding of our target community and offer valuable insights into best practices in ML engineering and data science. Subsequent sections delve into participant recruitment (Subsection 2.3), our interview structure (Subsection 2.3), and our coding and analysis techniques (Subsection 2.3).

Participant Recruitment & Selection

We recruited individuals who were responsible for the development, regular retraining, monitoring and deployment of ML models *in production*. A description of the 18 MLEs is shown in Table 2.1. The MLEs interviewed varied in their educational backgrounds, years of experience, roles, team size, and work sector. Recruitment was conducted in rounds over the course of an academic year (2021-2022). Our recruitment strategy was underpinned by a deliberate, iterative process that built upon the insights from each round. The primary goal was to cultivate a representative sample that captured the rich diversity of Machine Learning Engineers (MLEs) across various dimensions.

Initial Recruitment: Relying on Professional Networks

In the initial recruitment round (RR=1), we leaned heavily on the established professional networks of our faculty co-authors. This approach, while convenient and efficient, resulted in a sample that was geographically skewed towards the US-West. It also led to a greater representation from larger organizations, as well as certain sectors like Autonomous Vehicles and Cloud Computing. This initial cohort provided valuable insights but also highlighted the potential biases and gaps in our sample.

Course Correction: Diversifying the Sample

Recognizing the need for a more representative and diversified sample, our strategy in subsequent rounds shifted. Specifically for RR=2, we made a concerted effort to engage candidates outside our immediate professional networks and particularly targeted those at smaller companies. This shift in approach was operationalized by posting recruitment advertisements and flyers in various online communities. Prospective participants who responded to our outreach underwent a screening process for the same inclusion criteria mentioned previously. Their professional backgrounds and affiliations were verified through platforms such as professional websites, LinkedIn, and online portfolios. As a result, we observed a stronger representation from domains like NLP and Finance.

Building a Representative Sample: Iterative Refinement

Each recruitment round served as a feedback loop, informing the strategy for the subsequent round. As patterns emerged from our data analysis, we fine-tuned our recruitment focus to fill identified gaps. This iterative process ensured that, over time, our sample grew to be more balanced in terms of roles, experience, organization sizes, sectors, and geographical locations. By employing this iterative recruitment strategy, we believe our study encapsulates a comprehensive cross-section of the MLE community, offering insights that are both deep and broad.

In each round, between three to five candidates were reached by email and invited to participate. We relied on our professional networks and open calls posted on MLOps channels in Discord[2], Slack[3], and Twitter to compile a roster of candidates. The roster was incrementally updated roughly after every round of interviews, integrating information gained from the concurrent coding and analysis of transcripts (Section 2.3). Recruitment rounds were repeated until we reached saturation on our findings [130].

Interview Protocol

With each participant, we conducted semi-structured interviews over video call lasting 45 to 75 minutes each. Over the course of the interview, we asked descriptive, structural, and

[2]https://discord.com/invite/Mw77HPrgjF

[3]mlops-community.slack.com

contrast questions abiding by ethnographic interview guidelines [179]. The questions are listed in **??**. Specifically, our questions spanned six categories: i) the type of ML tasks they work on, ii) the approaches used for building or tuning models, iii) the transition from development to production, iv) how they evaluate their models before deployment, v) how they monitor their deployed models, and vi) how they respond to issues or bugs. Participants received and signed a consent form before the interview, and agreed to participate free of compensation. As per our agreement, we automatically transcribed the interviews using Zoom software. In the interest of privacy and confidentiality, we did not record audio or video of the interviews. Transcripts were redacted of personally identifiable information before being uploaded to a secured drive in the cloud.

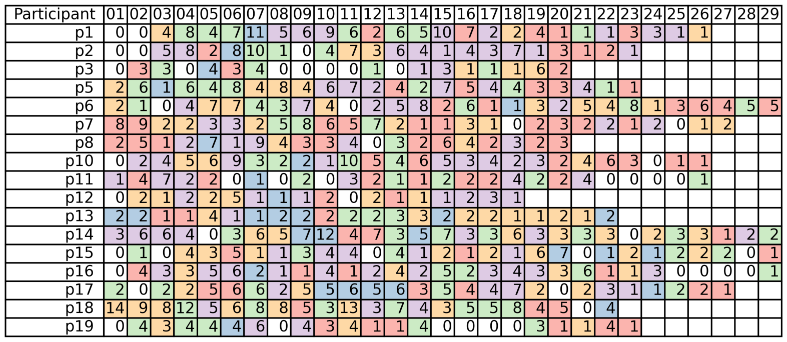
Transcript Coding & Analysis

We employed grounded theory to systematically analyze our interview transcripts. Grounded theory is a robust methodology focused on iterative data collection and analysis for theory discovery [184, 28]. One of its key features is the seamless integration of data collection and analysis, aiming to identify emerging themes and concepts through a constant review of transcripts. In employing grounded theory, we followed its key processes: open, axial, and selective coding. During *open coding*, the initial phase of categorizing data, we deconstructed our interview transcripts into discrete ideas or phenomena and assigned codes to these ideas (e.g., *A/B testing*). Then, in *axial coding*, where the goal is to identify patterns and relationships between different concepts, we merged duplicate codes and drew edges between similar codes. For example, we grouped the codes *scenario testing* and *A/B testing* under the broader *testing* code. Finally, through *selective coding*, we distilled our codes into five core themes that represent the essence of our transcripts. Figure 2.3 shows our hierarchy of codes, with core themes such as **Tasks**, **Operations**, and **Systems & Tools**. As illustrated in Figure 2.2, we allocated roughly equal time to each main theme, which correspondingly informed our findings. The themes relate to our findings as follows:

1. **Tasks** refers to activities that are routinely performed by ML engineers. The analysis of code segments descended from *tasks*, and its decomposition into constituent parts, culminated in the creation of the MLOps workflow (Figure 2.1), and is instrumental in the structure and presentation of Section 2.4 (*Findings*).

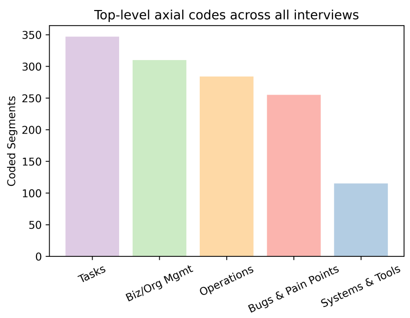
2. **Biz/Org (Business-Organizational) Management** refers to modes of interaction between MLEs and their co-workers or managers, and between MLEs and customers or other stakeholders. Relevant sub-codes form the theoretical basis for Section 2.4 (*Collaboration*) and Section 2.4 (*Product Metrics*).

3. **Operations** refers to repeatable work that must be performed regularly and consistently for the continued operation of the business. *Operations* is the "Ops" in MLOps. Relevant sub-codes form the theoretical basis for Section 2.4 (*Pipeline Automation*).



Participant	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
p1	0	0	4	8	4	7	11	5	6	9	6	2	6	5	10	7	2	2	4	1	1	1	3	3	1	1			
p2	0	0	5	8	2	8	10	1	0	4	7	3	6	4	1	4	3	7	1	3	1	2	1						
p3	0	3	3	0	4	3	4	0	0	0	0	1	0	1	3	1	1	1	6	2									
p5	2	6	1	6	4	8	4	8	4	6	7	2	4	2	7	5	4	4	3	3	4	1	1						
p6	2	1	0	4	7	7	4	3	7	4	0	2	5	8	2	6	1	1	3	2	5	4	8	1	3	6	4	5	5
p7	8	9	2	2	3	3	2	5	8	6	5	7	2	1	1	3	1	0	2	3	2	2	1	2	0	1	2		
p8	2	5	1	2	7	1	9	4	3	3	4	0	3	2	6	4	2	3	2	3									
p10	0	2	4	5	6	9	3	2	2	1	10	5	4	6	5	3	4	2	3	2	4	6	3	0	1	1			
p11	1	4	7	2	2	0	1	0	2	0	3	2	1	1	2	2	2	4	2	2	4	0	0	0	0	1			
p12	0	2	1	2	2	5	1	1	1	2	0	2	1	1	1	2	3	1											
p13	2	2	1	1	4	1	1	2	2	2	2	2	3	3	2	2	2	1	1	2	1	2							
p14	3	6	6	4	0	3	6	5	7	12	4	7	3	5	7	3	3	6	3	3	3	3	0	2	3	3	1	2	2
p15	0	1	0	4	3	5	1	1	3	4	4	0	4	1	2	1	2	1	6	7	0	1	2	1	2	2	2	0	1
p16	0	4	3	3	5	6	2	1	1	4	1	2	4	2	5	2	3	4	3	3	6	1	1	3	0	0	0	0	1
p17	2	0	2	2	5	6	6	2	5	5	6	5	6	3	5	4	4	7	2	0	2	3	1	1	2	2	1		
p18	14	9	8	12	5	6	8	8	5	3	13	3	7	4	3	5	5	8	4	5	0	4							
p19	0	4	3	4	4	4	6	0	4	3	4	1	1	4	0	0	0	0	3	1	1	4	1						

(a) Color-coded Overview of Transcripts



(b) Color Legend

Figure 2.2: Visual summary of coded transcripts. The x-axis of (a), the color-coded overview, corresponds to a segment (or group) of transcript lines, and the number in each cell is the code's frequency for that transcript segment and for that participant. Segments are blank after the conclusion of each interview, and different interviews had different time duration. Each color in (a) is associated with a top-level axial code from our interview study, and presented in the color legend (b). The color legend also shows the frequency of each code across all interviews.

4. **Bugs & Pain Points** refers to failure modes encountered at any stage in the MLOps workflows, MLE complaints generally, and author-noted anti-patterns. These are discussed in Monitoring and Response (Section 2.4).

5. **Systems & Tools** refers to storage and compute infrastructure, programming languages, ML training frameworks, experiment execution frameworks, and other tools or systems that MLEs use in their MLOps work. We discuss implications for tool design in Section 2.5. We include a table of common tools referenced by interviewees in **??**.

1. **Tasks**
   a) Data collection, cleaning & labeling: *human annotators*, *exploratory data analysis*
   b) Embeddings & feature engineering: *normalization*, *bucketing / binning*, *word2vec*
   c) Data modeling & experimentation: *accuracy*, *F1-score*, *precision*, *recall*
   d) Testing: *scenario testing*, *AB testing*, *adaptive test-data*
2. **Biz/Org Management**
   a) Business focus: *service level objectives*, *ad click count*, *revenue*
   b) Teams & collaboration: *institutional knowledge*, *on-call rotations*, *model ownership*
   c) Process maturity indicators: *pipeline-on-a-schedule*, *fallback models*, *model-per-customer*
3. **Operations**
   a) CI/CD: *artifact versioning*, *multi-staged deployment*, *progressive validation*
   b) Data ingestion & integration: *automated featurization*, *data validation*
   c) Model retraining: *distributed training*, *retraining cadence*, *model fine-tuning*
   d) Prediction serving: *hourly batched predictions*, *online inference*
   e) Live monitoring: *false-positive rate*, *join predictions w/ ground truth*
4. **Bugs & Pain Points**
   a) Bugs: *data leakage*, *dropped special characters*, *copy-paste anomalies*
   b) Pain points: *big org red tape*, *performance regressions*, *label quality*, *scale*
   c) Anti-patterns: *muting alerts*, *keeping GPUs warm*, *waiting it out*
   d) Known challenges: *data drift*, *feedback delay*, *class imbalance*, *sensor divergence*
   e) Missing context: *someone else's features*, *poor documentation*, *much time has passed*
5. **Systems & Tools**
   a) Metadata layer: *Huggingface*, *Weights & Biases*, *MLFlow*, *TensorBoard*, *DVC*
   b) Unit layer: *PyTorch*, *TensorFlow*, *Jupyter*, *Spark*
   c) Pipeline layer: *Airflow*, *Kubeflow*, *Papermill*, *DBT*, *Vertex AI*
   d) Infrastructure layer: *Slurm*, *S3*, *EC2*, *GCP*, *HDFS*

Figure 2.3: Abridged code system: A distilled representation of the evolved code system resulting from our qualitative study, capturing the primary tasks, organizational aspects, operational methodologies, challenges, and tools utilized by Machine Learning Engineers.

2.4 Summary of Findings

Going into the interview study, we assumed the workflow of human-centered tasks in the production ML lifecycle was similar to the production data science workflow presented by Wang et al. (2019), which is a loop consisting of the following:

1. **Preparation**, spanning data acquisition, data cleaning and labeling, and feature engineering,

2. **Modeling**, spanning model selection, hyperparameter tuning, and model validation, and

3. **Deployment**, spanning model deployment, runtime monitoring, and model improvement.

From our interviews, we found that the repeated-use production ML workflow that ML engineers engage in differs slightly from the work by Wang et al. [200]. As preliminary research papers defining and providing reference architectures for MLOps have pointed out, operationalizing ML brings new requirements to the table, such as the need for teams, not just individual people, to understand, sustain, and improve ML pipelines and systems over time [89, 189, 50]. In the pipelines that our interviewees build and supervise, most technical components are automated—e.g., data preprocessing jobs run on a schedule, and models are typically retrained regularly on fresh data. We found the ML engineering workflow to revolve around the following stages (Figure 2.1):

1. **Data Preparation**, which includes scheduled data acquisition, cleaning, labeling, and transformation,

2. **Experimentation**, which includes both data-driven and model-driven changes to increase overall ML performance, and is typically measured by metrics such as accuracy or mean-squared-error,

3. **Evaluation & Deployment**, which consists of a model retraining loop which periodically rolls out a new model—or offline batched predictions, or another proposed change—to growing fractions of the population, and

4. **Monitoring & Response**, which supports the other stages via data and code instrumentation (e.g., tracking available GPUs for experimentation or the fraction of null-valued data points) and dispatches engineers and bug fixes to identified problems in the pipeline.

For each stage, we identified human-centered practices from the *Tasks*, *Biz/Org Management*, and *Bugs & Pain Points* codes, and drew on *Operations* codes for automated practices (refer to Section 2.3 for a description of these codes). An overview of findings for each workflow stage can be found in Table 2.2.

Stage	Description	Non-Automated Challenges
Data Preparation	Collection, wrangling, and cleaning pipelines run on a schedule	- Ensuring label quality at scale (Section 2.4) - Handling feedback or ground-truth delays (Section 2.4)
Experimentation	Prototyping ideas to improve end-to-end ML pipeline performance by iterating on datasets, model architectures, or both	- Managing the underlying software or code for data-centric experiments (Section 2.4) - Engaging in cross-team collaboration (Section 2.4) - Manually and thoughtfully identifying promising experiment configurations (Section 2.4)
Evaluation and Deployment	Pushing experimental changes to small, then large fractions of users, evaluating at every step	- Continuously updating dynamic validation datasets for future experiments (Section 2.4) - Using product metrics for evaluation (Section 2.4)
Monitoring and Response	Supervising live ML pipeline performance and minimizing pipeline downtime	- Tracking and investigating data quality alerts (Section 2.4) - Managing pipeline "jungles" of models and hard-coded rules (Section 2.4) - Debugging a heavy-tailed distribution of errors (Section 2.4)

Table 2.2: Overview of challenging activities that ML engineers engage in for each stage in their workflow. While each stage relies on automated infrastructure and pipelines, ML engineers still have many difficult manual responsibilities.

The following subsections organize our findings around the four stages of MLOps. We begin each subsection with a quote that stood out to us and conversation with prior work; then, in the context of what is automated, we discuss common human-centered practices and pain points.

Data Preparation

"It takes exponentially more data to keep getting linear increases in performance." –Lg1

Data preparation is the process of constructing "well-structured, complete datasets" for data scientists [131]. Data preparation activities consist of collection, wrangling, and cleaning and are known to be challenging, often taking up to 80% of practitioners' time [82, 200]. This tedious process encourages larger organizations to have dedicated teams of data engineers to manage data preparation [81]. Like Amershi et al. (2019), we observed that mature ML organizations automated data preparation through dedicated teams as much as possible (Lg1, Lg2, Lg3, Sm3, Md3, Sm6, Md6, Lg5, Lg6). As a result, the MLEs we interviewed spent a smaller fraction of their time on data preparation, collaborating instead with data engineering teams. We first discuss pipeline automation to provide key context for the work of MLEs. Then, we mention two key challenges MLEs face: ensuring labeling quality at scale and coping with feedback delays.

Pipelines automatically run on a schedule

Unlike data science, where data preparation is often ad-hoc and interactive [81, 131], we found that data preparation in production ML is batched and more narrowly restricted, involving an organization-wide set of steps running at a predefined cadence. In interviews, we found that preparation pipelines were defined by Directed Acyclic Graphs, or DAGs, which ran on a schedule (e.g., daily). Each DAG node corresponded to a particular task, such as ingesting data from a source or cleaning a newly ingested partition of data. Each DAG edge corresponded to a dataflow dependency between tasks. While data engineers were primarily responsible for the end-to-end execution of data preparation DAGs, MLEs interfaced with these DAGs by loading select outputs (e.g., clean data) or by extending the DAG with additional tasks, e.g. to compute new features (Md1, Lg2, Lg3, Sm4, Md6, Lg6).

In many cases, automated tasks relating to *ML models*, such as model inference (e.g., generating predictions with a trained model) and retraining, were executed in the same DAG as data preparation tasks (Lg1, Md1, Sm2, Md4, Md5, Sm6, Md6, Lg5). ML engineers included retraining as a node in the data preparation DAG for simplicity: as new data becomes available, a corresponding model is automatically retrained. Md4 mentioned automatically retraining the model every day so model performance would not suffer for longer than a day:

> Why did we start training daily? As far as I'm aware, we wanted to start simple—we could just have a single batch job that processes new data and we wouldn't need to worry about separate retraining schedules. You don't really need to worry about if your model has gone stale if you're retraining it every day.

Retraining cadences ranged from hourly (Lg5) to every few months (Md6) and were different for different models within the same organization (Lg1, Md4). None of the participants interviewed reported any scientific procedure for determining the pipeline execution cadence. For example, Md5 said that "the [model retraining] cadence was just like, finger to the wind." Cadences seemed to be set in a way that streamlined operations for the organization in the easiest way. Lg5 mentioned that "retraining took about 3 to 4 hours, so [they] matched the cadence with it such that as soon as [they] finished any one model, they kicked off the next training [job].” Engineers reported an inability to retrain unless they had fresh and labeled data, motivating their organizations to set up dedicated teams of annotators, or hiring crowd workers, to operationalize labeling of live data (Lg1, Sm3, Sm4, Md3, Sm6).

MLEs ensure label quality at scale

Although it is widely recognized that model performance improves with more labels [161]—and there are tools built especially for data labeling [153, 215]—our interviewees cautioned that the quality of labels can degrade as they try to label more and more data. Md3 said:

> No matter how many labels you generate, you need to know what you’re actually labeling and you need to have a very clear human definition of what they mean.

In many cases, ground truth must be *created*, i.e., the labels are what a practitioner “thinks up” [131]. When operationalizing this practice, MLEs faced problems. For one, Sm3 spoke about how expensive it was to outsource labeling. Moreover, labeling conflicts can erode trust in data quality, and slow ML progress [141, 165]: When scaling up labeling—through labeling service providers or analysts within the organization—MLEs frequently found disagreements between different labelers, which would negatively impact quality if unresolved (Sm3, Md3, Sm6). At their organization, Sm3 mentioned that there was a human-in-the-loop labeling pipeline that both outsourced large-scale labeling and maintained an internal team of experts to verify the labels and resolve errors manually. Sm6 described a label validation pipeline for outsourced labels that itself used ML models for estimating label quality.

Feedback delays can disrupt pipeline cadences

In many applications, today’s predictions are tomorrow’s training data, but many participants said that ground-truth labels for live predictions often arrived after a delay, which could vary unpredictably (e.g., human-in-the-loop or networking delays) and thus caused problems for knowing real-time performance or retraining regularly (Md1, Sm2, Sm3, Md5, Md6, Lg5). This is in contrast to academic ML, where ground-truth labels are readily available for ML practitioners to train models [140]. Participants noted that the impact on models was hard to assess when the ground truth involved live data—for example, Sm2 felt strongly about the negative impact of feedback delays on their ML pipelines:

> I have no idea how well [models] actually perform on live data. Feedback is always delayed by at least 2 weeks. Sometimes we might not have feedback...so when we realize maybe something went wrong, it could [have been] 2 weeks ago.

Feedback delays contribute to “data cascades,” or compounding errors in ML systems over time [165]. Sm3 mentioned a 2-3 *year* effort to develop a human-in-the-loop pipeline to manually label live data as frequently as possible to side-step feedback delays: “you want to come up with the rate at which data is changing, and then assign people to manage this rate roughly". Sm6 said that their organization hired freelancers to label "20 or so" data points by hand daily. Labeling was then considered a task in the broader preparation pipeline that ran on a schedule (Section 2.4).

Experimentation

> "You want to see some degree of experimental thoroughness. People will have principled stances or intuitions for why things should work. But the most important thing to do is achieve scary high experimentation velocity...Number one [Key Performance Indicator] is rate of experimentation." –Lg1

While most prior work studying the data science and MLOps workflows includes *modeling* as an explicit step in the workflow [200, 213, 205, 186], we found that iterating on model ideas and architectures is only part of a broader "experimentation" step. This is because in many production ML pipelines, MLEs can focus on tuning or improving existing models through data-centric development, and *modeling* in a data science sense is only necessary when the company wishes to expand its service offerings or grow its ML capabilities. In fact, many of our interviewees did not build the initial model in the pipeline that their organization assigned them to work on, so their goal wasn't necessarily to train more models. As an example, Md6 said, "some of our models have been around for, like, 6 or 7 years." Garg et al. (2021) also call this workflow step "experimentation" instead of "modeling" in their MLOps lifecycle overview, and we expand on this finding in this chapter by relating it to collaboration and data-driven exploration, as well as MLE reservations toward experiment automation or AutoML [50].

MLEs find it better to innovate on the data than the model

Holstein et al. (2020) mention that it is challenging for practitioners to determine where to focus experimentation efforts—they could try "switching to a different model, augmenting the training data in some way, collecting more or different kinds of data, post-processing outputs, changing the objective function, or something else" [68]. Our interviewees recommended focusing on experiments that provided additional context to the model, typically via new features, to get the biggest gains (Lg2, Lg3, Md3, Lg4, Md4, Sm6, Md6, Lg5, Lg6). Lg5 said:

> I'm focusing my energy these days on signals and feature engineering because even if you keep your code static and the model static, it would definitely help you with getting better model performance.

In a concurring view, Md3 adds:

> I'm gonna start with a [fixed] model because it means faster iterations. And often—like most of the time empirically—it's going to be something in our data that we can use to push the boundary [. . .] obviously, it's not a dogmatic "we will never touch the model" but it shouldn't be our first move.

Interestingly, older work claims that iterating on the model is often more fruitful than iterating on the data [145], but this could be because ML modeling libraries weren't as mature as they are now. Recent work has also identified the importance of data-centric experimentation in production ML deployments [165, 140, 131, 153]. Md6 mentioned that most ML projects at their organization centered around adding new features. Md4 mentioned that one of their current projects was to move feature engineering pipelines from Scala to SparkSQL (a language more familiar to ML engineers and data scientists), so experiment ideas could be coded and validated faster.

When asked how they managed the underlying software or code for data-centric experiments, interviewees emphasized the importance of keeping their code changes as small as possible for multiple reasons, including faster code review, easier validation, and fewer merge conflicts (Md1, Lg2, Lg3, Sm4, Md3, Lg5, Lg6). This is in line with good software engineering practices [6]. Additionally, changes in large organizations were primarily made in configuration (config) files instead of main application code (Lg1, Md1, Lg2, Sm4, Lg4, Lg6): instead of editing parameters directly in a Python model training script, MLEs preferred to edit a config file of parameters (e.g., JSON or YAML), and would feed the config file to the model training script. When larger changes were necessary, especially changes touching the language layer (e.g. changing PyTorch or TensorFlow architectures), MLEs would fork the code base and made their edits in-place (Md2, Lg3). Although forking repositories can be a high-velocity shortcut, absent streamlined merge procedures, this can lead to a divergence in versions and accumulation of technical debt. Lg3 highlighted the tension between experiment velocity and strict software engineering discipline:

> I used to see a lot of people complaining that model developers don't follow software engineering. At this point, I'm feeling more convinced that it's not because they're lazy. It's because [software engineering is] contradictory to the agility of analysis and exploration.

Feature engineering is social and collaborative

Prior work has stressed the importance of collaboration in data science projects, often lamenting that technical tasks happen in silos [91, 165, 200, 140]. Our interviewees similarly believed cross-team collaboration was critical for successful experiments. Project ideas, such as new features, came from or were validated early by domain experts: data scientists and analysts who had already performed a lot of exploratory data analysis. Md4 and Md6 independently recounted successful project ideas that came from asynchronous conversations on Slack: Md6 said, "I look for features from data scientists, [who have ideas of] things that are correlated with what I'm trying to predict." We found that organizations explicitly prioritized cross-team collaboration as part of their ML culture. Md3 said:

> We really think it's important to bridge that gap between what's often, you know, a [subject matter expert] in one room annotating and then handing things over the wire to a data scientist—a scene where you have no communication. So we make sure there's both data science and subject matter expertise representation [in our meetings].

To foster a more collaborative culture, Sm6 discussed the concept of "building goodwill" with other teams through tedious tasks that weren't always explicitly a part of company plans: "Sometimes we'll fix something [here and] there to like build some goodwill, so that we can call on them in the future...I do this stuff [to build relationships], not because I'm really passionate about doing it."

MLEs like having manual control over experiment selection

One challenge that results from fast exploration is having to manage many experiment versions [200, 89]. MLEs are happy to delegate experiment tracking and execution work to ML experiment execution frameworks, such as Weights & Biases[4], but prefer to choose subsequent experiments themselves. To be able to make informed choices of subsequent experiments to run, MLEs must maintain awareness of what they have tried and what they haven't (Lg2 calls it the "exploration frontier"). As a result, there are limits to how much automation MLEs are willing to rely on for experimentation, a finding consistent with results from prior work [209]. Lg2 mentioned the phrase "keeping GPUs warm" to characterize a perceived anti-pattern that wastes resources without a plan:

> One thing that I've noticed is, especially when you have as many resources as [large companies] do, that there's a compulsive need to leverage all the resources that you have. And just, you know, get all the experiments out there. Come up with a bunch of ideas; run a bunch of stuff. I actually think that's bad. You can be overly concerned with keeping your GPUs warm, [so much] so that you don't actually think deeply about what the highest value experiment is.

In executing experiment ideas, we noticed a tradeoff between a guided search and random search. Random searches were more suited to parallelization—e.g., hyperparameter searches or ideas that didn't depend on each other. Although computing infrastructure could support many different experiments in parallel, the cognitive load of managing such experiments was too cumbersome for participants (Lg2, Sm4, Lg5, Lg6). Rather, participants noted more success when pipelining learnings from one experiment into the next, like a guided search to find the best idea (Lg2, Sm4, Lg5). Lg5 described their ideological shift from random search to guided search:

[4]https://wandb.ai/

> Previously, I tried to do a lot of parallelization. If I focus on one idea, a week at a time, then it boosts my productivity a lot more.

By following a guided search, engineers are, essentially, significantly pruning a large subset of experiment ideas without executing them. While it may seem like there are unlimited computational resources, the search space is much larger, and developer time and energy is limited. At the end of the day, experiments are human-validated and deployed. Mature ML engineers know their personal tradeoff between parallelizing disjoint experiment ideas and pipelining ideas that build on top of each other, ultimately yielding successful deployments.

Evaluation and Deployment

> "We don't have a good idea of how the model is going to behave in production until production." –Lg3

After the experimentation phase, when MLEs have identified a change they want to make to the ML pipeline (e.g., adding a new feature), they need to evaluate it and deploy it to production. Prior work that prescribes frameworks for MLOps typically separates evaluation and deployment into two different stages [189, 186, 94], but we combine them into one step because they are tightly intertwined, with deployments spanning long periods of time and evaluations happening multiple times during deployment.

Prior work describes evaluation as an "offline," automated process that happens at training time: a small portion of the training dataset is held out, and the model should achieve high accuracy on this held-out set [140, 205]. Recent related work in MLOps claims that evaluation and deployment are highly amenable to automation [50, 119]. As such, we also originally hypothesized that evaluation and deployment could be automated—once validated, an engineer could simply create a new task in their DAG to retrain the model on a cadence (Section 2.4).

As expected, engineers did automatically validate and codify their changes into DAGs to retrain models on a schedule. However, they also manually supervised the deployment over a long period of time, evaluating throughout the time frame. Amershi et al. (2019) state that software teams "flight" changes or updates to ML models, often by testing them on a few cases prior to live deployment [6]. Our work provides further context into the evaluation and deployment process for production ML pipelines: we found that several organizations, particularly those with many customers, employed a *multi-stage deployment process* for new models or model changes, progressively evaluating at each stage (Sm1, Lg2, Lg3, Sm2, Sm3, Lg4, Md5, Md6, Lg5, Lg6). As such, we combine evaluation and deployment into one step, rather than separating the process into evaluation followed by deployment as other papers do [189, 186]. Lg3 described the multi-staged deployment process as follows:

> We have designated test clusters, [stage 1] clusters, [stage 2] clusters, then the global deployment [to all users]. The idea here is you deploy increasingly along these clusters, so that you catch problems before they've met customers.

Each organization had different names for its stages (e.g., test, dev, canary, staging, shadow, A/B) and different numbers of stages in the deployment process (usually between one and four). The stages helped invalidate models that might perform poorly in full production, especially for brand-new or business-critical cases. Occasionally, organizations had an offline "sandbox" stage preceding deployment to any fraction of customers—for example, Md5 described a sandbox where they could stress-test their chatbot product:

> You can pretend to be a customer and say all sorts of offensive things, and see if the model will say cuss words back at you, or other sorts of things like that.

Although the model retraining process was automated, we find that MLEs personally reviewed validation metrics and manually supervised the promotion from one stage to the next. They had oversight over every evaluation stage, taking great care to manage complexity and change over time: specifically, changes in data, product and business requirements, users, and teams within organizations. We discuss two human-centered practices: maintaining dynamic datasets and evaluating performance in the context of the product or broader organizational value.

MLEs continuously update dynamic validation datasets

Many engineers reported processes to analyze live failure modes and update the validation datasets to prevent similar failures from happening again (Lg1, Md1, Lg2, Lg3, Sm3, Md3, Md5, Sm6, Md6, Lg5). Lg1 described this process as a departure from what they had learned in school:

> You have this classic issue where most researchers are evaluat[ing] against fixed data sets [. . . but] most industry methods change their datasets.

We found that these dynamic validation sets served two purposes: (1) the obvious goal of making sure the validation set stays current with live data as much as possible, given new knowledge about the problem and general shifts in the data distribution, and (2) the more specific goal of addressing localized shifts within sub-populations, such as low accuracy for minority groups. The challenge with (2) is that many sub-populations are often overlooked, or they are discovered post-deployment [68]. In response, Md3 discussed how they systematically bucketed their data points based on the model's error metrics and created validation sets for each under-performing bucket:

> Some [of the metrics in my tool] are standard, like a confusion matrix, but it's not really effective because it doesn't drill things down [into specific subpopulations that users care about]. Slices are user-defined, but sometimes it's a little bit more automated. [During offline evaluation, we] find the error bucket that [we] want to drill down, and then [we] either improve the model in very systematic ways or improve [our] data in very systematic ways.

Rather than follow a proactive approach of constructing different failure modes in an offline validation phase like Md3 did, Sm3 offered a reactive strategy of spawning a new dataset for each observed live failure: "Every [failed prediction] gets into the same queue, and 3 of us sit down once a week and go through the queue...then our [analysts] collect more [similar] data." This dataset update (or delta) was then merged into the validation dataset, and used for model validation in subsequent rounds. While processes to dynamically update the validation datasets ranged from human-in-the-loop to periodic synthetic data construction (Lg3), we found that higher-stakes applications of ML (e.g., autonomous vehicles), created dedicated teams to manage the dynamic evaluation process. Often, this involved creating synthetic data representative of live failures (Lg1, Lg3, Md4). For example, Lg1 said:

> What you need to be able to do in a mature MLOps pipeline is go very quickly from user recorded bug, to not only are you going to fix it, but you also have to be able to drive improvements to the stack by changing your data based on those bugs.

Notwithstanding, participants found it challenging to collect the various kinds of failure modes and monitoring metrics for each mode. Lg6 added, "you have to look at so many different metrics. Even very experienced folks question this process like a dozen times."

MLEs use product metrics for validation

While prior work discusses how prediction accuracy doesn't always correlate with real-world outcomes [200, 140], it's unclear how to articulate clear and measurable ML goals. Patel et al. (2008) discuss how practitioners trained in statistical techniques "felt that they must often manage concerns outside the focus of traditional evaluation metrics" [145]. Srivastava et al. (2020) note that an increase in accuracy might not improve overall system "compatibility." In our study, we found that successful ML deployments tied their performance to product metrics. First, we found that *prior to initial deployment*, mature ML teams defined a product metric in consultation with other stakeholders, such as business analysts, product managers, or customers (Lg2, Sm2, Md5, Sm6, Md3, Md6, Lg5, Lg6). Examples of product metrics include click-through rate and user churn rate. Md3 felt that a key reason many ML projects fail is that they don't measure metrics that will yield the organization value:

> Tying [model performance] to the business's KPIs (key performance indicators) is really important. But it's a process—you need to figure out what [the KPIs] are, and frankly I think that's how people should be doing AI. It [shouldn't be] like: hey, let's do these experiments and get cool numbers and show off these nice precision-recall curves to our bosses and call it a day. It should be like: hey, let's actually show the same business metrics that everyone else is held accountable to to our bosses at the end of the day.

Since product-specific metrics are, by definition, different for different ML models, it was important for engineers to treat the choice of metrics as an explicit step in their workflow and align with other stakeholders to make sure the right metrics were chosen. After agreeing on a product metric, engineers only promoted experiment ideas to later deployment stages if there was an improvement in that metric. Md6 said that every model change in production was validated by the product team: "if we can get a statistically significant greater percentage [of] people to subscribe to [the product], then [we can fully deploy]." Kim et al. (2016) also highlight the importance of including other stakeholders (or people in "decision-making seats") in the evaluation process [86]. At each stage of deployment, some organizations placed additional emphasis on important customers during evaluation (Lg3, Sm4). Lg3 mentioned that there were "hard-coded" rules for "mission-critical" customers:

> There's an [ML] system to allocate resources for [our product]. We have hard-coded rules for mission critical customers. Like at the start of COVID, there were hospital [customers] that we had to save [resources] for.

Finally, participants who came from research or academia noted that tying evaluation to the product metrics was a different experience. Lg3 commented on their "mindset shift" after leaving academia:

> I think about where the business will benefit from what we're building. We're not just shipping fake wins, like we're really in the value business. You've got to see value from AI in your organization in order to feel like it was worth it to you, and I guess that's a mindset that we really ought to have [as a community].

Monitoring and Response

> "This data is supposed to have 50 states, there's only 40, what happened to the other 10?" –Md6

We found that organizations centered their monitoring and response practices around engineers, much like in the DevOps agile framework, which organizes software development around teams [29]. Prior work has stated that monitoring is critical to MLOps [189, 89, 119, 186], and, broadly, that Agile practices can be useful in supervising production ML [6]. We provide further insight by discussing two specific examples of Agile practices that our interviewees commonly adapted to the ML context. First, Lg3, Lg4, Md4, Sm6, Lg5, and Lg6 described *on-call processes* for supervising production ML models. For each model, at any point in time, some ML engineer would be on call, or primarily responsible for it. Any bug or incident observed (e.g., user complaint, pipeline failure) would receive a ticket, created by the on-call engineer. On-call rotations typically lasted one or two weeks. At the end of a shift, an engineer would create an incident report—possibly one for each bug—detailing major issues that occurred and how they were fixed. Additionally, Lg3, Sm2, Sm4, and Md5 discussed having *Service-Level Objectives (SLOs)*, or commitments to minimum standards of performance, for pipelines in their organizations. For example, a pipeline to classify images could have an SLO of 95% accuracy. A benefit of using the SLO framework for ML pipelines is a clear indication of whether a pipeline is performing well or not—if the SLO is not met, the pipeline is broken, by definition.

Our interviewees stressed the importance of logging data across all stages of the ML pipeline (e.g., feature engineering, model training) to use for future debugging. Monitoring ML pipelines and responding to bugs involved tracking live metrics (via queries or dashboards), slicing and dicing sub-populations to investigate prediction quality, patching the model with non-ML heuristics for known failure modes, and finding in-the-wild failures that could be added to future dynamic validation datasets. While MLEs tried to automate monitoring and response as much as possible, we found that solutions were lacking and required significant human-in-the-loop intervention. Next, we discuss data quality alerts, pipeline jungles, and diagnostics.

On-call MLEs track data quality alerts and investigate a fraction of them

In data science, data quality is of utmost importance [141, 81]. Prior work has stressed the importance of monitoring data in production ML pipelines [169, 165, 86], and the data management literature has proposed numerous data quality metrics [20, 146, 166]. But what metrics do practitioners actually use, what data do practitioners monitor, and how do they manually engage with these metrics? We found that engineers continuously monitored features for and predictions from production models (Lg1, Md1, Lg3, Sm3, Md4, Sm6, Md6, Lg5, Lg6) Md1 discussed hard constraints for feature columns (e.g., bounds on values), Lg3 talked about monitoring completeness (i.e., fraction of non-null values) for features, Sm6 mentioned embedding their pipelines with "common sense checks," implemented as hard constraints on columns, and Sm3 described schema checks—making sure each data item adheres to an expected set of columns and their types. These checks were automated and executed as part of the larger pipeline (Section 2.4).

While off-the-shelf data validation was definitely useful for the participants, many of them expressed pain points with existing techniques and solutions. Lg3 discussed that it was hard to figure out how to trigger alerts based on data quality:

> Monitoring is both metrics and then a predicate over those metrics that triggers alerts. That second piece doesn't exist—not because the infrastructure is hard, but because no one knows how to set those predicate values...for a lot of this stuff now, there's engineering headcount to support a team doing this stuff. This is people's jobs now; this constant, periodic evaluation of models.

We also found that employee turnover makes data validation unsustainable (Sm2, Md4, Sm6, Md6, Lg5). If one engineer manually defined checks and bounds for each feature and then left the team, another engineer would have trouble interpreting the predefined data validation criteria. To circumvent this problem, some participants discussed using black-box data monitoring services but lamented that their statistics weren't interpretable or actionable (Sm2, Md4).

Another commonly discussed pain point was *false-positive alerts*, or alerts triggered even when the ML performance is adequate. Engineers often monitored and placed data quality alerts on each feature and prediction (Lg2, Lg3, Sm3, Md3, Md4, Sm6, Md6, Lg5, Lg6). If the number of metrics tracked grew too large, false-positive alerts could become a problem. An excess of false-positive alerts led to fatigue and silencing of alerts, which could miss actual performance drops. Sm3 said "people [were] getting bombed with these alerts." Lg5 shared a similar sentiment, that there was "nothing critical in most of the alerts." The only time there was something critical was "way back when [they] had to actually wake up in the middle of the night to solve it...the only time [in years]." When we asked what they did about the noncritical alerts and how they acted on the alerts, Lg5 said:

> You typically ignore most alerts...I guess on record I'd say 90% of them aren't immediate. You just have to acknowledge them [internally], like just be aware that there is something happening.

Seasoned MLEs thus preferred to view and filter alerts themselves, than to silence or lower the alert reporting rate. In a sense, even false-positives can provide information about system health, if the MLE knows how to read the alerts and is accustomed to the system's reporting patterns. When alert fatigue materialized, it was typically when engineers were on-call, or responsible for ML pipelines during a 7 or 14-day shift. Lg6 recounted how on-call rotations were dreaded amongst their team, particularly for new team members, due to the high rate of false-positive alerts. They said:

> On-call ML engineers freak out in the first 2 rotations. They don't know where to look. So we have them act as a shadow, until they know the patterns.

Lg6 also discussed an initiative at their company to reduce the alert fatigue, ironically with another model, which would consider how many times an engineer historically acted on an alert of a given type before determining whether to surface a new alert of that type.

Over time, ML pipelines may turn into "jungles" of rules and models

Sculley et al. (2015) introduce the phrase "pipeline jungles" (i.e., different versions of data transformations and models glued together), which was later adopted by participants in our study [169]. While prior work demonstrates their existence and maintenance challenges, we provide insight into why and how these pipelines become jungles in the first place. Our interviewees noted that reacting to an ML-related bug in production usually took a long time, motivating them to find strategies to quickly restore performance (Lg1, Sm2, Sm3, Sm4, Md4, Md5, Md6, Lg6). These strategies primarily involved adding non-ML rules and filters to the pipeline. When Sm3 observed, for an entity recognition task, that the model was misdetecting the Egyptian president due to the many ways of writing his name, they thought it would be better to patch the predictions for the individual case than to fix or retrain the model:

> Suppose we deploy [a new model] in the place of the existing model. We'd have to go through all the training data and then relabel it and [expletive], that's so much work.

One way engineers reacted to ML bugs was by adding filters for models. For the Egypt example, Sm3 added a simple string similarity rule to identify the president's name. Md4 and Md5 each discussed how their models were augmented with a final, rule-based layer to keep live predictions more stable. For example, Md4 mentioned working on an anomaly detection model and adding a heuristics layer on top to filter the set of anomalies that surface based on domain knowledge. Md5 discussed one of their language models for a customer support chatbot:

> The model might not have enough confidence in the suggested reply, so we don't return [the recommendation]. Also, language models can say all sorts of things you don't necessarily want it to—another reason that we don't show some suggestions. For example, if somebody asks when the business is open, the model might try to quote a time when it thinks the business is open. [It might say] "9 am," but the model doesn't know that. So if we detect time, then we filter that [reply]. We have a lot of filters.

Constructing such filters was an iterative process—Md5 mentioned constantly stress-testing the model in a sandbox, as well as observing suggested replies in early stages of deployment, to come up with filter ideas. Creating filters was a more effective strategy than trying to retrain the model to say the right thing; the goal was to keep some version of a model working in production with little downtime. As a result, filters would accumulate in the pipeline over time, effectively creating a pipeline jungle. Even when models were improved, Lg5 noted that it was too risky to remove the filters, since the filters were already in production, and a removal might lead to cascading or unforeseen failures.

Several engineers also maintained fallback models for reverting to: either older or simpler versions (Lg2, Lg3, Md6, Lg5, Lg6). Lg5 mentioned that it was important to always keep some model up and running, even if they "switched to a less economic model and had to just cut the losses." Similarly, when doing data science work, both Passi and Jackson (2018) and Wang et al. (2019) echo the importance of having some solution to meet clients' needs, even if it is not the best solution [141, 200]. Another simple solution engineers discussed was serving a separate model for each customer (Lg1, Lg3, Sm2, Sm4, Md3, Md4). We found that engineers preferred a per-customer model because it minimized downtime: if the service wasn't working for a particular customer, it could still work for other customers. Patel et al. (2018) also noted that per-customer models could yield higher overall performance [145].

Bugs in production ML follow a heavy-tailed distribution

ML debugging is different from debugging during standard software engineering, where one can write test cases to cover the space of potential bugs [140, 6]. Lg3, Sm2, Sm3, Sm4, Lg4, Md4, Md5, Sm6, Lg5, and Lg6 mentioned having a *central queue of production ML bugs* that every engineer added tickets to and processed tickets from. Often this queue was larger than what engineers could process in a timely manner, so they assigned tags to tickets to prioritize what to debug.

Interviewees discussed ad-hoc approaches to debugging production ML issues, which could require them to spend a lot of time diagnosing any given bug (Lg3, Lg2, Sm3, Sm4, Lg5). One common issue was *data leakage*—i.e., assuming during training that there is access to data that does not exist at serving time—an error typically discovered after the model was deployed and several incorrect live predictions were made (Lg1, Md1, Md5, Lg5). Interviewees felt that anticipating all possible forms of data leakage during experimentation was tedious; thus, sometimes leakage was retroactively checked during code review in an evaluation stage (Lg1. There were other types of bugs that were discussed by multiple participants, such as accidentally flipping labels in classification models (Lg1, Sm1, Lg3, Md3) and forgetting to set random seeds in distributed training when initializing workers in parallel (Lg1, Lg4, Sm5). However, the vast majority of bugs described to us in the interviews were seemingly bespoke and not shared among participants. For example, Sm3 forgot to drop special characters (e.g., apostrophes) for their language models. Lg3 found that the imputation value for missing features was once corrupted. Lg5 mentioned that a feature of unstructured data type (e.g., JSON) had half of the keys' values missing for a "long time."

When asked how they detect these one-off bugs, interviewees mentioned that their bugs showed similar symptoms of failure. One symptom was a large discrepancy between offline validation accuracy and production accuracy immediately after deployment (Lg1, Lg3, Md4, Lg5). However, if there were no ground-truth labels available immediately after deployment (as discussed in Section 2.4), interviewees had to resort to other strategies. For example, some inspected the results of data quality checks (Section 2.4). Lg1 discussed their struggle to debug without "ground-truth:":

> Um, yeah, it's really hard. Basically there's no surefire strategy. The closest that I've seen is for people to integrate a very high degree of observability into every part of their pipeline. It starts with having really good raw data, observability, and visualization tools. The ability to query. I've noticed, you know, so much of this [ad-hoc bug exploration] is just—if you make the friction [to debug] lower, people will do it more. So as an organization, you need to make the friction very low for investigating what the data actually looks like, [such as] looking at specific examples.

To diagnose bugs, interviewees typically sliced and diced data for different groups of customers or data points (Md1, Lg3, Md3, Md4, Md6, Lg6). Slicing and dicing is known to be useful for identifying bias in models [165, 68], but we observed that our interviewees used this technique beyond debugging bias and fairness; they sliced and diced to determine common failure modes and data points similar to these failures. Md4 discussed annotating bugs and only drilling down into their queue of bugs when they observed "systematic mistakes for a large number of customers."

Interviewees mentioned that after several iterations of chasing bespoke ML-related bugs in production, they had developed a sense of paranoia while evaluating models offline—possibly as a coping mechanism (Lg1, Md1, Lg3, Md5, Md6, Lg6). Lg1 said:

> ML [bugs] don't get caught by tests or production systems and just silently cause errors. This is why [you] need to be paranoid when you're writing ML code, and then be paranoid when you're coding.

Lg1 then recounted a bug that was "impossible to discover" after a deployment to production: the code for a change that added new data augmentation to the training procedure had two variables flipped, and this bug was miraculously caught during code review even though the training accuracy was high. Lg1 claimed that there was "no mechanism by which [they] would have found this besides someone just curiously reading the code." Since ML bugs don't cause systems to go down, sometimes the only way to find them is to be cautious when inspecting code, data, and models.

2.5 Discussion

Our findings suggest that automated production ML pipelines are enabled by MLEs working through a continuous loop of i) data preparation, ii) experimentation, iii) evaluation & deployment, and iv) monitoring and response (Figure 2.1). Although engineers leverage different tools to help with technical aspects of their workflow, such as experiment tracking and data validation [211, 20], patterns began to emerge when we studied how MLE practices varied across company sizes and experience levels. We discuss these patterns as "the three Vs of MLOps" (Section 2.5), and follow our discussion with implications for both production ML tooling (Section 2.5), and opportunities for future work (Section 2.5).

Velocity, Visibility, Versioning: Three Vs of MLOps

Findings from our work and prior work suggest three broad themes of successful MLOps practices: Velocity, Visibility, and Versioning. These themes have synergies and tensions across each stage of MLEs' workflow, as we discuss next.

Velocity

Since ML is so experimental in nature, it's important to be able to prototype and iterate on ideas quickly (e.g., go from a new idea to a trained model in a day). Interviewees attributed their productivity to development environments that prioritized high experimentation velocity and debugging environments that allowed them to test hypotheses quickly. Prior work has extensively documented the Agile tendencies of MLEs, describing how they iterate quickly (i.e. with *velocity*) to explore a large ML or data science search space [6, 98, 66, 145, 213]. Amershi et al. (2019) describe how experimentation can be sped up when labels are annotated faster (i.e., rapid data preparation) [6]. Garcia et al. (2021) explore tooling to help MLEs correct logging oversights from too much velocity in experimentation [47], and Paleyes et al. (2022) mention the need to diagnose production bugs quickly to prevent future similar issues from occurring [140]. First, our study re-enforces the view the MLEs are agile workers who value fast results. P1 said that people who achieve the best outcomes from experimentation are people with "scary high experimentation velocity." Similarly, the multi-stage deployment strategy can be viewed as an optimistic or high-velocity solution to deployment: deploy first, and validate gradually across stages. Moreover, our study provides deeper insight into how practitioners rapidly debug deployments—we identify and describe practices such as on-call rotations, human-interpretable filters on model behavior, data quality alerts, and model rollbacks.

At the same time, high velocity can lead to trouble if left unchecked. Sambasivan et al. (2021) observed that, for high-stakes customers, practitioners iterated too quickly, causing ML systems to fail—practitioners "moved fast, hacked model performance (through hyperparameters rather than data quality), and did not appear to be equipped to recognise upstream and downstream people issues" [165] Our study exposed strategies that practitioners used to prevent themselves from iterating too quickly: for example, in Section 2.4, we described how some applications (e.g., autonomous vehicles) require separate teams to manage evaluation, making sure that bad models don't get promoted from development to production. Moreover, when measuring ML metrics outside of accuracy, e.g., fairness [68] or product metrics (Section 2.4), it is challenging to make sure all metrics improve for each change to the ML pipeline [140]. Understanding which metrics to prioritize requires domain and business expertise [86], which could hinder velocity.

Visibility

In organizations, since many stakeholders and teams are impacted by ML-powered applications and services, it is important for MLEs to have an end-to-end view of ML pipelines. P1 explicitly mentioned integrating "very high degree of observability into every part of [the] pipeline" (Section 2.4). Prior work describes the importance of visibility: for example, telemetry data from ML pipelines (e.g., logs and traces) can help engineers know if the pipeline is broken [86], model explainability methods can establish customers' trust in ML predictions [140, 200, 87], and dashboards on ML pipeline health can help align nontechnical stakeholders with engineers [81, 91]. In our view, the popularity of Jupyter notebooks among MLEs, including among the participants in our study, can be explained by Jupyter's gains in velocity and visibility for ML experimentation, as it effectively combines REPL (Read-Eval-Print-Loop)-style interaction and visualization capabilities despite its *versioning* shortcomings. Our findings corroborate these prior findings and provide further insight on how visibility is achieved in practice. For example, engineers proactively monitor feedback delays (Section 2.4). They also document live failures frequently to keep validation datasets current (Section 2.4), and they engage in on-call rotations to investigate data quality alerts (Section 2.4).

Visibility also helps with velocity. If engineers can quickly identify the source of a bug, they can fix it faster. Or, if other stakeholders, such as product managers or business analysts, can understand how an experiment or multi-staged deployment is progressing, they can better use their domain knowledge to assess models according to product metrics (see Section 2.4), and intervene sooner if there's evidence of a problem. One of the pain points we observed was that end-to-end experimentation—from the conception of an idea to improve ML performance to validation of the idea—took too long. The uncertainty of project success stems from the unpredictable, real-world nature of experiments.

Versioning

Amershi et al. (2019) mention that "fast-paced model iteration" requires careful versioning of data and code [6]. Other work identifies a need to also manage model versions [193, 211]. Our work suggests that mananging *all* artifacts—data, code, models, data quality metrics, filters, rules—in tandem is extremely challenging but vital to the success of ML deployments. Prior work explains how these artifacts can be queried during debugging [19, 26, 169], and our findings additionally show that versioning is particularly useful when *teams* of people work on ML pipelines. For instance, during monitoring, on-call engineers may receive a flood of false-positive alerts; looking at old alerts might help them understand whether a specific type of alert actually requires action. In another example, during experimentation, ML engineers often work on models and pipelines they didn't initially create. Versioning increases visibility: engineers can inspect old versions of experiments to understand ideas that may or may not have worked.

Not only does versioning aid visibility, but it also enables workflows to maintain high velocity. In Section 2.4, we explained how pipeline jungles are created by quickly responding to ML bugs by constructing various filters and rules. If engineers had to fix the training dataset or model for every bug, they would not be able to iterate quickly. Maintaining different versions for different types of inputs (e.g., rules to auto-reject incomplete data or different models for different users) also enables high velocity. However, there is also a tension between velocity and versioning: in Section 2.4, we discussed how parallelizing experiment ideas produces many versions, and ML engineers could not cognitively keep track of them. In other words, having high velocity can mean drowning in a sea of versions.

Opportunities for ML Tooling

Our main takeaway is that production ML tooling needs to aid *humans* in their workflows, not just automate technical practices (e.g., generating a feature or training a model). Tools should help improve at least one of the three Vs, and there are opportunities for tools in each stage of the workflow. We discuss each in turn.

Data Preparation

As mentioned in Section 2.4, separate teams of data engineers typically manage pipelines to ingest, clean, and preprocess data on a schedule. While existing tools automate scheduling these activities, there are unadressed ML needs around retraining and labeling. Prior work and our interviews indicate that ML engineers retrain models on some arbitrary cadence [140, 89], without understanding the effect of the cadence on the quality of predictions. Models might be stale if they are retrained only monthly, or they might retrain using invalid or corrupt data if they are retrained faster than the data is validated and cleaned (e.g., hourly). Moreover, the optimal retraining cadence depends on the data, ML task, and amount of organizational resources, such as compute, training time, and number of engineers on the team. New tools can help with these challenges and determine the best retraining cadence for ML pipelines. With respect to labeling, existing tools help with either labeling at scale [153] or labeling with high quality [90], but it is hard to achieve both. As a result, organizations have custom infrastructure and teams to resolve label mismatches, apply domain knowledge, and reject incorrect labels. Labeling tools can leverage ensembling and add postprocessing filters to reject and resolve incorrect and inconsistent labels. Moreover, they should track feedback delays and surface this information to users.

Experimentation

As discussed in Section 2.4, experiments are typically done in development environments and then promoted to production clusters during deployment. The mismatch between the two (or more!) environments can cause bugs, creating an opportunity for new tools. The development cluster should maximize iteration speed (velocity), while the production cluster should minimize end-user prediction latency [31]. Hardware and software can be different in each cluster, e.g., GPUs for training vs. CPUs for inference, and Python vs. C++, which makes this problem challenging. New tools are prioritizing reproducibility—turning Jupyter notebooks into production scripts [178], for instance—but should also standardize how engineers interact with experimentation workflows. For example, while experiment tracking tools can literally keep track of thousands of experiments, how can engineers sort through all these versions and actually understand what the best experiments are doing? Our findings and prior work show that the experimental nature of ML and data science leads to undocumented tribal knowledge within organizations [85, 81]. Documentation solutions for deployed models and datasets have been proposed [51, 124], but we see an opportunity for tools to help document *experiments*—particularly, failed ones. Forcing engineers to write down institutional knowledge about what ideas work or don't work slows them down, and automated documentation assistance would be quite useful.

Evaluation and Deployment

Prior work has identified several opportunities in the evaluation and deployment space. For example, there is a need to map ML metric gains to product or business gains [89, 115, 86]. Additionally, tools could help define and calculate subpopulation-specific performance metrics [68]. From our study, we have observed a need for tooling around the multi-staged deployment process. With multiple stages, the turnaround time from experiment idea to having a full production deployment (i.e., deployed to all users) can take several months. Invalidating ideas in earlier stages of deployment can increase overall, end-to-end velocity. Our interviewees discussed how some feature ideas no longer make sense after a few months, given the nature of how user behaviors change, which would cause an initially good idea to never fully and finally deploy to production. Additionally, an organization's key product metrics—e.g., revenue or number of clicks—might change in the middle of a multi-stage deployment, killing the deployment. This negatively impacts the engineers responsible for the deployment. We see this as an opportunity for new tools to streamline ML deployments in this multi-stage pattern, to minimize wasted work and help practitioners predict the end-to-end gains for their ideas.

Monitoring and Response

Recent work in ML observability identifies a need for tools to give end-to-end visibility on ML pipeline behavior and debug ML issues faster [175, 19]. Basic data quality statistics, such as missing data and type or schema checks, fail to capture anomalies in the values of data [126, 20, 146]. Our interviewees complained that existing tools that attempt to flag anomalies in the values of data points produce too many false positives (Section 2.4). An excessive number of false-positive alerts, i.e., data points flagged as invalid even if they are valid, leads to two pain points: (1) unnecessarily maintaining many model versions or simple heuristics for invalid data points, which can be hard to keep track of, and (2) a lower overall accuracy or ML metric, as baseline models might not serve high-quality predictions for these invalid points. Moreover, due to feedback delays, it may not be possible to track ML performance (e.g., accuracy) in real time. What metrics can be reliably monitored in real time, and what criteria should trigger alerts to maximize precision and recall when identifying model performance drops? How can these metrics and alerting criteria automatically tune themselves over time, as the underlying data changes? We envision this to be an opportunity for new data management tools.

Moreover, as discussed in Section 2.4, when engineers quickly respond to production bugs, they create pipeline jungles. Such jungles typically consist of several versions of models, rules, and filters. Most of the ML pipelines that our interviewees discussed were pipeline jungles. This combination of modern model-driven ML and old-fashioned rule-based AI indicates a need for managing filters (and versions of filters) in addition to managing learned models. The engineers we interviewed managed these artifacts themselves.

Limitations and Future Work

Since we wanted to find common themes in production ML workflows across different applications and organizations, our interview study's scope was quite broad: we set out on a quest to discover shared patterns, rather than to predict or explain. We asked practitioners open-ended questions about their MLOps workflows and challenges, but did not probe them about questions of fairness, risk, and data governance: these questions could be studied in future interviews. Moreover, we did not focus on the *differences* between practitioners' workflows based on their company sizes, educational backgrounds, or industries. While there are interview studies for specific applications of ML [148, 18, 44], we see further opportunities to study the effect of organizational focus and maturity on the production ML workflow. There are also questions for which interview studies are a poor fit. Given our findings on the importance of collaborative and social dimensions of MLOps, we would like to explore these ideas further through participant action research or contextual inquiry.

Moreover, this chapter focuses on a *human-centered* workflow surrounding production ML pipelines. Focusing on the *automated* workflows in ML pipelines—for example, continuous integration and continuous deployment (CI/CD)—could prove a fruitful research direction. Finally, we only interviewed ML engineers, not other stakeholders, such as software engineers or product managers. Kreuzberger et al. (2022) present a diagram of technical components of the ML pipeline (e.g., feature engineering, model training) and interactions between ML engineers and other stakeholders [89]. Another interview study could observe these interactions and provide further insight into practitioners' workflows.

2.6 Conclusion

In this chapter, we presented results from a semi-structured interview study of 18 ML engineers spanning different organizations and applications to understand their workflow, best practices, and challenges. Engineers reported several strategies to sustain and improve the performance of production ML pipelines, and we identified four stages of their MLOps workflow: i) data preparation, ii) experimentation, iii) evaluation and deployment, and iv) monitoring and response. Throughout these stages, we found that successful MLOps practices center around having good velocity, visibility, and versioning. Finally, we discussed opportunities for tool development and research.

Chapter 3

Context: The Missing Piece in the Machine Learning Lifecycle

Machine learning (ML) models have become ubiquitous in modern applications. The ML lifecycle describes a three-phase process used by data scientists and data engineers to develop, train, and serve models. Unfortunately, context around the data, code, people, and systems involved in these pipelines is not captured today. In this chapter[1], we first discuss common pitfalls that missing context creates. Some examples where context is missing include tracking the relationships between code and data and capturing experimental processes over time. We then discuss techniques to address these challenges and briefly mention future work around designing and implementing systems in this space.

3.1 Introduction

Most modern applications—ranging from personal voice assistants to manufacturing services—rely on machine learning in some form. These applications rely on machine learning *models* to render predictions in response to a query.

The development, training, and serving of machine learning models is the result of a process that we call the *Machine Learning Lifecycle*. This lifecycle has three phases (Figure 3.1): pipeline development, training, and inference. The pipeline development phase is an iterative process in which data scientists transform and visualize data, explore various model designs, and experiment with many features. Note that the focus on model design often leads to the term "model development." However, the true product of *pipeline development* is a reusable pipeline that describes how to construct a model from a given dataset. This pipeline is then executed on a much larger, near-real time dataset to generate a production-ready model, and these trained models are in turn used to serve predictions for new inputs to the application.

The ML lifecycle is data-intensive and spans many individuals. Each stage is often managed by a different team, with different incentives and different structures. The transitions

[1]Chapter published in KDD Workshop on Common Model Infrastructure (CMI) [46].

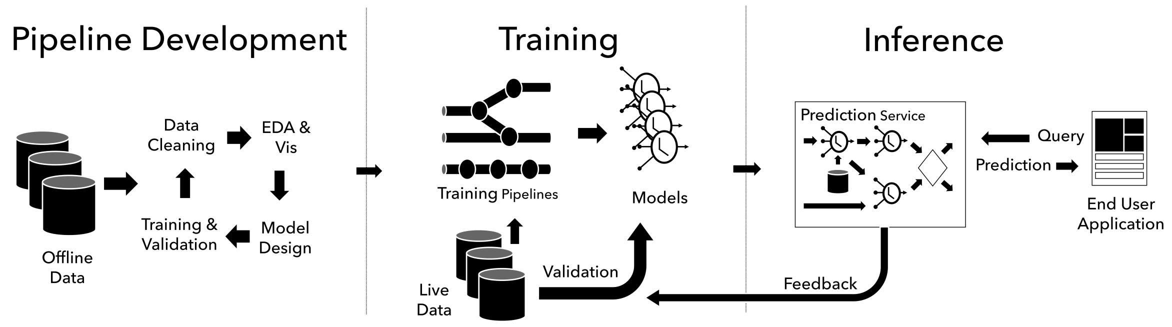


Figure 3.1: The end-to-end machine learning lifecycle.

between stages and teams are usually *ad-hoc* and unstructured. As a result, in serious machine learning deployments today, no one person or system has an end-to-end view of the ML lifecycle. This is problematic for a variety of reasons, including irreproducibility of experimental results, complicated debugging, and a lack of accountability. We believe there is a key missing component required to capture this view: the *context* that surrounds the ML lifecycle.

Recent work highlights data context [62] as a critical aspect of any data-centric effort, including ML pipelines. That work defines data context as "all the information surrounding the use of data in an organization." It goes on to distinguish three key types of data context: application context, behavioral context, and change over time (the "ABCs" of context). The application context (A) captures semantic and statistical models that explain how bits should be interpreted. Behavioral context (B) extends the traditional notion of data provenance to capture how both people and software interact with data. Lastly, change over time (C) captures how each of the other two types of data context are evolving.

In the next section, we highlight common pitfalls that arise from ignoring data context in the ML lifecycle. We then discuss approaches to contextualize the ML lifecycle and briefly mention future work.

3.2 The Absence of Context

In this section, we describe how code and data should be interpreted, how they evolve over time, and what their relationship is to each other and the people that create and use them. Our discussion also includes the history of an organization's model management practices. We consider more than just the success stories: we consider the context surrounding the experiments that failed because of bugs, poor performance, excessive cost, and so on. We strive to leverage this context to learn from our mistakes and the mistakes of others, to reduce work duplication, and to formalize machine learning practices.

The Code and Data Ecosystem

Data is a first class citizen in any machine learning pipeline. The same pipeline trained on different data can yield a drastically different result. Unfortunately, most organizations today do not capture the relationship between code and data, both within and across the phases of the ML lifecycle. Most importantly, context around which data set was used to train a particular version of a model is lost. Other basic information—like schemas, distributions, and expectations—are even less available.

In the rest of this section, we describe a few common scenarios demonstrating the loss of context in the code and data ecosystem.

This data looks wrong! The first step in the ML lifecycle often consists of transforming raw data into a cleaned dataset. That dataset is often shared and reused. If a data scientist or analyst who receives the data encounter issues, they need access to the original data and transformation scripts. If this context is not explicitly logged, the derivation of the cleaned dataset is opaque to the receiving user. What's worse is that they may be unaware that the data they received was derived through a transformation process. **Missing Context:** the code and data used to construct the dataset.

If I could only find that model from last week! Pipeline development is inherently experimental with many iterations of trial and error. There are a variety of reasons that we may want to return to earlier versions of our models and data. For example, it is common to reach a dead-end in model design only to return to an earlier model. Recreating an earlier state requires reverting not only code but also data, parameters, and configurations. Finding the earlier best version may require searching through many alternative versions. **Missing Context:** the versions of code, data, parameters, and configurations over time.

It worked better yesterday! Models inevitably degrade in their predictive power. There are many reasons for this degradation. For example, a shift in the data distribution can result in a rapid decline in predictive power. Diagnosing this decline requires comparing training data with live data. Solving the problem may require retraining the model, revisiting earlier design decisions, or even rearchitecting the underlying model. **Missing Context:** the full lineage of the model through each stage of the ML lifecycle as it existed at the time of training.

I fixed it, who needs to know? Models are routinely composed in production. For example, modeling a users likelihood to buy a product might depend on predictions about the user's political preference. Changes to upstream model will impact the quality of predictions from a downstream model. Surprisingly, improving the accuracy of an upstream model can in some cases degrade the accuracy of downstream models because the downstream models were trained to compensate for errors. **Missing Context:** the eventual consumers of predictions from models.

Learning From Your Mistakes

An analyst's first question when debugging might be to ask if they are encountering a well-known problem. Answering this requires context in two scopes. First, it requires context around past experiments for this pipeline—both successful and unsuccessful. It could also potentially require surrounding other, related projects in the organization. We next look at two examples of this sort, one diagnosing a problem and the other remediating a problem.

I've seen this problem before. Any organization will develop a set of recurring problems in their ML lifecycle. Imagine a model for a ride-sharing service is predicting negative trip durations. A natural question might be to ask why the training data would lead to this prediction and whether there's a standard data cleaning step that is missing. An expert on trip data might know that canceled trips are logged as having -1 duration and that these trips should be dropped; however, this information is not readily available without experience. **Missing context:** Standard transformations applied to a dataset.

I tried that already! Imagine a data scientist is tasked with improving the performance of a certain predictor developed by a *different* team member. It is likely that the same set of common model designs has already been considered. Knowing this information can help avoid redundant work. **Missing context:** Past designs and their resulting test scores.

Proper Methodology

The statistical nature of the ML Lifecycle requires experimental discipline. To some degree, this discipline can be captured and checked by recording the behavior of analysts and the structure of the pipelines themselves. While no single approach will solve all of the problems surrounding statistical and methodological practices, context will play a crucial role.

Have I used this test data before? When developing models, it's important to separate training and testing datasets. Overuse of testing data during training can lead to poor generalization and performance. For example, tuning a parameter by repeatedly testing on the same data can lead to over-fitting. **Missing context:** Behavior of the analyst and content & structure of the pipeline.

3.3 System Requirements for Model Lifecycle Management

Software engineering systems exist for building complex software projects, managing versions, automatically testing and deploying built binaries, logging, monitoring, and so on. More importantly, software engineering systems may have close analogs for the ML Lifecycle, and could serve as a sound starting point for research. However, there are characteristics of the ML Lifecycle that are unique to model management, and as a result, we do not expect to find strong parallels from software engineering. The characteristics that are special to the ML Lifecycle are the following: unlike the software engineering lifecycle, the ML Lifecycle is *empirical*, *combinatorial*, and *data driven.* The ML Lifecycle is *empirical* and *combinatorial* because even the most experienced and meticulous pipeline engineers will have only a vague idea of the elements and structure of the final pipeline. Context will play a central role in navigating the vast space of possibilities: it will be necessary to understand which data artifacts were used to train which models, and with what configurations. The ML Lifecycle is also *data-driven* because the model, the output of training, is inextricably linked to the data it was trained on. Contrast this with a software project, where the build or compilation of such systems are data independent. Context will be indispensable for the data and sample management systems that are used throughout the phases of the ML Lifecycle. Finally, as [168] notes, the use of machine learning models in an application often results in substantial technical costs stemming from the failure of traditional abstractions and software design principles in the presence of machine learning. While decades of research in software engineering has established techniques and tools to manage the development and deployment of complex software applications, there has been very little research into managing the development and deployment of machine learning applications.

Pipeline Tools

For *pipeline development*, we must build tools with which we can easily change the elements and structure of pipelines and feel confident in undoing unsuccessful attempts. Tools that support and enable *pipeline development* are “composition tools” [158]. These tools must support hypothesis formation, evaluation of alternatives, interpretation of intermediary results, and dissemination of results. Experienced pipeline engineers tasked with designing an ML pipeline often have a vague idea of the elements and structure of the pipeline in advance, but the particular configuration and final architecture must be discovered. For example, the pipeline engineer may know to use a neural network rather than a Naive Bayes Classifier, but the final number of layers and neurons per layer are unknown at the onset. To the extent that *pipeline development* is a creative and empirical endeavor, pipeline engineers must be encouraged to explore the space of possible alternatives. Tools that encourage exploration must have “low viscosity”, meaning that they should easily enable changes to all aspects of the pipeline [158]; additionally, these tools must have very good *undo* capabilities, so the pipeline engineer feels comfortable trying new things, and powerful yet efficient previewing mechanisms to limit the consumption of scarce and valuable resources (such as computing time, memory, and data) without impeding exploration.

There are many possible roles for context in *pipeline development* tools. As a more general form of meta-data, context can help us interpret, and therefore compare the artifacts within pipelines and across their versions. Consider how a version control system, such as Git, would benefit from context. Git, which is tailored for source control and the software engineering lifecycle, defines change semantics as line-by-line differences. But these semantics are meaningless for binary and data artifacts. To detect meaningful change in data, we should look at metadata properties including the schema, distributions across different attributes, or topics in the data, instead of the exact contents of the records. When comparing binary objects such as two different models, it will be much more useful to compare the metadata of these models – e.g., their accuracy, recall, training hyper-parameter configurations, and so on – rather than just *diffing* their binaries.

Training Systems

Frameworks such as TensorFlow manage many of the problems of distributed training at scale; namely, scheduling computation to run on different devices, and managing the communication between these devices [1]. However, data engineers are still responsible for managing very large datasets and their versions over time, provisioning resources, and controlling for variability inherent in training in the cloud (attributable to multi-tenancy, hardware, workload, and data variability, and so on). In many cases, as deployed models interact with the world they produce new data that can be used to retrain existing models. In these cases, automated systems will periodically re-train existing models in response to changes in data or code. Unlike *pipeline development*, *training* does not require any human design considerations, and the search space is exhaustively enumerable in principle. However, data and model management requirements increase substantially. Of special significance to *training* will be techniques to automatically decide when to train, leverage knowledge of multiple pipelines to improve training performance, and study mechanisms to mitigate the risk of over-fitting.

Now, we consider how training systems may leverage and benefit from context. Context may benefit training in two ways: mitigating the risk of over-fitting and surfacing opportunities for optimization. Context about the training data and the processes that generated it, as well as context about how and how often that data is used can help reduce the risk of over-fitting to the data. As for optimization opportunities, organizations or cloud service providers will simultaneously run many training jobs often. In this case, it would be extremely valuable if the pipeline training system could characterize the pipelines that it is running and compare them to find common or equivalent transformations or sub-graphs. Today, this kind of context is not widely available to the distributed systems that train models. This means that the system cannot intelligently schedule similar pipelines to run together, and re-use or share intermediate artifacts. The failure to understand the resource needs (meta-data) of each action in the pipeline can also lead to lost opportunities to schedule the execution of dis-similar pipelines more optimally. In summary, missed application context about the pipelines results in re-computation and poor schedules.

Inference Systems

During *inference*, trained models are used to render predictions for new inputs. The primary systems challenge of inference is delivering low latency, highly available predictions under heavy and often bursty query load. However, an often overlooked but critical component of inference is managing model versions and tracking variation in queries and prediction errors.

Understanding, how models are performing production and debugging their failures depends critically on capturing their provenance.

Inference is the ML Lifecycle phase that already takes the most advantage of context. Prediction serving systems monitor the end-user application for feedback to measure the quality of the predictions. Other ways in which context could help inference is by leveraging context describing the input and output interfaces of models, together with the metadata about the training data and intended use for the model. If this context is available, the prediction serving system can more intelligently compose or combine models or their predictions to improve robustness and decrease latency.

3.4 Related Work

Data Context Services. Modern data analytics ecosystems can benefit from a rich notion of *data context*, which captures all the information surrounding the use of data in organization. Data context includes traditional notions of metadata combined with data provenance and version histories. Recent work has discussed systems, such as Ground [62] and ProvDB [121], that enable users to capture richer data context. As discussed earlier, our work here builds on data context. However, this chapter's key contribution is a domain-specific discussion of the benefits of data context in one domain. In other words, the ideas discussed here consistute an "Aboveground" application discussed in [62].

Model Management & Serving. A variety of recent work [32, 2, 33] has focused on the *inference* stage of the end-to-end ML lifecycle shown in Figure 3.1. Data context is essential for prediction serving systems that are fundamentally disconnected from the pipeline development and training stages of the ML lifecycle. For instance, explaining change in prediction accuracy is challenging without access to full lineage of the models, training data, and any hyperparameters. Similarly, ModelDB [195] captures context in the *pipeline development* phase but is disconnected from the broader ML lifecycle.

3.5 Conclusion

In this chapter, we characterized the ML Lifecycle and postulated that the crucial missing piece within and across every phase is *context*: "all the information surrounding the use of data in an organization." The transitions between stages and teams are usually *ad-hoc* and unstructured, meaning no single individual or system will have a global, end-to-end view of the ML Lifecycle. This can lead to problems like irreproducibility, over-fitting, and missed opportunities for improved productivity, performance, and robustness.

Some of the scenarios that illustrate the problems of missing context include those where the code and data used to clean data are lost, past versions are irretrievable, and deployed models degrade in performance over time. We also consider organizational context, often termed "tribal knowledge", that can be leveraged to reduce work duplication and respond to common problems, and some of the behavioral context that is generated during a pipeline engineer's activities, and how this context can be used to characterize and potentially help them improve their process.

The ML Lifecycle has some similarities with the software engineering lifecycle, such as how engineers describe pipelines for building a binary artifact, how there is a need to maintain different versions over time, and so on. This observation motivates our recommendation of drawing inspiration and guidance from software engineering when designing the tools for ML Lifecycle. However, we also note that the ML Lifecycle has some important differences from software engineering: namely, the ML Lifecycle is *empirical*, *combinatorial*, and *data-driven*. We also argue that context will be the key component in supplementing existing tools and creating future ones for the service of the ML Lifecycle.

Finally, we would like to end by noting that we are actively developing a system called Flor[2], a context-first tool for managing the ML Lifecycle. Our initial focus has been on tooling the *pipeline development* process, but we hope that these techniques will be applicable in the broader scope of the ML Lifecycle.

[2]`github.com/ucbrise/flor`

Chapter 4

Application Context: Hindsight Logging for Model Training

In modern Machine Learning, model training is an iterative, experimental process that can consume enormous computation resources and developer time. To aid in that process, experienced model developers log and visualize program variables during training runs. Exhaustive logging of all variables is infeasible, so developers are left to choose between slowing down training via extensive *conservative* logging, or letting training run fast via minimalist *optimistic* logging that may omit key information. As a compromise, optimistic logging can be accompanied by program checkpoints; this allows developers to add log statements post-hoc, and "replay" desired log statements from checkpoint—a process we refer to as *hindsight* logging. Unfortunately, hindsight logging raises tricky problems in data management and software engineering. Done poorly, hindsight logging can waste resources and generate technical debt embodied in multiple variants of training code. In this chapter[1] , we present methodologies for efficient and effective logging practices for model training, with a focus on techniques for hindsight logging. Our goal is for experienced model developers to learn and adopt these practices. To make this easier, we provide an open-source suite of tools for Fast Low-Overhead Recovery (FLOR) that embodies our design across three tasks: (i) efficient background logging in Python, (ii) adaptive periodic checkpointing, and (iii) an instrumentation library that codifies hindsight logging for efficient and automatic record-replay of model-training. Model developers can use each `flor` tool separately as they see fit, or they can use `flor` in hands-free mode, entrusting it to instrument their code end-to-end for efficient record-replay. Our solutions leverage techniques from physiological transaction logs and recovery in database systems. Evaluations on modern ML benchmarks demonstrate that `flor` can produce fast checkpointing with small user-specifiable overheads (e.g. 7%), and still provide hindsight log replay times orders of magnitude faster than restarting training from scratch.

[1]Chapter published as a VLDB conference paper [48].

4.1 Introduction

Due to the growing scale and complexity of sophisticated models [39, 154, 188], exploratory model development increasingly poses data management problems [182]. At every step of exploration, model developers routinely track and visualize time series data to diagnose common training problems such as exploding/vanishing gradients [65], dead ReLUs [112], and reward hacking [7]. Model developers use state-of-the-art loggers specialized to machine learning (e.g. TensorBoard [53], and WandB [17]) to efficiently trace and visualize data as it changes over time. The following are common examples of times series data logged in model training:

- **Training Metrics**: The loss, accuracy, learning rate, and other metrics as they change over time.
- **Tensor Histograms**: Histograms of weights, gradients, activations, and other tensors as they change over time.
- **Images & Overlays**: Segmentation masks, bounding boxes, embeddings, and other transformed images as they change over time.

In our experience, all model developers log some training metrics by default. Whether their logging practice is conservative or optimistic depends on whether they log additional training data by default. Next, we illustrate the relevant differences between conservative and optimistic logging, with the aid of three fictitious characters: Mike for methodical conservative logging, Chuck for ad-hoc optimistic logging, and Judy for methodical optimistic logging.

Conservative Logging

Conservative logging is eager and characterized by stable expectations about what data (and how much of it) will be necessary for analysis [210]. It is especially well-suited to later stages of the machine learning lifecycle, where models are periodically trained for many hours on fresh data [185], and refinements of the model training pipeline are usually light and limited to some tuning [46].

Mike records everything (mnemonic for microphone)

Mike is a model developer for a major tech company. His organization's policy is that model developers should log training metrics, tensor histograms, and some images and overlays by default. Although his logging practices can add substantial overhead to training (black bar in Figure 4.1), his jobs usually run as offline batches, and his productivity is not blocked on the results of training. Moreover, when he receives an alert from the training or monitoring system, the execution data he needs for post-hoc analysis will be ready.

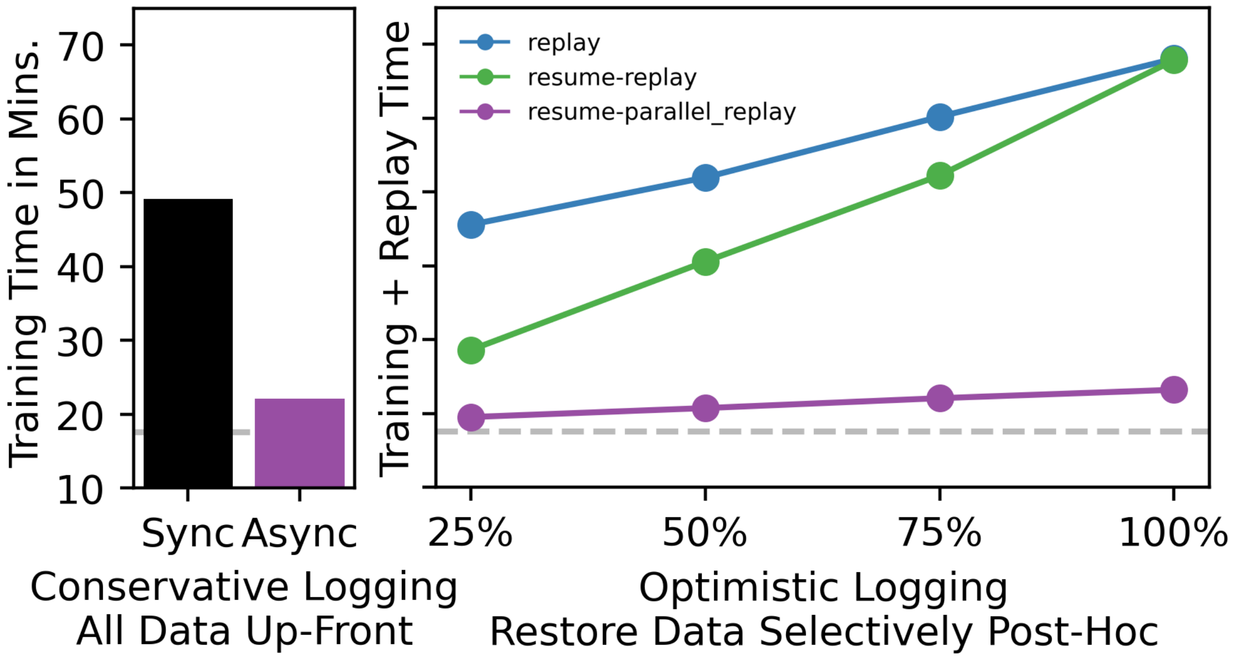


Figure 4.1: Conservative v. Optimistic logging performance at 100 epochs of Squeezenet on CIFAR-100 [76]. All workloads log tensor histograms for the activations, weights, and gradients 4× per epoch. The gray horizontal line corresponds to the same training job but without any logging. Both, the purple bar and line, correspond to `flor` logging.

Although reducing logging overhead is not a high priority for Mike, he is not the only developer in his organization using high-end GPU clusters. At scale, even minor improvements to training efficiency will translate into measurable benefits for the organization. Later in this chapter, we will present a tool for **low-overhead materialization in the background (Section 4.3)**. Our tool enables Mike to continue to log data at the same rate and with his logger of choice (e.g. tensorboardx), at a fraction of the original overhead (purple bar in Figure 4.1).

Optimistic Logging

In contrast to conservative logging, optimistic logging is an agile and lazy practice especially well-suited to early and unstructured stages of exploratory model development. In optimistic logging, model developers log training metrics such as the loss and accuracy by default, and defer collection of additional data until analysis time, when they may restore it selectively. Execution data is restored by adding logging statements to training post-hoc, and replaying—possibly from checkpoint. We refer to this practice as *hindsight logging*. Optimistic logging consists of (i) logging some training metrics by default, and (ii) selectively restoring additional training data post-hoc with hindsight logging. Model developers gain agility in exploration from optimistic logging in three ways:

- **Deferred Overhead**: Each training batch executes and produces results as quickly as possible. Faster evaluations means more trials in the same time span.
- **Flexible Cost Schedule**: Model developers can selectively restore *just* the data they need post-hoc. The fewer epochs they need to probe; the fewer resources they burn.
- **Separation of Concerns**: Concerns about what data to log and how much of it do not burden the developer during design and tuning—these are postponed until analysis time.

In the fast path of the common case, model developers get all the relevant information from the training loss, and move on. In exceptional cases, however, training replay may be necessary for post-hoc data restoration. Compare how Chuck and Judy would restore data.

Chuck doesn't record anything (mnemonic for toss)

Chuck is a first year graduate student in Astronomy who is knowledgeable about the latest developments in machine learning, but ill-versed in software engineering practices. Chuck logs the training loss and accuracy by default, but does not save checkpoints during training.

Judy uses good judgment (mnemonic for judge)

Judy is an experienced model developer with a strong software engineering background. Like Chuck, she only logs the training loss and accuracy by default; unlike Chuck, she also checkpoints model training periodically, and manages the many versions of her code and data.

When either Chuck or Judy need to restore training data post-hoc—say, the tensor histograms for the gradients at a rate of $4\times$ per epoch—they will selectively add the necessary logging statements to their training code, and re-train. In many cases, Chuck and Judy will only want to restore data for a small fraction of training (e.g. 25% of the epochs), near the region where the loss exhibits an anomaly (e.g. near the middle of training). Because Judy checkpointed her training execution, she is able to resume training at an arbitrary epoch. Chuck, on the other hand, must retrain from the start. In the right pane of Figure 4.1, we plot training plus replay times for Chuck (blue line) and Judy (green line).

In this chapter, we will concretely define the methodology that enables Judy to achieve effective record-replay of model training (Section 4.2). Our goal is for experienced model developers to learn and adopt these best practices, for their numerous benefits. One surprising consequence of Judy's approach is that she can **parallelize replay of model training** with her periodic checkpoints (Section 4.2). The purple line in Figure 4.1 represents Judy's parallel replay. Additionally, we evaluate (Section 5.6) and open-source our Fast Low-Overhead Recovery suite (abbreviated as FLOR) for hindsight logging—with the following set of tools:

- An optimized materialization library for low-overhead logging and checkpointing in Python (Section 4.3).
- An adaptive periodic checkpointing mechanism, so record overhead never exceeds a specifiable limit (Section 4.3).
- An instrumentation library that can transparently transform Python training code to conform with the methodical hindsight logging approach (Section 4.3). This protects model developers from incurring technical debt as a consequence of lapses in discipline, and enables novices to restore time series data as efficiently as experts.

4.2 Methodical Hindsight Logging

In hindsight logging, model developers can choose what to log long after model training: at analysis time and with a question in mind. In essence, we want to query past execution state, without versioning that state in full. We draw inspiration from the rich body of work in databases dedicated to fast recovery [127, 201, 216]. Although that work focuses mostly on transactions, the lessons and trade-offs transfer naturally to execution recovery for arbitrary programs. There are two means for recovering execution data: physically, by reading it from disk; and logically, by recomputing it. Both a purely physical approach and a purely logical approach are unattractive in our setting, due to prohibitive overhead on record and prohibitive latency on replay, respectively. Instead, hindsight logging—like transaction logging—embraces a hybrid "physiological" [57] approach that takes partial checkpoints on the first pass (henceforth the *record* phase), and uses those checkpoints to speedup redo (henceforth the *replay* phase). In this section, we give a high-level overview of the enabling methodology behind efficient hindsight logging:

1. First and foremost, **checkpoint periodically** during training. At least once per epoch for partial replay, but much less frequently is sufficient for parallel replay.
2. Additionally, **enclose long-running code inside a conditional statement to exploit memoization** speedups. On record, the execution materializes the side-effects of each memoized block. On replay, model developers will run their code from the start without modification, and the execution will intelligently skip the recomputation of some blocks by loading their side-effects from disk.
3. Finally, include logic to **resume training from a checkpoint**. Replay of model training is embarrassingly parallel given periodic checkpoints. To parallelize replay of model training, a model developer dispatches multiple training jobs in parallel, each loading checkpoints to resume training from a different epoch and terminating early.

If at any point through our forthcoming discussion the programming burden seems too high, the reader should note that we also provide a tool that codifies and automatically applies

```
1  init_globals()
2  checkpoint_resume(args, (net, optimizer))
3  for epoch in range(args.start, args.stop):
4    if skipblock.step_into(...):
5      for batch in training_data:
6        predictions = net(batch.X)
7        avg_loss = loss(predictions, batch.y)
8        avg_loss.backward()
9        optimizer.step()
10   skipblock.end(net, optimizer)
11   evaluate(net, test_data)
```

Figure 4.2: Training script prepared for methodical hindsight logging: checkpoint resume (line 2), block memoization (lines 4 - 10), and periodic checkpointing (line 10). The semantics of skipblock are covered in subsection 4.2.

these methods for the benefit of the user: an instrumentation library that features a hands-free mode for convenience and robustness (Section 4.3).

Periodic Checkpointing & Memoization

Many model developers already checkpoint training periodically. This is traditionally done for warm-starting training as well as for fault tolerance. In this section, we show how to exploit further benefits from periodic checkpointing, without incurring additional overheads. In Figure 4.2, we provide an example of how a model developer would materialize the model and optimizer state once per epoch (line 10). This state serves a dual purpose. First, it comprises the relevant side-effects of the preceding code block (lines 4-9), so it serves a memoization purpose (computation skipping). Second, it captures all of the state that is modified every epoch of training, so it comprises a valid and complete checkpoint. This dual purpose of selective state capture is a fortunate coincidence that arises naturally from the nested loops structure of model training.

We make use of the skipblock language construct [25], to denote block memoization. The first two requirements for efficient record-replay are periodic checkpointing and block memoization. Both are achievable by the following functionality, which is encapsulated by the skipblock for ease of exposition:

- **Parameterized Branching**: skipblock always applies the side-effects of the enclosed block to the program state, but does so in one of two ways: (a) by executing the enclosed block, or (b) by skipping the block and instead loading the memoized side-effects from its corresponding checkpoint. skipblock automatically determines whether to execute or skip the enclosed block. It is parameterized by relevant execution state:

i.e. record-execution, replay-resume, replay-execution, and whether the enclosed block is probed.

- **Side-Effect Memoization** (i.e. Periodic Checkpointing): When the enclosed block is executed, skipblock materializes its side-effects (the arguments passed to the call in line 10, Figure 4.2). It is possible to optimize the skipblock for low-overhead background materialization (Section 4.3), and adaptive periodic materialization (Section 4.3), but these optimizations do not alter the semantics of skipblock.

- **Side-Effect Restoration**: Whenever the enclosed block is skipped, skipblock restores its side-effects from its corresponding checkpoint (line 10, Figure 4.2). skipblock is able to efficiently locate an execution's corresponding checkpoint on disk, and apply its side-effects to the program state.

A block may not be skipped on replay when the model developer adds a hindsight logging statement inside the block. Although skipblock memoizes the block's final state (i.e. state that is visible to subsequent program statements), it does not materialize intermediate state, such as the activations of the model (e.g. line 6 in Figure 4.2). Consequently, if the model developer wishes to restore the model activations post-hoc, it will not be possible to skip the nested training loop. To restore data in such cases, parallel replay is the only option for reducing the latency of hindsight logging.

Parallel Replay by Checkpoint Resume

As we saw in the previous subsection, our approach cannot avoid expensive recomputation when intermediate training state, such as the gradients or activations are logged post-hoc. In such cases, model developers will want to reduce replay latency by utilizing additional resources—specifically, more GPUs for parallelism. Although auto-parallelizing arbitrary sequential code remains an open challenge [12], the replay of checkpointed model training is a special case: training replay is embarrassingly parallel given periodic checkpoints. As we multi-purposed periodic checkpointing in the previous section for memoization, so too we now multi-purpose checkpoint resume—a current staple in the training code of many model developers—for parallel replay. Parallel replay enables us to substantially cut hindsight logging latency, and due to the prevalence of checkpoint resume, this is possible without incurring a programming burden. To parallelize replay, a model developer simultaneously resumes training from various checkpoints:

1. First, they dispatch multiple training jobs in parallel.

2. Then, each job loads the checkpoint (line 2 in Figure 4.2) that corresponds to the epoch it is resuming from. For example, to resume training at epoch 25, the job loads the checkpoint stored at the end of epoch 24.

```
 1 init_globals()
 2 for epoch in range(0, args.stop):
 3   if skipblock.step_into(...
 4       && epoch >= args.start):
 5     for batch in training_data:
 6       predictions = net(batch.X)
 7       avg_loss = loss(predictions, batch.y)
 8       avg_loss.backward()
 9       optimizer.step()
10   skipblock.end(net, optimizer)
11   lr_scheduler.step()
12   evaluate(net, test_data)
```

Figure 4.3: Training script prepared for methodical hindsight logging. Training can resume from a partial checkpoint (no `lr_scheduler` in checkpoint).

3. Finally, each job independently works on its share of work (see the `range` in line 3 of Figure 4.2).

Pseudoresuming from partial checkpoints

When model developers write code for periodic checkpointing themselves, they can ensure that the objects they capture constitute a complete checkpoint. However, when model developers entrust flor to instrument their code for automatic periodic checkpointing, flor will not be able to automatically determine whether the checkpoint is complete or partial with respect to training. As we will discuss in Section 4.3, flor can only estimate the side-effects of blocks of code enclosed by skipblocks: a restriction we use to render our static analysis tractable. flor will not estimate the side-effects of the program at arbitrary points, and it will not check whether the data materialized constitutes a complete (or partial) checkpoint with respect to training, since doing so statically (i.e. with low overhead) would be intractable in Python [67, 180, 136].

Consequently, flor assumes checkpoints materialized automatically are partial with respect to training. By partial, we mean that there are objects modified every epoch that are not stored by the checkpoints (e.g. the `lr_scheduler` in Figure 4.3). As a result, it is not possible to resume training from an arbitrary epoch merely by loading a partial checkpoint (i.e. a physical recovery approach). Instead, we start training from the beginning (line 2 in Figure 4.3), and use the partial checkpoints to skip recomputation of memoized blocks during the initialization — or *pseudoresume* — phase (lines 3-4 in Figure 4.3). This approach is characteristically physiological because it relies on a combination of recomputation and disk reads for recovery. Although *pseudoresume* is especially important for auto-parallelizing replay of model training, we share this method here because novice model developer may

accidentally store partial checkpoints. This technique allows them to resume training efficiently all the same. For illustration, suppose, that the model developer wants to resume training from epoch 25, using the script in Figure 4.3. The skipblock would be initialized to a *pseudoresume* state, and then toggle to an execution state.
Pseudoresume phase for epochs in the range 0-24 (inclusive):

1. Skip the nested training loop (lines 3-9 in Figure 4.3).
2. Load the side-effects of the skipped block (line 10 in Figure 4.3).
3. All other statements execute normally.

Execution phase from epoch 25 onward:

1. Step into the nested training loop (lines 3-9 in Figure 4.3).
2. All other statements execute normally.

In summary, memoization can resume model training from an arbitrary epoch, even in the absence of complete checkpoints. As we will show in the evaluation, the overhead of *pseudoresume* is amortized in parallel replay, so that the difference between checkpoint resume and *pseudoresume* is imperceptible to the end-user. This result is important because it enables us to efficiently auto-parallelize the replay of model training, even with partial checkpoints.

4.3 Tooling for Hindsight Logging

Model developers may adopt the methods described in Section 4.2 to achieve efficient hindsight logging. To facilitate this adoption, we provide a suite of *Fast Low-Overhead Recovery* tools—flor for short—as aid to the developer. Model developers may use each tool separately as they see fit, or they may use flor in hands-free mode, entrusting it to instrument their code end-to-end for efficient record-replay. flor provides the following tools:

- An optimized materialization library for low-overhead logging and checkpointing (Section 4.3).
- An adaptive periodic checkpointing mechanism, so record overhead never exceeds a specifiable limit (Section 4.3).
- An instrumentation library that can transparently transform training code to conform to the methodical hindsight logging approach (Section 4.3). This protects model developers from incurring technical debt as a consequence of lapses in discipline, and enables novices to restore time series data as efficiently as experts.

Background Logging

flor provides a background materialization mechanism optimized for PyTorch, which is compatible with model developer's machine learning logging service of choice (for example, TensorBoard [53], MLFlow [212], and WandB [17]). Background logging is used natively by skipblocks for low-overhead periodic checkpointing (Section 4.2). It is also available separately as a library for end-users.

Both logging and checkpointing can add measurable overhead to training because they require moving data from GPU memory to DRAM, serializing it into byte arrays, and then writing those arrays to disk. Of the latter two, serialization is typically much more expensive than I/O: by an average factor of 4.3× according to our microbenchmarks [108]. Consequently, after copying select data from GPU memory to DRAM (so it is protected from overwrites), we would like to take materialization (both serialization and I/O) off the main thread—which is dedicated to model training—and do it in the background. Despite its maturity and widespread popularity, Python makes this very difficult.

The Python interpreter has a notorious Global Interpreter Lock that prevents parallelism among Python threads. Unfortunately, the Python IPC schemes also require serialization by the sending process—returning us to our original problem. To avoid serialization we could use a solution like Apache Plasma, but it only avoids serialization for a subset of Python data types (notably dataframes and arrays) and actually cannot serialize other data types including Pytorch tensors. We eventually found a workaround at the operating system level, using `fork()` as a mechanism to achieve efficient one-shot, one-way IPC between a parent and child process, with copy-on-write concurrency. To materialize a record checkpoint, the main process forks and then immediately resumes model training; the child process serializes the checkpoint, writes it to disk, and then terminates. To prevent too many calls to `fork()`, we buffer up checkpoints and process them in batches of 5000 objects. Given the short lifespan of these child processes and an infrequent rate of forking due to batching, we have never seen more than two live children at any point in our evaluations—including in models that ran for many hours (Section 5.6).

In a technical report [108], we provide a more detailed discussion of the design and performance of our background materialization mechanism. This mechanism cuts logging overheads by 73.5% on average, according to our microbenchmarks [108]. Execution speedups due to background logging are modest for workloads whose logging overheads are dominated by periodic checkpointing ($\mu = 4.76\%$ overhead down to $\mu = 1.74\%$). This is because periodic checkpointing is already light and doesn't add much overhead to training. However, as we saw in Figure 4.1, background logging can have a drastic effect when used for conservative logging (180% overhead down to 26%), since logging overheads account for a much larger fraction of end-to-end training times in those cases.

Table 4.1: Symbol table for Adaptive Periodic Checkpointing

Symbol	Description
M_i	time to materialize side-effects of block identified by i
R_i	time to restore side-effects of block identified by i
C_i	time to compute block identified by i
n_i	count of executions (so far) for block i
k_i	count of checkpoints (so far) for block i
G	degree of replay parallelism
c	constant scaling factor
ϵ	tunable parameter denoting overhead tolerance

Adaptive Periodic Checkpointing

In this section, we present a decision rule for dynamically calculating an appropriate checkpointing *period* or frequency. This condition is automatically tested by skipblocks to adapt the frequency of checkpointing to each training workload. For many developers, checkpointing once per epoch is a good default, but in general, the right checkpointing frequency depends on the training workload: e.g. how fast or slow the code executes relative to the size of its checkpoints. The goal of adaptive periodic checkpointing is to automatically materialize checkpoints as frequently as will increase expected replay speedups, subject to the constraint that record overhead does not exceed a user-specifiable limit. Next we derive the invariants we use for adaptive periodic checkpointing. We refer the reader to the notation in Table 4.1.

The Record Overhead Invariant

We require that the materialization overhead of a block is at most a small fraction of its computation time: $M_i < \epsilon C_i$. This simplistic invariant is enough to ensure that record never exceeds a user-specifiable overhead (ϵ), but it is all-or-nothing: a block is memoized always or never. Since blocks are often nested inside loops, and model developers may parallelize replay even with a small number of checkpoints (e.g. 2 checkpoints: 3× parallelism), we need to relax our invariant to account for periodic checkpointing. Specifically, due to the nested loops structure of model training, we introduce n_i and k_i:

$$k_i M_i < n_i \epsilon C_i \quad \Rightarrow \quad \frac{M_i}{C_i} < \frac{n_i \epsilon}{k_i} \tag{4.1}$$

The Replay Latency Invariant

To avoid regret, record-replay should always be faster than two vanilla executions (with neither overhead nor speedups). Even for hindsight logging workloads that do not permit partial replay, the speedups from parallel replay alone should more-than-offset the overhead incurred on record. Accounting for record overhead, we can assess each block i for this condition as follows:

$$M_i + R_i + \left(\frac{n_i}{G} - 1\right) C_i < n_i C_i \tag{4.2}$$

The -1 in Equation 4.2 accounts for the fact that each parallel worker resumes from a stored checkpoint and does not need to compute its first iteration. Because G is determined on replay and is not known during record, we satisfy the Replay Latency Invariant by testing Equation 4.3 instead. Equation 4.3 guarantees the Replay Latency Invariant as long as there is some parallelism ($G \geq 2$); we omit the details for brevity.

$$\begin{gathered} M_i + R_i < \frac{n_i}{k_i} C_i \quad \text{and} \quad R_i = cM_i \\ \Rightarrow \frac{M_i}{C_i} < \frac{n_i}{k_i(1+c)} \end{gathered} \tag{4.3}$$

Because the time to restore is not known at record time, we assume that it is proportional to the time to materialize. Our naive assumption is $c = 1.0$, and this estimate is refined after observing materialization and restoration times from record-replay. In our case, the average scaling factor over all measured workloads (Table 5.1) turned out to be $c = 1.38$.

The Joint Invariant

The Joint Invariant is automatically checked by skipblocks at record time for adapting the frequency of checkpointing. Blocks are tested after executing, but before materialization. By restricting memoization to blocks that pass the Joint Invariant test, `flor` simultaneously satisfies the Record Overhead and Replay Latency invariants. This follows from the fact that the Joint Invariant is derived algebraically from the two invariants.

$$\begin{gathered} \frac{M_i}{C_i} < \frac{n_i}{k_i + 1} \min\left(\frac{1}{1+c}, \epsilon\right) \\ c = 1.38, \epsilon = 0.0667 \end{gathered} \tag{4.4}$$

Note the $k_i + 1$ in Equation 4.4: this accounts for the fact that the test is performed after the execution of the block but before the materialization of its checkpoint. The goal is for the invariant to continue to hold if the checkpoint is materialized. We derive the Joint Invariant, Equation 4.4, from Equation 4.1 and Equation 4.3. Both invariants are satisfied when the computed ratio, M_i/C_i, is less than the minimum of both thresholds.

Table 4.2: Set of rules for static side-effect analysis. At most one rule is activated by each program statement. The rules are sorted in descending order of precedence.

Rule	Pattern	ΔChangeset
0	$v_1, ..., v_n = u_1, ..., u_m \wedge \exists v_i \in$ `Changeset`	`No Estimate`
1	$v_1, ..., v_n = obj.method(arg_1, ..., arg_m)$	$\{obj, v_1, ..., v_n\}$
2	$v_1, ..., v_n = func(arg_1, ..., arg_m)$	$\{v_1, ..., v_n\}$
3	$v_1, ..., v_n = u_1, ..., u_m$	$\{v_1, ..., v_n\}$
4	$obj.method(arg_1, ..., arg_m)$	$\{obj\}$
5	$func(arg_1, ..., arg_m)$	`No Estimate`

Instrumentation for Hands-Free Mode

As desired by the user, flor can instrument their model training code for automatic and efficient record-replay. The principal objectives of flor instrumentation are twofold:

1. Memoization and periodic checkpointing by nesting loops inside a skipblock, and statically estimating their side-effects.
2. Auto-parallelization of training replay by a syntax-directed transformation of loop iterators, enabling *pseudoresume* from partial checkpoints (Subsection 4.2).

Autorecording Model Training

The first goal of instrumentation is to efficiently and correctly memoize loop executions for the model developer—without their intervention. flor memoizes loops because, in machine learning, they correspond to long-running code, and unlike arbitrary "long-running code", loops can be detected statically. Ensuring correct and efficient memoization requires (i) capturing all of the loop's side-effects, and (ii) avoiding the capture of too many redundancies. Unfortunately, due to the language's dynamic features and extensive reliance on (compiled) C extensions, an exact and efficient side-effect analysis in Python is intractable [67, 180, 136]. Past work overcomes Python's analysis limitations by restricting the expressiveness of the language [8, 97], making some assumptions (e.g. that the variables don't change types [13]), or relying on user source annotations [198]. In a similar vein, we achieve efficient side-effect analysis by assuming that loop bodies in model training are predominantly written in PyTorch [144]. To the extent that loops deviate from our assumption, our static analysis will be unsafe (i.e. may misdetect side-effects), so we will automatically perform deferred correctness checks after replay and report any anomalies to the programmer. We find that our assumption holds frequently enough to be useful for hindsight logging purposes. Model developers do not typically build models or write training algorithms from scratch. Instead, they rely on popular machine learning frameworks such as PyTorch. Like many 3rd-party libraries, PyTorch has a well-defined interface by which it modifies the user's program in limited ways [149]. The effects of PyTorch on the user's program are limited to (i) assignments and (ii) encapsulated state updates from method calls. As a result, all the side-effects of PyTorch code can be detected statically, with two notable exceptions: when an optimizer modifies a model, and when a learning rate scheduler modifies an optimizer [149].

First, flor estimates a set of changes ("changeset") for each block using the six rules in Table 4.2. flor walks the abstract syntax tree statement by statement, testing which rule is activated by each statement. The changeset for a block accumulates the individual changes of its member statements. Rules have a precedence such that at most one is activated per statement. Statements that activate no rule are ignored. Next, flor performs a filtering step on the changeset to remove variables that are scoped to the body of the loop. flor removes from the changeset any variable that is defined in the body of the loop (henceforth "loop-scoped variable"), under the assumption that this variable is local to the loop and is not read after the end of the loop. Finally, we make use of our encoded library-specific knowledge to augment the changeset at runtime (this is the only step that is not done statically). For PyTorch, it suffices to encode two facts [149]: (i) the model may be updated via the optimizer; and (ii) the optimizer may be updated via the learning rate schedule. flor augments the changeset to include side-effects which were not detected by the rules, but which can be inferred from other elements in the changeset.

Autoparallelizing Replay

The second goal of instrumentation is to autoparallelize replay of model training, assuming partial checkpoints exist. Replay instrumentation consists of wrapping the main loop's iterator inside a `flor.generator` (line 10 in Figure 4.4), to model the *pseudoresume* behavior we covered earlier (in Subsection 4.2 and Figure 4.3). Generators define an iterator by a series of `yield` statements, and allow us to control global program state between iterations of the main loop; namely, toggle the skipblocks from a *skip* state to a *step-into* state between epochs (lines 3, 6 in Figure 4.4). We implement parallel replay by having every parallel worker (`NPARTS` in total) execute the same instrumented code (as in Figure 4.4), and flor sets `PID` to a different value for each worker so they work on distinct segments of the main loop.

Deferred Checks for Correctness

As we have discussed, Python's dynamic features and extensive reliance on (compiled) C extensions, make an exact and efficient side-effect analysis intractable. flor's approach to detecting side-effects is efficient but unsafe: it may misdetect side-effects and thus fail to checkpoint sufficient state for correct replay. To mitigate risk, we automatically check that common user-observable state between record and replay matches [5]. The standard training metrics that get logged by default (e.g. the loss and accuracy) form a fairly unique fingerprint of a model's training characteristics, so it's hard to perturb state or data that the model

```python
def flor.generator(*args):
  pseudoresume_sgmnt, work_sgmnt = partition(*args)
  skipblock.set_state('replay', 'skip')
  for element in pseudoresume_sgmnt:
    yield element
  skipblock.set_state('replay', 'step_into')
  for element in work_sgmnt:
    yield element

for epoch in flor.generator(range(N), PID, NPARTS):
  ...
```

Figure 4.4: flor instrumentation nests the main loop's iterator inside a generator to parallelize replay (with *pseudoresume*). A generator defines an iterator, and enables us to control global state between iterations of the main loop.

Table 4.3: Computer vision and NLP benchmarks used in our evaluation.

Name	Benchmark	Task	Model	Dataset	Train/Tune	Epochs
RTE	GLUE	Recognizing Textual Entailment	RoBERTa	RTE	Fine-Tune	200
CoLA	GLUE	Language Acceptability	RoBERTa	CoLA	Fine-Tune	80
Cifr	Classic CV	Image Classification	Squeezenet	Cifar100	Train	200
RsNt	Classic CV	Image Classification	ResNet-152	Cifar100	Train	200
Wiki	GLUE	Language Modeling	RoBERTa	Wiki	Train	12
Jasp	MLPerf	Speech Recognition	Jasper	LibriSpeech	Train	4
ImgN	Classic CV	Image Classification	Squeezenet	ImageNet	Train	8
RnnT	MLPerf	Language Translation	RNN w/ Attention	WMT16	Train	8

depends on without this being reflected in one of the model's metrics. Consequently, at the end of replay, we run `diff`, and warn the user if the replay logs differ from the record logs in any way other than the statements added for hindsight logging.

4.4 Evaluation

To assess flor's ability to meet the goals of Section 4.1 in practice, we evaluated eight diverse machine learning workloads, taken from three separate benchmarks: classic computer vision, the General Language Understanding and Evaluation (GLUE) [199], and ML Perf [120] (Table 5.1). These workloads vary in their tasks, model architectures, execution time scales, and software engineering patterns; they are jointly representative of a large class of model training workloads. Every experiment was run on P3.8xLarge EC2 instances with 4 Tesla V100 GPUs, 64 GB of GPU memory in aggregate, 32 vCPUs, 244 GB of RAM, and an EBS bandwidth (IO throughput) of 7Gbps. The checkpoints generated by flor record were spooled from EBS to an S3 bucket by a background process.

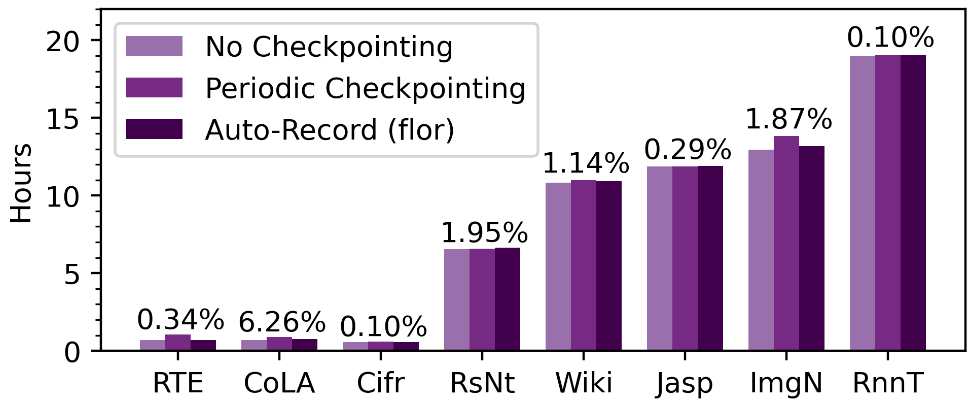


Figure 4.5: Comparison of model training times, with and without checkpointing, in hours. "Periodic Checkpointing" measures the time achievable when a model developer judiciously selects the contents of a checkpoint, at a frequency of once per epoch. The overhead added by flor Record is denoted by the text labels over each group of bars.

Flor Record Overhead is Low

We compared the overhead added by manual periodic checkpointing, at a rate of once per epoch, against the overhead added by automatic flor record (Figure 4.5). We did not manually set the period for any of the flor record experiments. The checkpointing period was automatically calibrated by the mechanism in Section 4.3. **`flor` record does not add significant overhead to training**, so it may be enabled by default. Moreover, the `flor` **instrumentation library achieves a comparable outcome as hand-tuned periodic checkpointing**, with competitive performance, and without intervention from the user. For manual periodic checkpointing, we assume that each checkpoint is complete with respect to training. For flor record we only assume partial checkpoints: each checkpoint corresponds to the side-effects of the code block it memoizes (Section 4.2), but it may be incomplete with respect to training.

Flor Record Overhead is Adaptive

Different model developers have different sensitivies to overhead. In this section, we measured that record is able to adjust its checkpointing frequency to stay within the user-specifiable overhead limits (e.g. $\epsilon = 6.67\%$). The nested training loops in most model training workloads are memoized every epoch by 's adaptive checkpointing mechanism. This is because the time to materialize their checkpoints is negligible compared to the time it takes to execute them. In contrast, the sharp drop in overhead for fine-tuning workloads is due to their less frequent checkpointing (Figure 4.6). Fine-tuning workloads are checkpointed less frequently because their loops have poor materialization time to computation time ratios: their checkpoints are

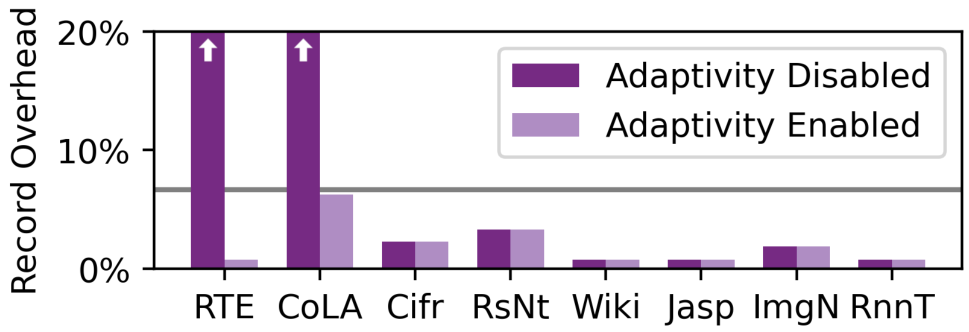


Figure 4.6: Impact of adaptivity on record overhead. The two upward arrows denote extreme values: adaptivity-disabled overhead is 91% for RTE and 28% for CoLA. The user-specifiable overhead tolerance (6.67%) is denoted by the gray horizontal line. No workload exceeds the overhead limit with adaptive checkpointing.

massive relative to their short execution times. This is the case because the vast majority of weights are frozen in model fine-tuning, so a loop execution quickly updates a small fraction of values in an enormous model [71]. We find that **adaptive checkpointing drastically reduces overhead** on model fine-tuning workloads (RTE & CoLA), and **ensures that no workload exceeds the user's overhead tolerance**.

Flor Replay Latency is Low

In this section, we measure the replay speedups achieved by replay, assuming record checkpoints were materialized during training. Consequently, we measure the replay speedups when instruments model developers' code end-to-end for efficient hindsight logging—without intervention from the developer.

Replay latencies are query dependent: they depend on the position of hindsight logging statements in the code. In cases when the model developer probes only the outer loop of training (as in line 13 of Figure 4.8), **partial replay can provide latencies on the order of minutes, even when model training takes many hours to execute**. This is achieved by skipping unnecessary recomputation with loop memoization (e.g. skipping the nested training loop). The top subplot in Figure 4.7 shows outer-loop probe latencies for each of our models. Note the improvements range from 7× to 1123×—with the more significant improvements favoring the longer experiments (recall Figure 4.5). When the model developer logs data post-hoc from the inner training loop (as in line 10 in Figure 4.8), then that loop must be re-executed on replay, and it will not contribute to savings from loop memoization. For these workloads, we will need to rely on parallelism to reduce latencies. We measured the hindsight logging latencies when a full re-execution of model training was necessary by running replay on multiple machines—this is shown in the bottom subplot in

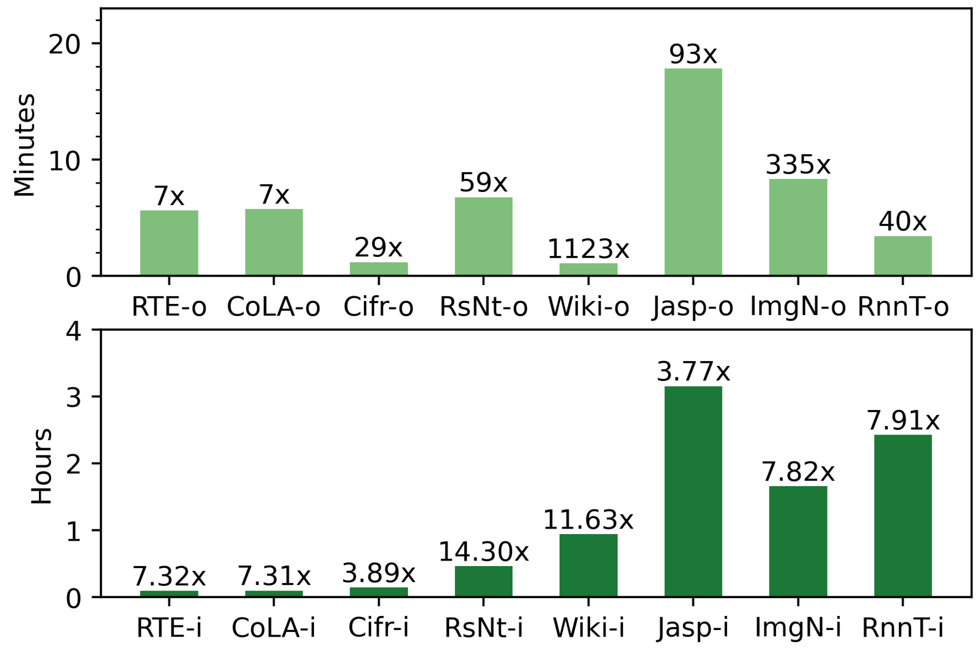


Figure 4.7: Replay latency, factored by the position of hindsight logging statements. The top plot reports partial and parallel replay speedups when the model developer probes only the outer main loop (as in line 13 of Figure 4.8). The bottom plot reports parallel-only replay speedups when the model developer probes the inner training loop and a full re-execution is needed (as in line 10 of Figure 4.8). Each workload uses as many machines, from a pool of four machines, as will result in parallelism gains. Text labels show speedup factors relative to naive re-execution.

Figure 4.7. Assuming no work or guidance from the model developer, beyond the insertion of a couple hindsight logging statements, **automatically parallelizes and conditionally skips computation on the re-execution of model training** (Subsection 4.3).

The parallel replay workloads used as many machines from the pool of 4 machines as would provide further parallelism gains. Each machine has 4 GPUs. In the limit, every epoch may re-execute in parallel, but the degree of parallelism may be increased even further by checkpointing additional state, which we leave as future work.

Ideal Parallelism and Scale-out

Next, we compare the performance of parallel replay with checkpoint resume against parallel replay with checkpoint pseudoresume (refer to Subsection 4.2). An expert model developer, such as Judy, who does periodic checkpointing by-hand can ensure that the checkpoints are complete with respect to training. Thus, they can achieve the *checkpoint resume* performance

```
1  init_globals()
2  checkpoint_resume(args, (net, optimizer))
3  for epoch in range(args.start, args.stop):
4    if skipblock.step_into(...):
5      for batch in training_data:
6        predictions = net(batch.X)
7        avg_loss = loss(predictions, batch.y)
8        avg_loss.backward()
9        optimizer.step()
10       tensorboard.add_histogram(net.params())
11   skipblock.end(net, optimizer)
12   evaluate(net, test_data)
13   tensorboard.add_overlays(net, test_data)
```

Figure 4.8: Model training example with checkpoint resume. Lines 10 and 13 correspond to hindsight logging statements, or logging statements added after training.

```
1  init_globals()
2  for epoch in range(0, args.stop):
3    if skipblock.step_into(...
4        && epoch >= args.start):
5      for batch in training_data:
6        predictions = net(batch.X)
7        avg_loss = loss(predictions, batch.y)
8        avg_loss.backward()
9        optimizer.step()
10       tensorboard.add_histogram(net.params())
11   skipblock.end(net, optimizer)
12   lr_scheduler.step()
13   evaluate(net, test_data)
14   tensorboard.add_overlays(net, test_data)
```

Figure 4.9: Model training example with checkpoint pseudoresume. Lines 10 and 14 correspond to hindsight logging statements (added after training).

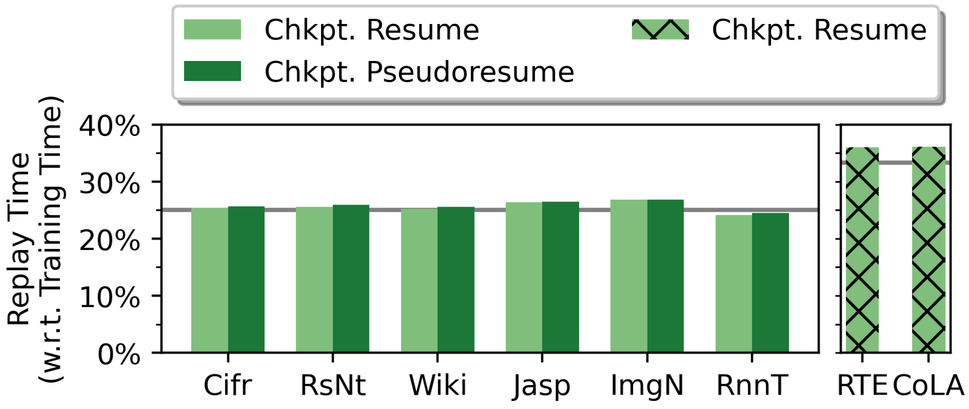


Figure 4.10: Parallel replay time of model training jobs (4x parallelism), as fraction of a serial re-execution. RTE & CoLA only have 6 work partitions each, so parallelism on 4 GPUs leads to at best 2/6 = 33% replay time.

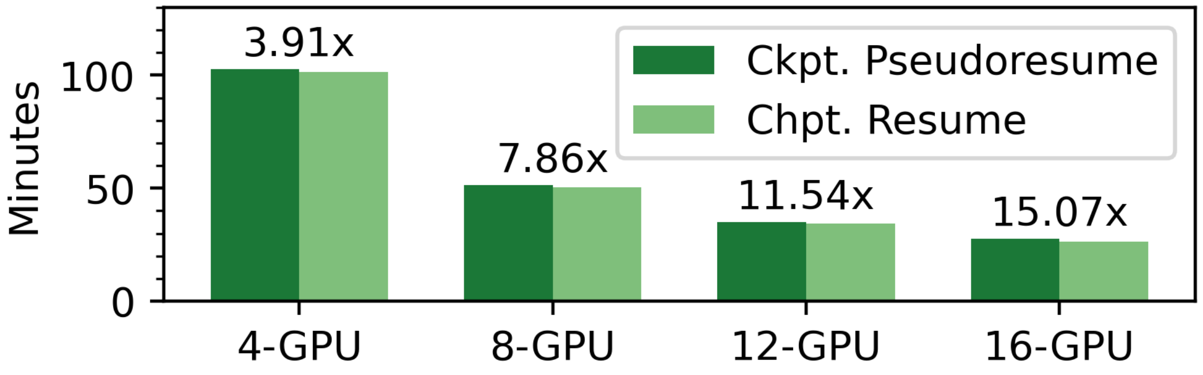


Figure 4.11: Replay time using GPUs from multiple P3.8xLarge machines, on experiment RsNt. The "checkpoint resume" speedup relative to a sequential execution is denoted by the text labels.

in Figures 4.10 and 5.11. On the other hand, when flor instruments training code on behalf of the developer, it will rely on *checkpoint pseudoresume*, because as we discussed earlier, flor cannot automatically ensure that its checkpoints are complete with respect to training, and it assumes that the checkpoints are partial. Our results show that, although pseudoresuming training adds initialization overhead, this overhead is amortized through the course of parallel replay, such that **there is a negligible difference between checkpoint resume and checkpoint pseudoresume**.

In Figure 4.10, we measured parallel replay performance, and observe that **flor replay achieves near-ideal parallelism**. Ideal parallelism is denoted by the gray horizontal line in each subplot (Figure 4.10). These results are possible because model training *replay* is embarrassingly parallel given (complete or partial) checkpoints.

Because parallel workers do not need to communicate or coordinate, flor replay is especially well-suited for elastic and horizontally scalable cloud computing, in which **it can scale out to more GPUs at low marginal costs**. To assess our scaling performance, in Figure 5.11 we illustrate the incremental speedup as we add 4-GPU machines. We choose RsNt as our experiment because it has 200 epochs to parallelize. The modest gap between our results and ideal here is due to load balancing limitations: balancing 200 epochs over 16 parallel workers results in each worker doing up to 13 epochs of work. Consequently, the maximum achievable speedup on 16 GPUs is $\frac{200}{13}$: 15.38×.

4.5 Related Work

ML lifecycle management

The machine learning lifecycle encompasses many tasks, including model design and development, training, validation, deployment, inference, and monitoring [46]. There is a wide range of research and tooling being developed to support these many tasks. ML lifecycle management is especially challenging because it involves many cycles of trial-and-error [99], and its dependencies are hard to scope [168]. When something goes wrong, ML engineers may need to rollback their model to an earlier version [123, 196], inspect old versions of the training data [72, 79, 117], or audit the code that was used for training [122, 167]. Those activities require the proper management, versioning, and provenance tracking of data, models, code, and other context; existing solutions provide some support [16, 62, 101, 105, 212]. Hindsight logging is a novel contribution in the lifecycle, and its minimalist, low-friction interface makes it complementary to the prior work. flor is designed to be compatible with any of the tools in the Python ecosystem. In terms of training libraries, we have focused on PyTorch, but adopting another training library involves only encoding any side-effects in the library's API (Section 4.3).

Model Debugging

There are many tools and techniques for helping users understand the behavior of their models [55, 113, 159, 160, 170], and for inspecting model internals [109, 155, 192, 194, 137]. These techniques only inspect models, so they are complementary to our work—which focuses on the execution data generated while training the models. The value of execution data is evidenced by widespread use of domain-specific loggers and visualization tools for that data, including TensorBoard [53], MLflow Tracking [212], and WandB [17]. Hindsight logging allows developers to keep their current logging practices and tools, and use them to "query the past".

Partial Materialization

Inspired by classical work on materialized views [27], a new body of work addresses partial materialization of state in ML workflows, to aid in iterative tasks like debugging. As representative examples, Columbus [214] accelerates the exploration of feature selection by choosing to cache feature columns; Helix [207] focuses on choosing to cache and reuse the outputs of black-box workflow steps; Mistique [194] focuses on techniques for compressing model-related state and deciding whether to materialize or recompute. These systems introduce bespoke languages for pre-declaring what to capture prior to computation; they also provide custom query APIs to interrogate the results. Hindsight logging is complementary: it enables post-hoc materialization in cases when it was *not* prespecified. Precisely because flor does not dictate a new API, it is compatible with this prior work: users of these systems (or any library with pre-declared annotations) can benefit from flor to add annotations in hindsight, and benefit from flor's efficient replay to add materialized state. At a more mechanistic level, some of the policies and mechanisms from this work (e.g., the model compression of Mistique) could be adapted into hindsight logging context to further improve upon our results.

Recovery and Replay Systems

Our techniques are inspired by literature on both database recovery and program replay. Hindsight logging is a redo-only workload, and we use a "physiological" approach [57]: in our view, a model training script is a complete logical log (in the WAL sense) of a model training execution, and occasional physical checkpoints serve solely to speed up redo processing. Parallel and selective redo recovery was studied as early as ARIES [127, 201]. Parallelism in those techniques is data-partitioned and recovers the most recent consistent state; we are in essence time-partitioned and recover all prior states. In that sense our work bears a resemblance to multiversion storage schemes from POSTGRES [183] onward to more recent efforts (e.g., [110, 134]). These systems focus on storing complete physical versions, which is infeasible in our setting due to constraints on runtime overhead.

Numerous program record-replay systems have been used in the past for less data-oriented problems. Jalangi is a system for dynamic program analysis that automatically records the required state during normal processing, and enables high-fidelity selective replay [171]. This is achieved by identifying and storing memory loads that may not be available at replay time, using a "shadow memory" technique. Unlike flor, Jalangi replay has strict correctness guarantees. flor uses side-effect analysis rather than shadow memory because the former is lighter on overhead: in this sense, we risk replay anomalies to reduce record overhead and replay latency.

Prior work on Output Deterministic Replay [5] makes a similar trade-off as we do. However that work pays for higher latencies to enable reproduction of nondeterministic bugs; we can avoid that overhead in Python model-training scenarios because sources of nondeterminism may be captured, and model-training frameworks are increasingly designed for reproducibility. An interesting line of work enables reverse replay with relatively high fidelity and without overhead by using memory dumps on a crash [35]—this impressive result is made possible by the spatial locality of bugs in the vicinity of execution crashes; one complication with model debugging is that training errors, such as over-fitting, may not crash the program. We borrow the skipblock language construct from Chasins and Bodik's record-replay system for web scraping [25].

4.6 Conclusion

At every step of exploration, model developers routinely track and visualize time series data to assess learning. Most model developers log training metrics such as the loss and accuracy by default, but there soon arise important differences between what additional training data model developers log—with major implications for data management. In contrast to conservative logging, optimistic logging is an agile and lazy practice especially well-suited to early and unstructured stages of exploratory model development. In optimistic logging, model developers log training metrics such as the loss and accuracy by default, and defer collection of more expensive data until analysis time, when they may restore it selectively with hindsight logging. In the common case, or fast path, model developers get all the relevant information from the training loss, and move on. In exceptional cases, however, training replay may be necessary for post-hoc data restoration. In this chapter, we documented a system of method for efficient record-replay of model training. Methodical hindsight logging consists of: (i) periodic checkpointing, (ii) block memoization, and (iii) checkpoint resume. To extend the benefits of methodical hindsight logging to novices and experts alike, we open source the flor suite for hindsight logging, which includes tools for: (i) low-overhead background logging, (ii) adaptive periodic checkpointing, and (iii) end-to-end instrumentation for efficient and fully automatic record-replay. We evaluated methodical hindsight logging to show that it achieves the goal of efficient record-replay, and then compared the instrumentation library provided by flor against the methodical expert-tuned approach, and find that the performance is comparable.

Chapter 5

Change Over Time: Multiversion Hindsight Logging for Continuous Training

Production Machine Learning is a data-intensive process, involving continuous retraining of multiple versions of models, many of which may be running in production at once. When model performance does not meet expectations, Machine Learning Engineers must explore and analyze numerous versions of code, logs and training data to identify root causes and mitigate problems. Traditional software engineering tools fall short in this data-rich context. FlorDB introduces *multiversion hindsight logging*, a form of acquisitional query processing that allows engineers to use the most recent version's logging statements to query past versions' logs, even when older versions logged different data. FlorDB provides a *replay query* interface with accurate cost estimates to help the end-user refine their queries. Once a replay query plan is confirmed, logging statements are propagated across code versions, and the modified training scripts are replayed based on checkpoints from previous runs. Finally, FlorDB presents a unified relational view of log history across versions, making it easy to explore behavior across past code iterations. We present a performance evaluation on diverse benchmarks confirming scalability and the ability to deliver real-time query responses, leveraging query-based filtering and checkpoint-based parallelism for efficient replay.

5.1 Introduction

Model development for machine learning (ML) is an iterative, experimental and data-intensive process that differs from traditional software engineering. The primary goal in model development is to boost predictive performance, leading developers to adopt empirical methods involving thorough logging and continuous updates to code and training data [208]. A recent study highlighted the importance of high-speed experimentation in this field: as one participant stated, "the most important thing to do is achieve scary high experimentation velocity" [176]. This approach, however, generates a significant data management challenge for engineers, who must handle numerous iterations of code, datasets, and logs.

For this aggressively agile approach to be successful, machine learning engineers (MLEs) must be able to retroactively examine rich and diverse data from past runs to guide ongoing experiments [46]. One method to enable model developers to "query the past" is by generating and maintaining comprehensive experiment logs, but this is burdensome and unrealistic. Ensuring comprehensive logs requires (a) foresight to add logging statements to the code for all relevant metrics, without knowing which ones are relevant in advance, and (b) data management discipline for the many log files, code versions, training data and requisite metadata that ensues.

In this chapter, we explore a different approach: we show it is possible to query the past of machine learning experiments efficiently and on-demand, even in the absence of experiment logs. We achieve this automatically on the developer's behalf, by managing model training checkpoints and the corresponding versions of code and data, and exposing experimental history—both what was actually logged *and what could have been logged*—as a virtual database that can be queried with familiar APIs like SQL or dataframe libraries.

Multiversion Hindsight Logging

Hindsight logging [48] is a lightweight record-replay technique that allows MLEs to add logging statements to a long-running Python program after it executes, and then incrementally *replay* parts of the program very quickly to generate the log outputs that would have emerged had the statements been in the code originally. This technique encourages MLEs to operate optimistically and flexibly with a minimal initial logging scheme, typically restricted to loss and accuracy metrics during the training phase. Additional logging is added *retroactively*, when evidence of issues arises (e.g., in deployment) and further key information is required for debugging.

We realized the concept of hindsight logging in Flor, a record-replay system specifically designed for model training [48]. Flor embodies two salient features: low-overhead checkpointing, and low-latency replay from checkpoint. Flor checkpointing is adaptive and runs in the background, facilitating low-overhead checkpointing, and limiting the computational resources expended during model training. Simultaneously, the system ensures reduced latency during replay by leveraging memoization and parallelism through checkpoint-resume.

While hindsight logging is a useful core technology, it is insufficient on its own to address typical ML workflows, which invariably involve many training runs. MLEs, in their pursuit of high-velocity experimentation, often wish to revisit not just the most recent run of an experiment, but also prior runs that were performed using different versions of code and data. This presents a complex challenge that goes beyond the capabilities of traditional logging and debugging tools.

To tackle this challenge, in this chapter we introduce *multiversion hindsight logging*, and a system called *FlorDB* that provides an efficient and easy-to-use solution. Multiversion hindsight logging is designed to track and manage multiple versions of ML experiments. In doing so, it provides a more comprehensive and effective solution than simple hindsight logging, ensuring that MLEs can readily look back through their iterative development processes, thereby better understanding and learning from their experimentation history. Achieving these goals requires overcoming a set of technical challenges described below.

Contributions

FlorDB, a Multiversion Hindsight Logging system, provides a tabular query interface, treating every run of a version as a set of rows, and each logging statement as a "virtual column". FlorDB includes automatic version control through Git and is designed to interoperate with multiple ML experiment management tools. In addressing the inherent challenges of implementing Multiversion Hindsight Logging, FlorDB makes the following key contributions:

1. **Historical Query Manager using a Unified Relational Model**: FlorDB presents MLEs with a simple metaphor of a relational view of queryable log results—whether the log statements were previously materialized, or need to be generated on demand via hindsight logging. The unified relational model allows users to issue ad-hoc queries using familiar SQL or pandas, thereby simplifying the process of exploring historical code executions .

2. **Multiversion Hindsight Logging as Acquisitional Query Processing**: FlorDB extends the concept of Acquisitional Query Processing (AQP) [116] by seamlessly integrating data acquisition into query execution through experiment replay. This framework differs from traditional systems that query static datasets. Selected experimental versions undergo sequential processing: logging statements are propagated to generate required data, followed by the partial or parallel replay of modified training scripts.

3. **Accurate Time Estimation for Query Refinement**: FlorDB incorporates highly accurate cost prediction for estimating the runtime of on-demand replay queries. By leveraging runtime profiling statistics collected during the initial record phase, FlorDB provides highly accurate (error $\leq 5\%$) replay cost estimates. This precision enables developers to refine their queries *before* being bogged down by unnecessarily lengthy execution times.

FlorDB's integrated features provide machine learning engineers with a flexible relational abstraction to capture and query the extended histories of their ML experiments. By harnessing the framework of FlorDB, MLEs can more quickly iterate from prior attempts to successful models.

5.2 Scenario: Catastrophic Forgetting

For readers who are not MLEs, we motivate FlorDB with an example of *catastrophic forgetting*, a pressing ML issue that is especially problematic in production settings [176]. This

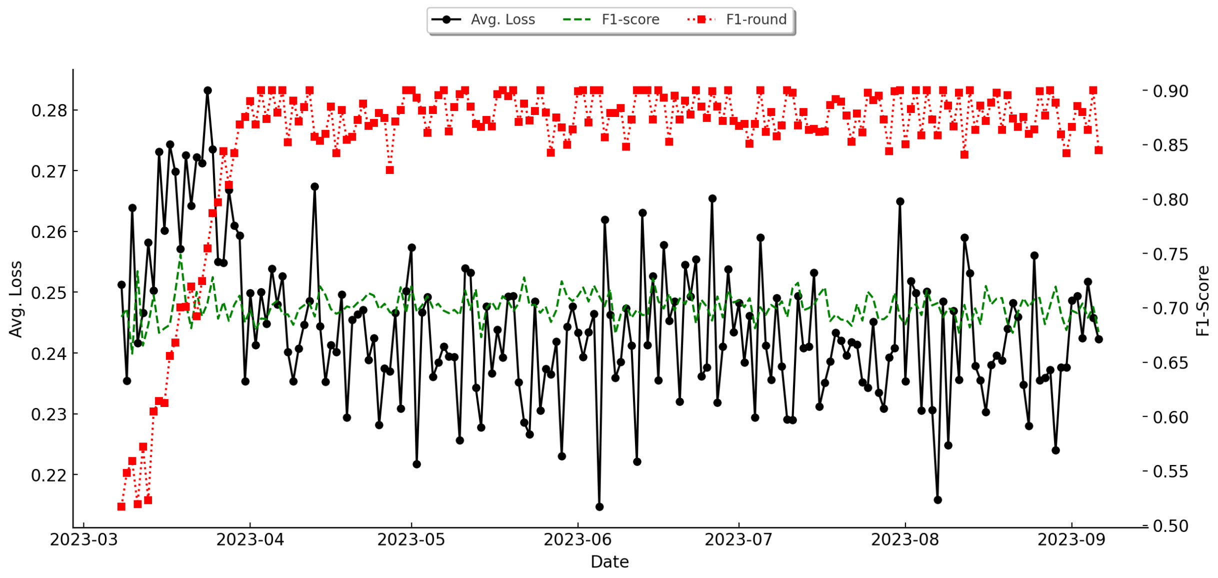


Figure 5.1: Average training losses and F1-scores for Alice's object detection model over the last 6 months. The model undergoes continuous training, with batches of labeled data added approximately twice a month. These batch dumps result in temporary fluctuations in the loss. `F1-round` is the F1-score for roundabouts; `F1-score` is the global F1-score.

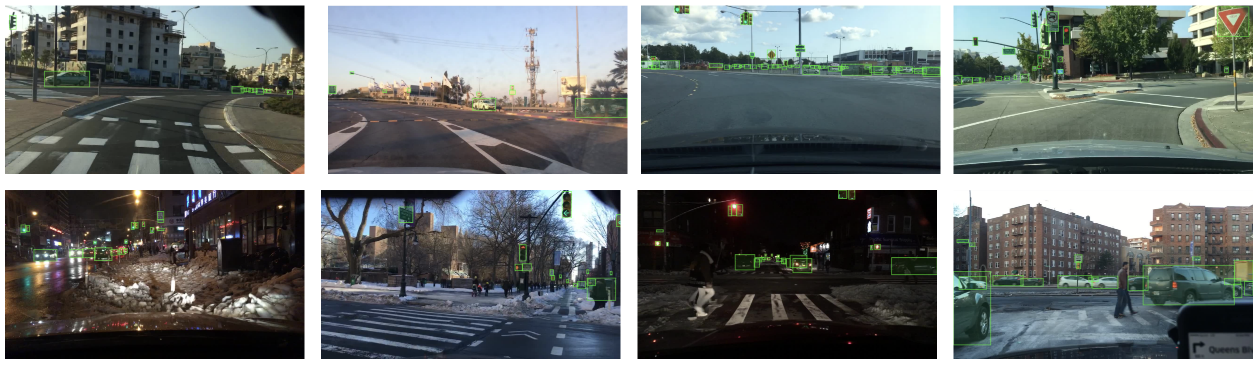


Figure 5.2: BDD100K dashcam images with bounding boxes. Top row contains sample images used by Alice for fine-tuning on roundabouts; bottom row corresponds to images for which the model fails to detect pedestrians.

phenomenon occurs when a model, upon being fine-tuned for a new task, loses its ability to perform well on a previously learned task. This creates a balancing act between updating the model (for new data or tasks) and maintaining its performance (for older tasks).

Object Detection in Autonomous Vehicles

Imagine an MLE named Alice, responsible for the ongoing fine-tuning of an object detection model used in autonomous vehicles. Operating in a dynamic environment, Alice frequently updates the model with bi-weekly batches of labeled data. Her current focus is the specialized task of object detection within traffic roundabouts. Standard metrics like loss functions and F1-scores, shown in Figure 5.1, serve as her evaluation benchmarks.

However, Alice encounters an unexpected problem. A colleague, Bob, informs her of persistent failures in pedestrian detection that have arisen in her new model versions. Upon examining a representative set of images (refer to the bottom row of Figure 5.2), she verifies the issue.

Responding to Performance Regressions

To combat this specific case of catastrophic forgetting in pedestrian detection, Alice employs hindsight logging for two key actions:

- **Model Rollback**: Alice would like to add hindsight logging statements to calculate the F1 score *for pedestrian detection* across previous model versions. This would allow her to identify the latest point where the model still successfully detected pedestrians, enabling an informed rollback.
- **Alert Tuning**: To prevent the issue from recurring, Alice wants to add alerting logic that checks the value of the pedestrian detector F1 score. However she needs to avoid "alert fatigue", when too many spurious alerts are issued. To ensure that her altering is calibrated, she again utilizes hindsight logging to ensure the F1 thresholds for alerting is not triggered by valid runs.

Alice's scenario underscores a significant limitation in current approaches to model versioning and logging. Given the expansive history of model versions, it is neither practical nor efficient for Alice to manually revisit past iterations of training to insert additional logging statements, replay training, and analyze the results. Multiversion hindsight logging provides the high performance replay mechanisms for retroactive logging statement replay with automated back-propagation of those statements to previous versions of code for a comprehensive evaluation. This enables Alice to retroactively analyze metrics (e.g. F1 scores specific to pedestrian detection) across all historical versions.

5.3 User Experience & API

In this section, we explore how Alice leverages FlorDB[1] for addressing performance regressions in pedestrian detection, notably without pre-logged F1-scores for pedestrians. FlorDB

[1]Available on PyPI, installable via `pip install flordb`

```python
import flor
import torch
from random import randint

hidden_size = flor.arg("hidden", 500)
learning_rate = flor.arg("lr", 1e-3)
batch_size = flor.arg("batch_size", 32)
seed = flor.arg("seed", randint(0,1e10))
...
ckpts = {'model': net, 'optimizer': optimizer}
with flor.checkpointing(**ckpts):
    for e in flor.loop("epoch", range(epochs)):
        for x in flor.loop("step", trainloader):
            inputs, labels = x
            optimizer.zero_grad()
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            flor.log("loss", loss.item())
        accuracy = validate(model)
        # Other validation logic

# logs F1-scores
eval(net, testloader)
```

Figure 5.3: Alice's PyTorch training with Flor API.

projid	tstamp	filename	hidden	lr	batch_size	seed	f1_score	f1_round	*f1_ped*
roundabouts	2023-06-23	train.py	500	0.001	32	81	0.7022215	0.90000000	0.5192609
roundabouts	2023-06-24	train.py	500	0.001	32	63	0.7043473	0.88589064	0.5590010
roundabouts	2023-06-29	train.py	500	0.001	32	157	0.6912616	0.86693724	0.5482590
roundabouts	2023-06-30	train.py	500	0.001	32	42	0.6994197	0.89759576	0.5170792
...	...	...	...	...	...	...	...	...	...
roundabouts	2023-08-27	train.py	500	0.001	32	213	0.6982518	0.84088466	0.2392322
roundabouts	2023-08-28	train.py	500	0.001	32	12	0.6770945	0.90000000	0.2493409
roundabouts	2023-08-29	train.py	500	0.001	32	333	0.7026935	0.86367500	0.2493730
roundabouts	2023-08-30	train.py	500	0.001	32	99	0.6818689	0.88571969	0.2351695
roundabouts	2023-08-31	train.py	500	0.001	32	475	0.6995291	0.87202768	0.2285919
roundabouts	2023-09-01	train.py	500	0.001	32	198	0.6850775	0.88872464	0.2265961
roundabouts	2023-09-02	train.py	500	0.001	32	94	0.7200135	0.89775851	0.2485293
roundabouts	2023-09-03	train.py	500	0.001	32	37	0.7030796	0.86352228	0.2305369
roundabouts	2023-09-04	train.py	500	0.001	32	292	0.7084305	0.89572726	0.2373573
roundabouts	2023-09-05	train.py	500	0.001	32	70	0.7090070	0.87411934	0.2364624
roundabouts	2023-09-06	train.py	500	0.001	32	553	0.6865721	0.84508553	0.2418998

Figure 5.4: Alice's `flor.dataframe` before and after hindsight logging (black and green text, respectively).

enables her to effectively identify and revert to a more efficient model version, and fine-tune alert thresholds. Our discussion underscores the FlorDB API's role in simplifying the ML workflow, facilitating rapid iteration and in-depth analysis for users with varied expertise. We illustrate the API's functionality in key activities such as logging, parameter management, and hindsight logging for model refinement. Key API features include:

- `flor.log(name: str, value: T) -> T`
  Logs a given value with the specified name. Useful for tracking variables or parameters during an experiment run.

- `flor.arg(name: str, default: T) -> T`
  Reads a value from the command line if provided; else uses the default value. During replay, retrieves logged values from history. Good for setting configuration parameters and allowing for quick adjustments via the command line.

- `flor.loop(name:str, vals:Iterable[T]) -> Iterable[T]`
  Functions as a Python generator that maintains global state between iterations. Useful for "indexing" replay and coordinating checkpoints.

- `flor.checkpointing(**kwargs) -> ContextManager[None]`
  Scopes the set of objects to be checkpointed, such as a model or optimizer, via a Python context manager. Checkpointing is done adaptively at `flor.loop` iteration boundaries, based on the checkpoint size and the loop iteration time [48].

- `flor.dataframe(*args) -> pd.DataFrame`
  Produces a Pandas DataFrame of FlorDB log information with a column corresponding to each argument in `*args`. The DataFrame also contains "dimension" columns, such

as the project id, version id, timestamp, and so on. It is the default view in FlorDB for querying log data and selecting model checkpoints to load, using either Pandas or DataFrame-compatible SQL engines like DuckDB [150].

Running Experiments

Returning to our scenario, Alice executes her model training via the following command that starts her `train.py` script with specific hyper-parameters:

```
python train.py --kwargs lr=0.001 batch_size=32
```

Alice's script (Figure 5.3) uses native Python functionality to run her ML experiment. Hyper-parameters such as learning rate and batch size are passed in using standard command-line arguments, which makes hindsight logging seamless and capable of interoperating with other tools typically used in the ML workflow.

Args & Logs

In the context of running experiments, Alice utilizes the Flor API to enhance configurability and traceability. The `flor.arg` function (lines 5-8 in Figure 5.3) is instrumental in parameterizing her experiments, allowing her to define hyper-parameters such as `hidden_size`, `learning_rate`, `batch_size`, and `seed` either via the command-line or by utilizing default values specified within the script. This ensures adaptability and reproducibility of her experiment's configuration. In addition, `flor.log` is employed (line 20 in Figure 5.3) to record the experiment's loss metrics and associated metadata at each optimizer step. The logging of the loss captures the dynamics of the model's performance, enabling Alice to monitor the training process.

Checkpointing on Loop Boundaries

Alice's code defines the set of objects to be checkpointed using `flor.checkpointing` (line 11); within that context, Alice calls `flor.loop` (lines 12-13) to use the checkpoints and control record-replay. After each iteration of the outermost loop, which here corresponds to the completion of an "epoch," FlorDB assesses whether to checkpoint. This decision is based on a balance between the checkpoint's size and the iteration's duration, adhering to a strategy from prior work [48]. Checkpoints are created adaptively, approximately once per epoch.

Automatically commit changes to Git

At the end of the experiment, FlorDB writes a JSON logfile containing the execution's sequence of log records to the experiment repository, and commits all changes. This is so future users of the repository can view historical logs and arguments, and better reproduce the experiments. By committing after every run, Flor ensures historical versions of code are later available for hindsight logging.

Experiment Analysis and Hindsight Logging

Alice reviews FlorDB's `flor.dataframe` output, comparing F1-scores across datasets (see Figure 5.4). After receiving insights from her colleague, Alice shifts her analysis focus to the pedestrian F1-scores. By editing `train.py` to include a new `flor.log` for `f1_ped`, Alice leverages hindsight logging to populate this metric retroactively. She invokes the replay feature via CLI as follows:

```
python -m flor replay f1_ped "tstamp >= '2023-06-23'"
```

This command extends `f1_ped` logging to any past experiments dated June 23, 2023 or later, in preparation for subsequent analysis.

Upon initiating the replay command, Alice is shown a schedule of experiments slated for replay and must confirm to proceed. After receiving confirmation, Flor prepares to replay the selected prior runs: it restores their versions from github, adds the new `f1_ped` logging statement to each, and executes all versions using the configuration variable values originally logged via `flor.arg`. Notably, during this replay, Flor's hindsight logging functionality can bypass the main loop (lines 12-22 in Figure 5.3) to directly load the model's final state, since the `f1_ped` statement is computed post-training (line 25 in Figure 5.3).

Once replay is finished, Alice views the updated results with the following command:

```
cols = ['hidden', 'lr', 'batch_size', 'seed']
cols += ['f1_score', 'f1_round', 'f1_ped']
flor.dataframe(*cols)
```

This command now generates a table including the `f1_ped` metric (green text in Figure 5.4) for entries dated June 23, 2023, or later. This new data reveals a previously unnoticed performance regression in pedestrian classifications, as Bob had indicated (`f1_ped` values in the range `tstamp` $\geq$ 08-27).

To further refine the analysis and calibrate alert thresholds, Alice updates the validation function to add a new `flor.log` statement as part of an assertion that pedestrian F1-scores never fall below some threshold. For retroactive application, she again employs FlorDB's replay feature:

```
python -m flor replay f1_alert \
"tstamp >= '2023-06-23' and tstamp < '2023-07-01'"
```

This process selectively executes each epoch's validation logic (lines 21-22), avoiding re-training (lines 13-20) by loading final epoch states from checkpoints. FlorDB's control via

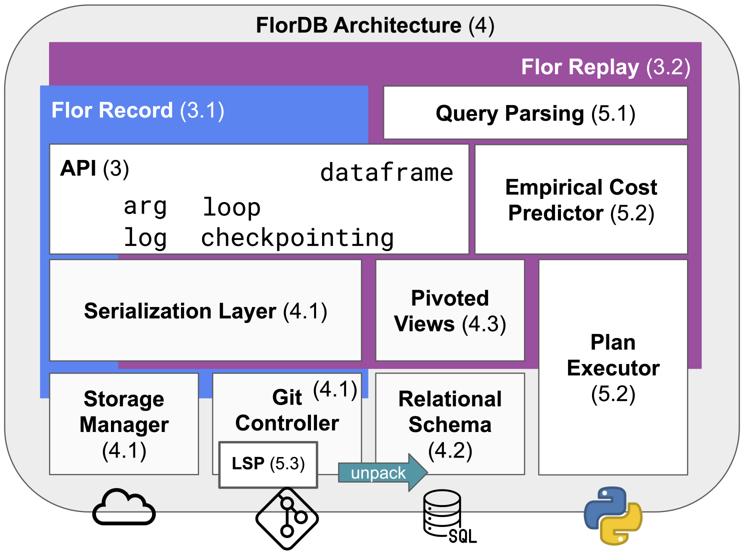


Figure 5.5: FlorDB architecture diagram with subsection headers in parentheses.

flor.loop facilitates this selective process: the outer loop cycles through epochs using checkpoints, while the inner loop is bypassed.

5.4 System Architecture

The design of FlorDB aims to provide a unified platform for the management of code, checkpoints, logs, and their evolution over time (Figure 5.5). FlorDB maintains a cross-referenced relational database that interconnects code versions in git, large files in your preferred storage solution (e.g. an S3 bucket), and metadata stored in the database. This section outlines the mechanisms FlorDB uses for executing experiments and replaying them for hindsight logging, highlighting how it stores, manages and tracks the data and metadata of model training.

Storage and Data Layout

As shown in the bottom of Figure 5.5, FlorDB's storage architecture consists of the following three units:

- **Git Repository**: For code version control, FlorDB uses git to capture the state of the working directory and JSON log files post-execution. FlorDB and the user commit to the same Git repository, but FlorDB auto-commits to a dedicated branch for added safety — the user may treat that branch as any other branch. This facilitates a comprehensive tracking of code versions and more transparent sharing of execution metadata, enhancing experiment reproducibility and analysis.

- **Relational Database**: Following execution and logging to JSON, FlorDB unpacks and normalizes data from the JSON logs to populate a relational database (which is pluggable; FlorDB uses SQLite by default). This process transforms the semi-structured JSON data into a structured format, suitable for efficient querying and analysis.
- **Object Storage**: This component is responsible for storing large files and checkpoints, integrating with the database system to offer scalable and flexible storage solutions, whether locally or in the cloud.

This heterogeneous approach to storage ensures that FlorDB is well-equipped to manage the complexities of code versioning, data checkpointing, and metadata storage.

Serialization Layer

The API utilizes a serialization layer to prepare objects for storage. This layer differentiates between object types — PyTorch objects are serialized using native PyTorch functionality, while cloudpickle provides a general-purpose serialization solution. For in-depth information on how fork and copy-on-write are used for background serialization, the reader is referred to a technical report [108].

Relational Schema

The relational schema of FlorDB is a structured and normalized view over the JSON logs. As depicted in Figure 5.6, the `logs` table is central to this schema, managing the data and relevant metadata of experiments. The schema for the `logs` table includes:

- **projid**: Basename of the root-level working directory.
- **tstamp**: Datetime marking when the run was started.
- **filename**: Name of the source file producing the log entry.
- **ctx_id**: An integer acting as a foreign key to associate each log entry with a related entry in the `loops` table.
- **value_name**: text descriptor for the variable being logged.
- **value**: The actual data logged, stored as text.
- **value_type**: Integer classifying the *type* of data logged.

Complementing the `logs` table, the `loops` table tracks the context of the execution flow (the call stack) as follows:

- **ctx_id**: Unique identifier for a specific loop context.
- **parent_ctx_id**: References the parent context in nested loops.
- **loop_name**: The name or identifier of the loop.
- **loop_entries**: Number of entries or iterations within the loop.
- **loop_iteration**: Specific iteration count within a loop.

The "unpack" process involves converting logging objects from JSON format into the relational structure of the database. Each JSON log record is parsed. The data is then mapped to the relevant columns in both the `logs` and `loops` tables. In the case of a *null* loop indicating a top-level log entry, the log record is mapped solely within the `logs` table; in cases of nested loops, the JSON object encapsulates the hierarchical structure of an experiment's iterations, and the corresponding log record is reflected in both `logs` and `loops` tables.

Pivoted Views

The `flor.dataframe` presents the content of the database in a single-table view (Figure 5.7). This view is generated from the relational schema (Figure 5.6) by joining the `logs` table with additional dimensions of data, such as the `loops` table, which are essential for capturing the full context of each experimental run. This is followed by a series of transformations that pivot the combined data, turning relational values (e.g., `logs.value_name`, `loops.loop_name`) into column headers, to create a wide-format table where each unique loop and log entry name becomes a distinct column in the DataFrame (e.g. Figure 5.4).

Domain Mapping

We use the `flor.dataframe` as the default Flor view because of its easy-to-understand application-level semantics. Specifically:

- **Running Experiments → Adding Rows:** The execution of a new experiment results in the addition of rows to `flor.dataframe`. Each row in the DataFrame signifies a discrete iteration within an experiment, collating related data such as metrics, hyperparameters, and state.
- **Adding Logging Statements → Adding Virtual Columns:** The inclusion of logging statements in the source code induces FlorDB to add virtual columns to the view produced by `flor.dataframe`. These virtual columns can then be referenced in subsequent queries like any other column in `flor.dataframe`.

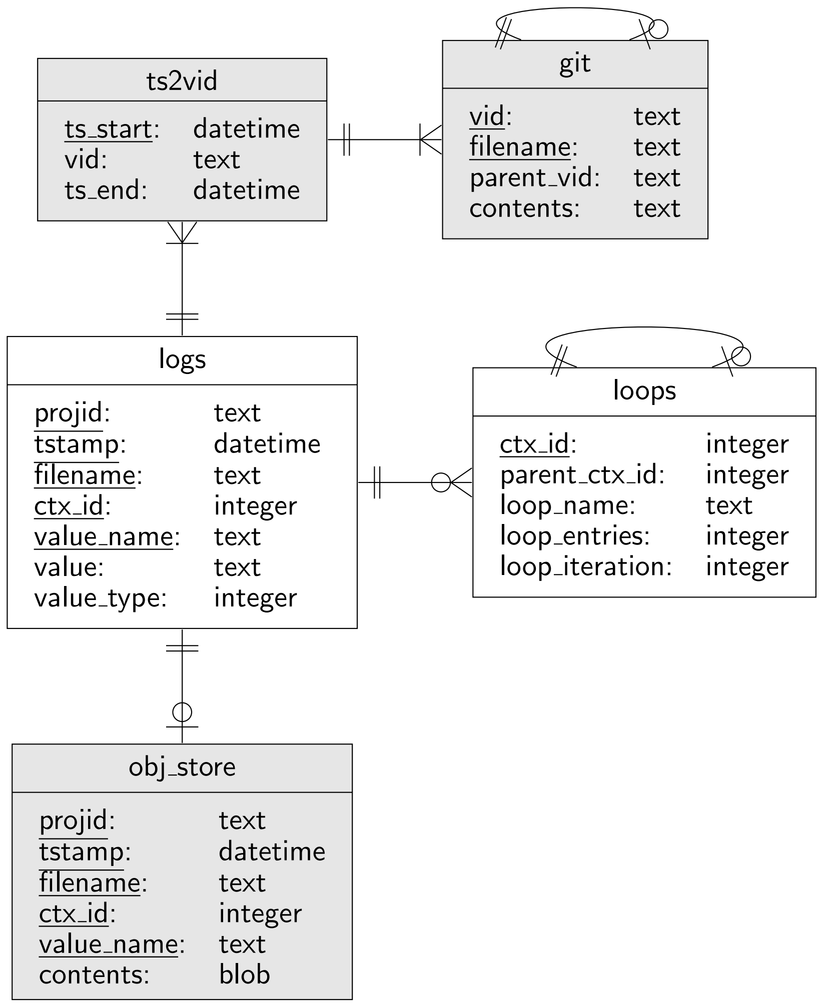


Figure 5.6: Data Model Diagram in Crow's Foot Notation. Tables with a white background are basic; those in gray are virtual.

- **Replay from Checkpoint → Backfilling Nulls:** Replaying an experiment from checkpoints corresponds to the backfilling of *null* values in the `flor.dataframe` view. This capability is critical to achieving the hindsight logging abstraction that FlorDB supports.

The ML developer can envision the `flor.dataframe` as a universal relational view over the columns they have selected, where the rows represent individual experiment runs or, in the presence of loops, iterations within those runs. The columns in the `flor.dataframe` view are potentially infinite; they can be defined post-hoc and subsequently populated using replay from a checkpoint. This fluidity in defining and back-filling columns and data allows a relational model to adapt to an experimenter's evolving analytical needs.

```
def dataframe(conn, *args):
    dataframes = []
    loops = pd.read_sql("SELECT * FROM loops", conn)
    for val_name in args:
        logs = pd.read_sql(f"""
          SELECT * FROM logs
          WHERE value_name = "{val_name}"
          """, conn)
        logs = logs.rename(columns={"value": value_name})
        # Unroll loop context
        while logs["ctx_id"].notna().any():
            # Iterate until fixpoint
            logs = pd.merge(left=loops, right=logs,
            how="inner", on=["ctx_id"])
            ln = logs["loop_name"].unique()
            logs = logs.drop(columns=["loop_name", loop_entries])
            logs = logs.rename(columns={"loop_iteration": ln[0]})
            logs["ctx_id"] = logs["p_ctx_id"]
            logs = logs.drop(columns=["p_ctx_id"])
        logs = logs.drop(columns=["ctx_id"])
        dataframes.append(logs)
    all_joined = reduce(outer_join_on_common_columns, dataframes)
    cols = [c for c in all_joined.columns if c not in args]
    return all_joined[cols + list(args)]
```

Figure 5.7: Python code implementing data reshaping for `flor.dataframe`. This figure illustrates the sequence of operations—merging, transforming, and pivoting—used to construct the experiment view within FlorDB.

5.5 Acquisitional Query Processing

FlorDB extends the concept of Acquisitional Query Processing (AQP) [116], which involves efficient data acquisition during query execution. AQP was originally presented in the context of live sensing; here we acquire data by re-running training code from checkpoints. The process begins at the query construction stage, where criteria for selecting experimental versions are established (Section 5.5). Each selected version has logging statements propagated to it in order to generate the required data (Section 5.5). Subsequently, a modified training script is executed with a `--replay_flor` flag, enabling replay query operators to perform partial or parallel model training replay (Section 5.5). In this section, we describe multiversion hindsight logging via the sequence of mechanisms by which FlorDB integrates data acquisition into query execution. This also paves the way for comprehensive post-hoc analysis and continuous model refinement.

Query Parsing

The `flor replay` operation begins with query construction, a step that determines the scope and precision of the retrospective analysis. This stage is crucial for identifying the code versions to be replayed and for setting the level of detail in the logging process. The query is formulated using the following command structure:

```
python -m flor replay [-h] VARS [where_clause]
```

The components of the query include:

- **VARS**: This argument specifies the logging variables that the user wishes to generate during the replay. These variables correspond to the log statements that will be regenerated across different code versions.
- **where_clause**: This optional argument acts as a filter to refine the replay's scope. It can be used to constrain the replay to certain conditions, such as experiments conducted within a specific timeframe or parameter set. The `where_clause` is evaluated against a `flor.dataframe` prior to replay.

By judiciously crafting the query, users can ensure that the replay operation is both focused on the necessary aspects of the experiment and efficient in terms of computational resource usage.

Empirical Cost Prediction & Plan Execution

Machine Learning (ML) training scripts can be long-running, and even selective replay can be time-consuming. Therefore, ML engineers need to know how long a replay will take, to decide whether further query refinement is needed. Traditional query cost estimation is a notoriously hard problem, often exhibiting orders of magnitude errors in estimation [102]. By contrast, we can estimate query runtime for replay executions extremely accurately (Figure 5.8). This is because FlorDB profiles runtime information during the record phase. When initiating a Flor replay, these runtime statistics are queried using the `flor.dataframe` API call. This allows us to reliably predict the time required for replay, enabling users to tailor their queries for efficiency. We categorize the cost estimation into four levels:

1. **Prefix Scan**: Executes statements before the main loop, logging preliminary setup operations (e.g., lines 1-10 in Figure 5.3. This level is useful when examining data preparation and featurization.
2. **Suffix Scan**: Executes the initial setup code (as in prefix scan), loads the end state of the outermost loop from a checkpoint, and runs the final script segments (lines 23-EOF

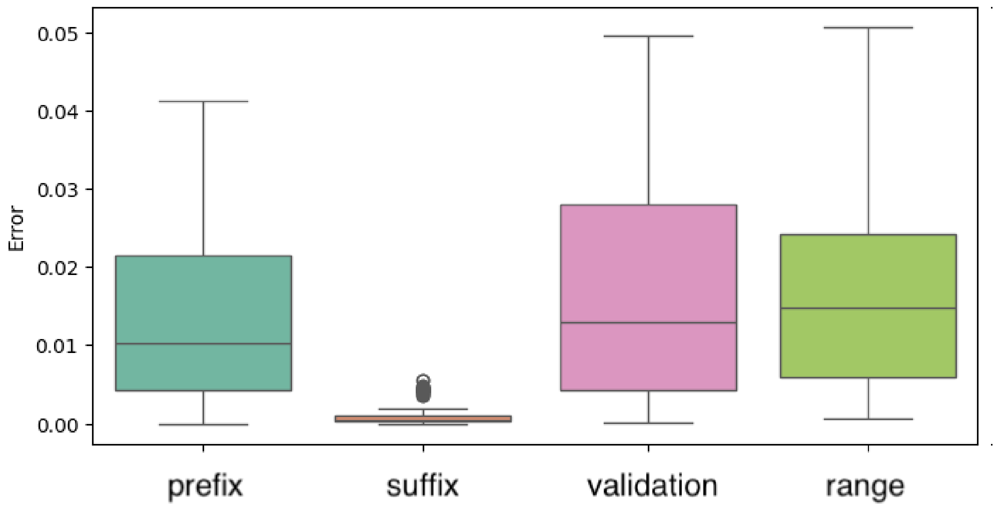


Figure 5.8: Time estimation error per replay operator; micro-benchmarks correspond to a 2-layer neural network.

in Figure 5.3). This level provides a more detailed view, useful when examining the results of training (e.g. accuracy, recall, F1-scores).

3. **Validation Scan**. Steps through the main training loop to execute model validation logic (lines 21-22 in Figure 5.3)., loading checkpoints once per epoch but skipping the nested training loop. This level is useful for examining the validation process in detail.

4. **Range Scan**: Runs the training loop in depth (lines 12-22 in Figure 5.3) over a selected range to capture nested logging statements — used for the most granular analyses (e.g. of gradients). A range scan running from epoch 0 to N is considered a full scan. To achieve replay parallelism, FlorDB executes range scans over non-overlapping intervals.

Plan Execution

Once the replay's depth level has been determined by a static analysis of the training script, and the end-user has accepted the replay plan, a replay subprocess is invoked for each version in the plan, with CLI flags and arguments formatted as strings parameterizing the replay's depth level. Flor replay arguments can be used to break out of the execution in a prefix scan upon encountering the first `flor.loop` or set to skip the first `flor.loop` loading its end checkpoint as part of a suffix scan; or set to step into the the first `flor.loop` as part of a validation scan, and so on. As shown above, validation scans and range scans control access to the outermost loop, and the nested loop, a key difference being that in validation scan the nested loop is skipped but not in the range scan.

Logging Statement Propagation

After selecting the versions and log variables for replay, but before the execution of individual Flor replay subprocesses, FlorDB adapts the code from the git repository to incorporate new log statements. This involves propagating log statements from a newer version Y to an older version X ($X < Y$). The core challenge here is one of code-block alignment: we must identify the most appropriate line in version X to insert a logging statement that was originally in line L of version Y. Code alignment is a fundamental task in software engineering, essential for understanding and managing changes in source files [69, 96]. It allows developers to track modifications and maintain consistency across different versions of a file. In this context, GumTree emerges as a state-of-the-art technique for code block alignment.

Why GumTree is Sufficient for Our Use Case.

FlorDB's code alignment uses GumTree [43] for aligning code blocks, aiding in the accurate insertion of log statements across versions. The `flor.loop()` function establishes stable anchor points, with main and nested training loops typically serving as consistent markers across versions. This consistency, enhanced by GumTree's alignment capabilities, ensures accurate log statement propagation. While GumTree is capable of aligning code blocks without Flor API anchors, the presence of these static loop identifiers all but ensures its efficiency and accuracy: in all instances of logging statement propagation evaluated in Section 5.6, this combined approach was successful.

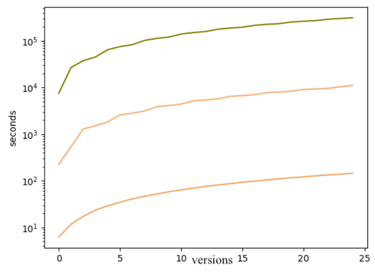
To assess the effectiveness of these techniques, we conducted an evaluation focusing on accuracy and comparing GumTree with the Myers algorithm [132, 43]. The evaluation included synthetic benchmarks, randomly mutated PyTorch programs, and ecological benchmarks from GitHub. Our findings, detailed in a technical report [37], show GumTree's superior performance, especially in handling code refactoring and variable renaming.

5.6 Evaluation

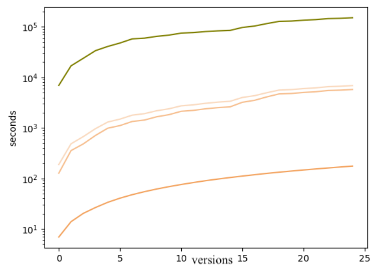
This evaluation aims to demonstrate the key strengths of FlorDB: its ability to efficiently handle multiversion hindsight logging and its potential for significant speed-ups during replay. We showcase FlorDB's versatility by applying it to a diverse range of model architectures,

Table 5.1: Computer vision and NLP benchmarks used in our evaluation.

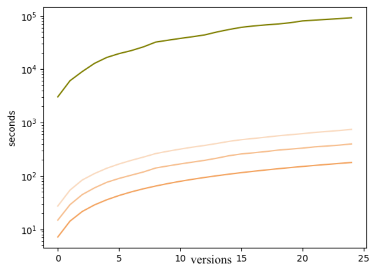
Model	Model Size	Data	Data Size	Objective	Evaluation	Application
ResNet-152 [61]	242 MB	ImageNet-1k [164]	156 GB	image classification	accuracy	computer vision
BERT [40]	440 MB	Wikipedia [45]	40.8 GB	masked language modeling	accuracy	natural language processing
GPT-2 [152]	548 MB	Wikipedia [45]	40.8 GB	text generation	perplexity	natural language processing
LayoutLMv3 [73]	501 MB	FUNSD [59]	36 MB	form understanding	F1-score	document intelligence
DETR [23]	167 MB	CPPE-5 [36]	234 MB	object detection	μ-precision	computer vision
TAPAS [63]	443 MB	WTQ [143]	429 MB	table question answering	accuracy	document intelligence



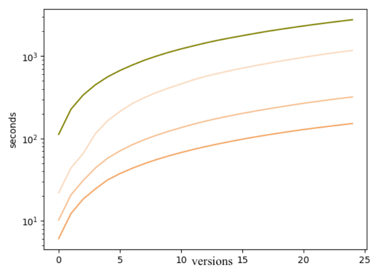
(a) ResNet-152



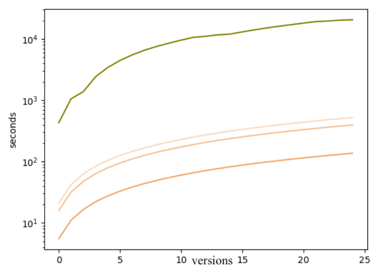
(b) BERT



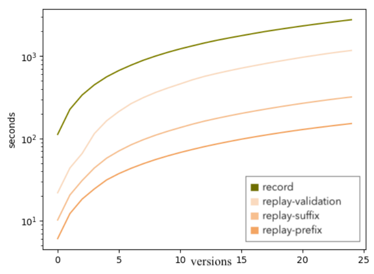
(c) GPT-2



(d) LayoutLMV3



(e) DETR



(f) Tapas

Figure 5.9: Comparative visualization of processing times for different Flor record-replay operations against the number of experiment versions. Note the logarithmic scale on the y-axis, emphasizing the time disparities between different operations.

as detailed in Table 5.1. This table summarizes each model's characteristics, evaluation metrics, and real-world applications. Building upon our prior work that established Flor's low-overhead recording and parallel replay capabilities for single versions [48], this study focuses on FlorDB's ability to generalize and scale effectively across multiple versions. Our evaluation delves into two key areas: scalability and responsiveness. FlorDB's ability to handle growing numbers of model versions is assessed, ensuring linear scaling without performance degradation or cross-version interference. Experiments were run on our own server with 4 CPU cores (11th Gen Intel Core i7), 32 GB of RAM, 1 TB SSD, and a GeForce RTX 3070 GPU with 8GB GDDR6.

Efficient Linear Scaling across Versions

To assess FlorDB's ability to manage multiversion hindsight logging without cross-version interference, we evaluated its performance across different model architectures. For each model architecture, we measured the runtime for the following modes of execution (as defined in Section 5.5):

- **record**: First run of model training; generates checkpoints.

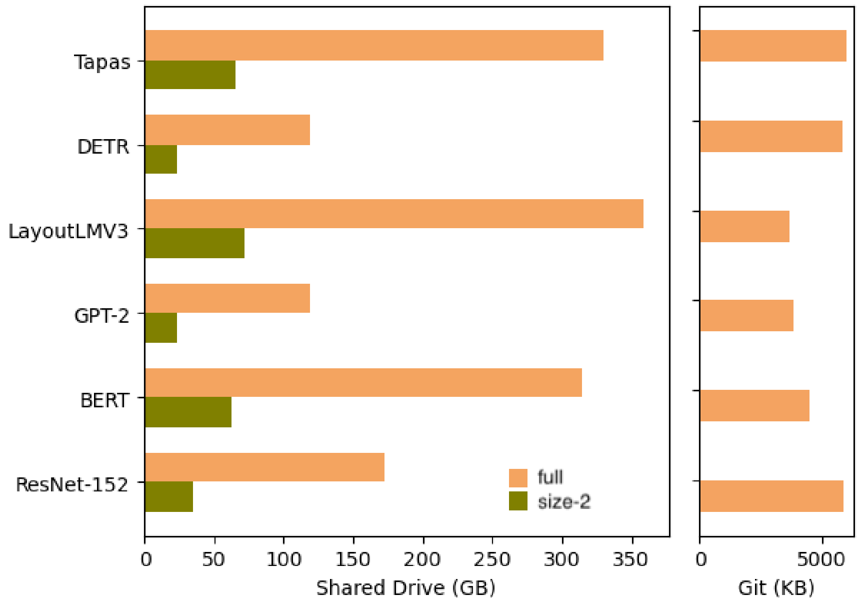


Figure 5.10: Storage Requirements

- **replay-prefix**: query runs a `prefix scan` over all versions.
- **replay-suffix**: query runs a `suffix scan` over all versions.
- **replay-validation**: query runs a `validation scan` over all versions.

Our findings, depicted in Figure 5.9, reveal that FlorDB's time complexity scales linearly with an increasing number of versions, as demonstrated by the log-scale on the y-axis. This evidence of linear scalability, devoid of interaction effects or incremental overhead, validates FlorDB's ability to handle both large models and big datasets.

Interactive Response Times

In evaluating the impact of different replay operations on system response times (see orange lines in Figure 5.9), the replay-prefix operation maintains the fastest processing times, consistently delivering results in under 10 seconds. Even as the number of versions increases, the processing time for replay-prefix remains the most stable, ensuring a responsive user experience. In contrast, replay-suffix and replay-validation operations exhibit longer processing times as the version count rises, with replay-validation times increasing more steeply. One factor is the amount of time it takes to evaluate the model. While replay-suffix may still offer interactive speeds at lower version counts, replay-validation quickly surpasses the interactive threshold: in scenarios where immediate feedback is crucial, users can refine the selectivity of their queries for faster response times (as discussed in Section 5.5).

Storage Requirements

Despite frequent commits across numerous versions, Git repositories remain remarkably compact, typically under 5MB (see right pane of Figure 5.10). In contrast, checkpoints for large models stored on shared storage can easily consume hundreds of gigabytes (see orange bars in the left pane). To address potential local storage depletion, FlorDB's default approach is to spool least-recently-used checkpoints to a cloud object store like S3[2]. If this is not feasible, FlorDB may retain a subset of checkpoint per version—as few as just two (see green bars in Figure 5.10)—evicting the rest. FlorDB retains checkpoints that are evenly spaced across iterations. This ensures balanced work partitions and avoids stragglers that would hinder replay parallelism. Importantly, even retaining just two checkpoints (one mid-run, one at completion) offers significant benefits: i) the final checkpoint enables fast restores for replay-prefix and replay-suffix queries, and the mid-run checkpoint enables 2x faster replay execution compared to the original run.

Speed-up through Selectivity & Scale-out

As discussed in the previous section, multiversion hindsight logging queries may be long-running tasks. In order to maintain interactive response times, it may be desirable to refine the query to be more selective (Section 5.5), or to allocate more resources (e.g., GPUs) and run the query in parallel. In some cases, a developer may be willing to pay the cost of generating extensive hindsight log messages from the inner loop of their training script. In this case each iteration of the inner loop must be replayed from checkpoint, and logs generated; a compute- and data-intensive task. In these cases, parallelism comes into play: because each loop iteration is based on a separate checkpoint, we can replay as many loop iterations as possible at once with embarrassing parallelism. We refer the reader to the experimental results in our earlier work measuring this effect [48]. To address the potential for long query times and the desire for selectivity or resource scaling, let's now examine the bottlenecks involved and the cost/performance trade-offs associated with resource use.

Bottleneck Analysis — length-1 range scan

Figure 5.11 (top) depicts the runtime of a range scan over a single epoch. Assuming *unbounded* resources (e.g. in an idealized cloud), every version would be replayed in its own machine. In this ideal scenario, a query's bottleneck would be the time it takes for a single Flor replay operation to finish. Replay-prefix, replay-suffix, and replay-validation would each finish in about 100 seconds or less (Figure 5.9). In contrast, the cost of a range scan, even for a single epoch, is higher. This operation requires stepping into the nested training loop, performing a forward pass over the neural network, and back-propagating gradients. Consequently, the range scan emerges as a bottleneck operator. In Figure 5.11 we evaluate its runtime on a single epoch; running times for more epochs or versions can be linearly

[2]Storing 100 GB of data in S3 costs approximately $2.30 a month.

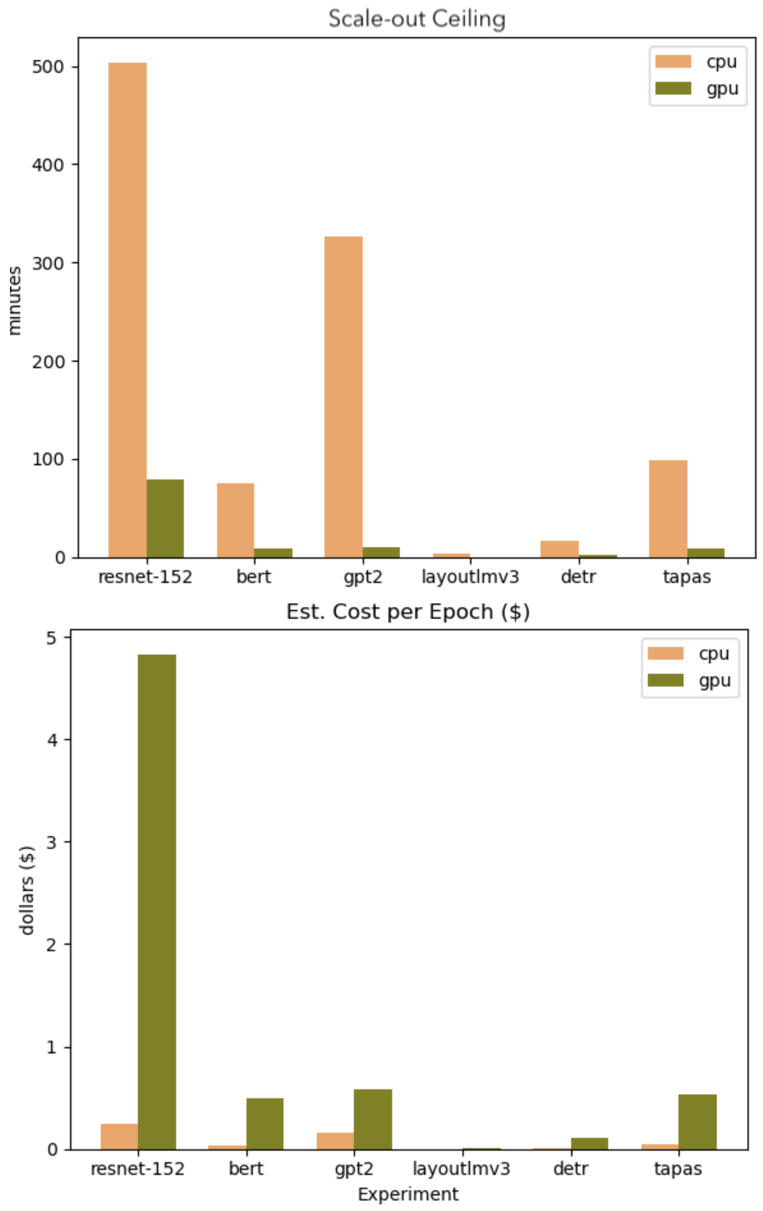


Figure 5.11: Comparison of runtime (top) vs rental cost (bot).

extrapolated. This is due to the ideal parallelism of replay-from-checkpoint (confirmed by prior studies [48]) and the embarrassing parallelism of replay across versions.

Cost and Performance Trade-offs

Our bottleneck analysis assumed an ideal of unbounded resources. While parallelism can significantly accelerate computation, it comes with increased resource consumption and financial costs. Figure 5.11 (bottom) compares the estimated cost per epoch across CPU and GPU platforms. Costs are based on AWS EC2 instance rental fees, with and without GPUs, factoring in execution time. GPUs provide order-of-magnitude superior computational speed but incur order-of-magnitude higher expenses. CPUs have the added benefit that they can be elastically allocated *en masse*. This can improve throughput across many versions or epochs, but as the bottleneck analysis indicates, there are limits to response times when range scans are used.

5.7 Related Work

ModelDB [197] is a system designed to store, version, and manage ML models effectively. Similar systems include Weights & Biases [17], VisTrails [15], and others [194, 21, 93]. ModelDB allows users to log complete model metadata, including hyper-parameters, data splits, evaluation metrics, and the final model file. Moreover, it provides the ability to query over this metadata, making it an effective tool for analysis and comparison of different ML models and experiment runs. However, the focus of ModelDB is mainly on managing the straightforward metadata of deployed models, and it does not provide any features for hindsight logging. ModelDB's focus on metadata is orthogonal to FlorDB's multiversion hindsight logging facilities; the two systems could be used in concert or individually.

MLFlow [212] is an open-source platform that helps manage the end-to-end machine learning lifecycle, including experimentation, reproducibility, and deployment. It provides functionalities to log parameters, versioned code, metrics, and output artifacts from each run and later query them. Its modular design allows it to be used with any existing ML library and development process. Similar systems include TFX [16], Airflow [11], Helix [206], and others [9, 54]. Like ModelDB, MLFlow has no support for hindsight logging or versioned log management. However, also like ModelDB, MLFlow and FlorDB can be fruitfully used together or separately.

FlorDB's query model builds on the notion of Acquisitional Query Processing (AQP) [116]. Traditional query processing assumes that data is pre-stored and ready for querying; AQP introduced the concept of efficiently acquiring data as a part of the query process. The idea is particularly useful for applications like sensor networks, where querying can be expensive in terms of energy or computational resources. FlorDB's multiversion hindsight logging also uses a form of AQP, where queries are not just made over pre-stored data, but also consider acquiring data through experiment replay. Unlike the original work on AQP, our acquisition task—i.e., multiversion hindsight logging—raises unique technical challenges of its own, forming the bulk of our work.

R3 [104], with its record-replay-retroaction mechanism, is primarily focused on database queries and transactions, capturing and replaying states at the database level. This approach is efficient for debugging with low-overhead recording and storage-efficient replay. While R3 aims to enhance the debugging and testing capabilities of database-backed applications, FlorDB addresses the specific needs of machine learning model development and analysis with its hindsight logging and query processing features.

5.8 Conclusion

This chapter introduces FlorDB, a system designed to address the unique challenges faced by machine learning engineers (MLEs) in managing the iterative, data-rich model development process. FlorDB's approach to multiversion hindsight logging, a record-replay technique, allows MLEs to add logging statements post-hoc, thereby enabling MLEs to "query the past." FlorDB's key contributions include a unified relational model for querying log results, automatic propagation of new logging statements across versions as part of acquisitional query processing, and a highly accurate empirical cost predictor for replay query refinement. These features streamline the ML experimentation process, enabling more efficient analysis and faster iteration. We evaluate the system's performance across various computer vision and NLP benchmarks, demonstrating the scalability of FlorDB across multiple versions.

Chapter 6

Context is All You Need: Closing the Loop in the Machine Learning Lifecycle

In the dynamic field of Artificial Intelligence (AI) and Machine Learning (ML), effective software engineering remains a significant challenge, particularly in the maintenance of long-running intelligent applications. Traditional software engineering systems often struggle with the complexities introduced by the integration of code, data, and configuration parameters into predictive models, leading to issues with context conservation throughout the AI/ML development lifecycle. This chapter addresses these challenges by proposing a streamlined ML lifecycle that minimizes manual interventions and automates key processes. Essential elements include robust data infrastructure, automated model training and evaluation, continuous integration and deployment (CI/CD), and effective monitoring and feedback loops.

In this chapter, we extend FlorDB to manage the complete "ABC" of context [62], making it a comprehensive context management system for the Machine Learning lifecycle. By managing a full range of contextual metadata, FlorDB enables efficient workflow and streamlined operations, enhancing the scalability and robustness of ML solutions. FlorDB simplifies the workflow by focusing on what developers need most: logging and dataframe queries. These fundamental tools are all that developers require to effectively manage and interact with their ML projects. By providing a robust logging mechanism and powerful dataframe query capabilities, FlorDB ensures that developers can easily track, retrieve, and analyze the context of their ML experiments and deployments without additional overhead.

Traditional context management often emphasizes a "metadata first" approach, which can introduce significant friction for developers. FlorDB reduces this friction through hindsight logging, allowing developers to add and refine metadata after the fact. This "metadata later" approach provides flexibility and ease of use, enabling developers to focus on building and improving models without being burdened by the immediate need to meticulously document every step up front. By supporting hindsight logging, FlorDB offers a more intuitive and developer-friendly way to manage contextual metadata, thereby enhancing productivity and lowering friction for developers. Through a series of use-cases, we demonstrate how FlorDB facilitates high-velocity validation loops, reliable experimentation, and improved efficiency for both individual developers and small teams. Our contributions highlight the importance of context management in closing the loop in the ML lifecycle.

6.1 Introduction

In the rapidly evolving field of Artificial Intelligence (AI) and Machine Learning (ML), the software engineering and maintenance of intelligent applications has emerged as an enduring challenge [168]. The blending of code, data, and configuration parameters into predictive models blurs traditional abstraction boundaries, straining conventional software engineering systems that rely on stricter separation of concerns and narrower problem scopes (e.g. "git is for source code, not data"). Furthermore, the complexity introduced by this conglomeration limits the ability of team members to effectively partition and manage distinct components of the project [177]. We argue that these challenges — and countless others [46] — primarily stem from either the absence of context or inadequate context conservation throughout the AI/ML application development lifecycle.

Streamlining the ML Lifecycle

The ML lifecycle involves several interconnected stages, from problem formulation and data collection to model deployment and maintenance. Optimizing this lifecycle requires addressing technical challenges at each stage and integrating processes and tools to reduce friction and speed up development. A streamlined ML lifecycle minimizes manual interventions, automates repetitive tasks, and ensures smooth data flow through the pipeline, resulting in faster iterations, more robust models, and increased value generation [177]. Key elements of this streamlined process include: i) building and maintaining robust data infrastructure; ii) automating model training and evaluation; iii) continuous integration and multi-stage deployment (CI/CD); and iv) monitoring and merging feedback loops. Effective streamlining enables organizations to maintain intelligent decision-based applications with minimal human intervention, improving efficiency, reducing operational overhead, and enhancing the scalability and robustness of ML solutions.

One of the persistent challenges in software engineering, particularly within the ML domain, is balancing agility with the rigor of "strong typing" or "metadata first" approaches. Agility requires flexibility and speed, allowing developers to iterate quickly and respond to changing requirements. In contrast, "strong typing" emphasizes structure and strict adherence to predefined metadata, which can slow down the development process but ensures consistency and reliability. FlorDB's approach substantially ameliorates this tension in two ways:

1. **Low-Friction Metadata via Log Statements and Dataframes** Our system allows developers to log and analyze metadata in a standard, open, low-friction manner.

Metadata can be captured naturally — as easily as adding a `print` statement — through log statements as part of the development workflow, without imposing significant overhead or disrupting the developers' focus on coding. Subsequently, these log statements can be read back directly as tabular data using the standard Python dataframe library, Pandas, without any need for data wrangling.

2. **Hindsight Logging for "Metadata on Demand"** Thanks to our hindsight logging mechanism, we eliminate the need for "metadata first" altogether. Developers can add or refine metadata post-hoc, on demand. This means that metadata can be captured and integrated long after the initial development phase, allowing for flexibility and adaptability without sacrificing the benefits of structured and comprehensive metadata management. By supporting hindsight logging, we offer a more intuitive and developer-friendly way to manage contextual metadata. This "metadata later" approach ensures that developers can focus on agility and rapid iteration while still maintaining the benefits of structured metadata management, reducing cognitive load and enhancing productivity.

FlorDB & The ABCs of Context

The ML lifecycle is characterized by numerous fast-changing components, where it is easy to lose the thread of essential metadata — what we term *context*. Context is metadata broadly conceived: it represents a comprehensive framework that captures the nature, origins, evolution, and functional significance of data and digital artifacts within an organization. Our conceptualization of context is drawn from the work of Hellerstein et al. (2017) [62], who proposed it as an extension of traditional database metadata. They introduced the "ABCs of Context" mnemonic[1], which we find particularly useful for understanding and navigating the complex landscape of ML metadata. This framework provides a structured approach to capturing and managing the rich tapestry of information that underpins real-world ML applications. The ABCs of Context are as follows:

A. **Application Context (What)**: This is core information that describes **what** raw bits an application sees and interprets for its use. Most comprehensively, this involves any information that *could be* logged; i.e., the value of arbitrary expressions at runtime.

B. **Build Context (How)**: This is information about **how** data is created and used: e.g., dependency management for distribution and building across different machines and by different people; provenance and lineage; routes, pathways, or branches in pipelines; and flow of control and data.

[1]The "B" in [62] stands for "Behavior"; we change it to "Build" because we think it more accurately describes the nature of metadata managed by FlorDB. Behavior implies human-human interactions, most of which occurs outside of Python, whereas Build context is contained in build files and can be probed by profiling tools and syscalls.

C. **Change Context (When)**: This is information about the version **history** of data, code, configuration parameters, and associated information, including changes over time to both structure and content.

Despite the critical nature of this metadata, modern ML applications lack a standard mechanism for managing these details comprehensively. FlorDB addresses this gap by capturing and managing extensive metadata in a familiar, unified API of *log statements* and *dataframe queries*, ensuring the essential context is consistently maintained throughout the ML lifecycle.

Goals and Contributions

In this chapter, our goals are to expand the management of context beyond continuous training [49] to span the entire workflow, emphasizing not only individual tasks (e.g. *training*) but entire pipelines and their regular execution. We demonstrate this through a three-part case study, starting with a single user-developer and expanding to a small team, highlighting the importance of context in streamlining development cycles and operational efficiency. Key goals and their corresponding contributions include:

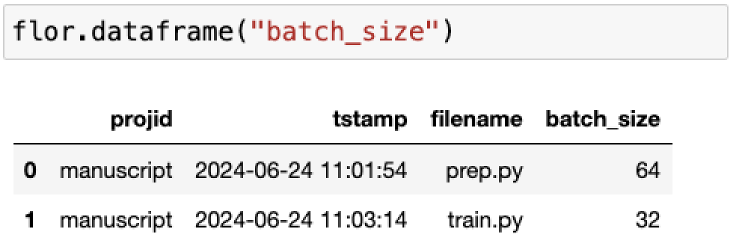
1. **Widening the Scope**: We aim to cover the whole workflow, or ML lifecycle, focusing not just on individual tasks or operations but entire pipelines and their regular execution. To support this, we have developed Build extensions for FlorDB that enable the management of the full ABCs of context. This includes mechanisms like `flor.log` (for writing) and `flor.dataframe` (for reading), which were initially designed for multiversion hindsight logging [49] but are now adapted to manage Build context in any workflow.

2. **Build System Agnosticism**: FlorDB is designed to function seamlessly with any build system (e.g. Make, MLFlow [211], etc.). FlorDB uses Python profiling tools such as `inspect` at import time to probe Build context automatically, such as the name and location of the hosting Git repository, and the name of the Python file being executed. FlorDB intercepts Build context at the Python-script level, making it operable with any build system of choice.

3. **API Stability Following Build Extensions**: The FlorDB APIs to manage change over time, i.e. with multiversion hindsight logging, remain backward-compatible and fully operational. We have extended FlorDB to passively capture and expose the Build context using the same APIs previously developed for managing Application and Change contexts. The capabilities of FlorDB have been extended to manage the full ABCs of context without affecting the API in Chapter 5.

4. **Simplified yet Powerful Abstraction**: A significant benefit of our open, standard approach is that we simplify and improve the abstraction of metadata, subsuming

```
# Makefile
prep:
    python prep.py

train: prep
    python train.py
```



```
flor.dataframe("batch_size")
```

	projid	tstamp	filename	batch_size
0	manuscript	2024-06-24 11:01:54	prep.py	64
1	manuscript	2024-06-24 11:03:14	train.py	32

Figure 6.1: Makefile and example Flor Dataframe after one call of `make train`. The logging of `batch_size` is unambiguous, given the `projid`, `tstamp`, and `filename` which are captured automatically and make up the relevant build and change context.

features from various bespoke ML metadata systems — such as feature stores, model registries, and label registries — into a unified and robust framework. By demonstrating effective methodology by which ML engineers can manage context, we illustrate how FlorDB can be used to build workflows and streamline repeated ML operations, closing the loop in the ML lifecycle.

6.2 Flow with FlorDB

In this section, we provide background information on Flor and FlorDB. Flor is a record-replay system specifically designed for model training [48]. Its functionality is divided into two main features: low-overhead checkpointing and low-latency replay from checkpoints. Flor's adaptive checkpointing runs in the background, minimizing computational resources spent on model training. At the same time, it ensures reduced latency during replay by leveraging memoization and parallelism through checkpoint-resume. As covered in detail in Chapter 4, Flor manages application context via hindsight logging, which allows effectively unbounded context to be queried or materialized post-hoc and on-demand.

FlorDB extends Flor to manage change over time by combining application context with change context through multiversion hindsight logging [49]. As covered in detail in Chapter 5, FlorDB employs a robust relational model, exposed via calls to `flor.dataframe`, that maps individual logging statements into columns in a pivoted view, making it easy for users to track changes over time. Building on FlorDB, we extend it in this chapter to add support for build context, completing the ABCs of context. The extended FlorDB provides a seamless and flexible framework for defining and executing complex and changing machine learning pipelines or workflows. FlorDB works with Make by default but was designed to be agnostic to the calling build system or workflow manager, and can be used with alternatives like Airflow, MLFlow, Slurm, or any workflow management system of choice.

Build Context & Unambiguous Logging

The extended capabilities of FlorDB are designed to complement dependency management systems like Make. While Make (or equivalent) handles the execution order of tasks based on defined dependencies, FlorDB extends this process by automatically capturing, tracking, and organizing the associated metadata throughout the execution of a workflow. Consider a simple workflow defined directly in a Makefile, which manages the execution of tasks such as `prep` and `train`. Each task executed by this Makefile triggers specific scripts that perform designated operations. During these tasks, FlorDB is activated with `import flor` by the executing process to capture crucial metadata. This includes:

- **Start and end times** of each task, providing a timeline of the workflow execution.
- **Dependency details**, which are managed by the developer in a build file (Makefile by default) and the build file is committed to git on every run.
- **Environmental parameters** such as the operating system, Python environment specifics, and hardware configurations, ensuring that all contextual factors affecting the execution are documented.
- **Project context**, capturing the directory of the Git repository at the current working directory to identify the project context clearly.
- **File-specific metadata**, utilizing the Python `inspect` module to log the filenames of scripts being executed, thereby linking the execution logs directly to the specific code being run.

This data is automatically captured by FlorDB at import time, and is included in every write of `flor.log(name, value)` so there's no ambiguity about where the log originated from. Suppose that both `prep` and `train` log the batch size. Because the `projid` and `filename` are captured and included in every log record, there will be no ambiguity when calling `flor.dataframe(batch_size)` (see Figure 6.1). By hooking into Python profiling tools, as well as tapping into system calls, FlorDB can gather comprehensive data without disrupting the workflow's normal operations.

FlorDB Extended API

The Flor API captures metadata about the executing file, so dataflow dependencies don't need to be re-stated in the API; the Makefile is sufficient. Because Flor profiles runtime metadata, including the name of the file being executed, it is agnostic to the workflow or dataflow management system used. Flor logging works seamlessly whether called by Make or Airflow, and switching between these systems does not require any refactoring.

The Flor API includes the following functionalities as presented in Garcia et al. (2023) [49]:

- `flor.log(name: str, value: T) -> T`
  Logs a given value with the specified name. Constructs a log record with the associated `projid`, `tstamp`, `filename`, and nesting dimensions defined by `flor.loop` (e.g. `epoch` or `page`). Useful for tracking variables or parameters during a run.

- `flor.arg(name: str, default: T) -> T`
  Reads a value from the command line if provided; otherwise, uses the default value. During replay, retrieves historical values. Good for setting configuration parameters for the scripts and allowing for quick adjustments via the command line.

- `flor.loop(name:str, vals:Iterable[T]) -> Iterable[T]`
  Functions as a Python generator that maintains global state between iterations. Useful for addressing `flor.log` records, "indexing" replay and coordinating checkpoints.

- `flor.checkpointing(kwargs: Dict) -> ContextManager`
  Acts as a Python context manager that defines the set of objects, such as model or optimizer, to be checkpointed adaptively — based on the size of the checkpoint and the loop iteration time [48] — at `flor.loop` iteration boundaries.

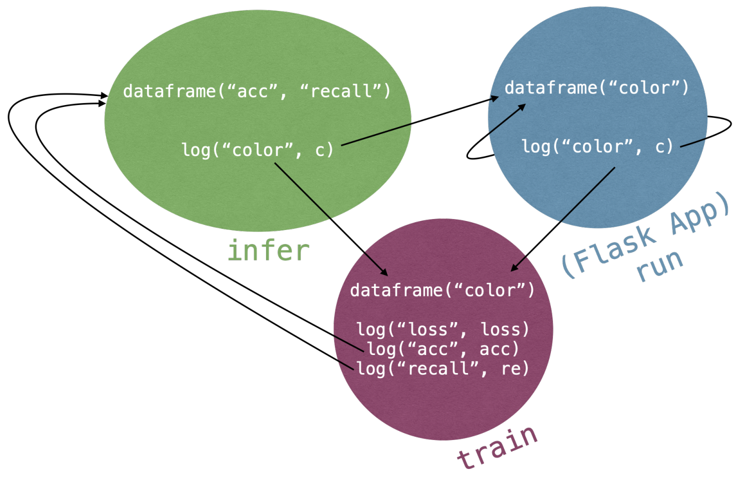
- `flor.dataframe(*args) -> pd.DataFrame`
  Produces a Pandas DataFrame of FlorDB log information with a column corresponding to each argument in `*args`. The DataFrame also contains "dimension" columns, such as the `projid`, `tstamp`, `filename` and so on. Each `arg` in `*args` corresponds to the `name` of a `flor.log` and is mapped to a column in the `flor.dataframe`. It is the default view in FlorDB for querying log data and selecting model checkpoints to load, using either Pandas or DataFrame-compatible SQL engines like DuckDB [150].

In this chapter, FlorDB further extends the API as follows:

- `flor.commit() -> None`
  An application-level transaction commit marker to support controlling the visibility of long-running processes, such as Flask applications serving a front-end that commits multiple transactions. On commit, FlorDB writes a log file, commits changes to git, and increments the `tstamp`. This method is invoked automatically (by `atexit`) at the end of a Python execution.

6.3 Closing the Loop in the ML Lifecycle

In a decoupled architecture of multiple applications and backend services, a unified view of overall context can serve as a narrow gateway — a single point of access for the basic information about data, metadata and their usage. We envision FlorDB streamlining



```
# Makefile

prep:
    python prep.py

infer: prep
    python infer.py

run: infer
    flask run

train: prep
    python train.py
```

Figure 6.2: Dataflow diagram and Makefile. `infer` uses `flor.dataframe("acc", "recall")` to selects the model checkpoint with highest recall (or a fallback model if no checkpoint exists) and uses it to segment documents, labeling pages with colors such that contiguous pages with the same color belong to the same segment. `run` launches a Flask web application to display the model predictions, overlaying a color layer over each page for human review. On confirmation, Flask logs the final color of each page. `train` uses the human-reviewed data (or feedback) to tune the model and improve its performance on future rounds of document segmentation. The process is repeated again and again, oscillating between calls to `make run` and `make train`. `flor.log(name, value)` manages writes, and `flor.dataframe(*args)` manages reads.

ML engineer's workflow by integrating fragmented metadata from scattered sources within `flor.dataframe`: a single, unified solution.

FlorDB promotes an agile approach that emphasizes tight-loop development. This approach can involve an iterative multi-phase development process (or workflow) in which an initial phase (e.g. `infer` in Figure 6.2) leverages inference pipelines to generate predictions based on existing models. These predictions are then presented to users in a subsequent phase (e.g. `run` in Figure 6.2), where human feedback is incorporated to refine and train subsequent models. This creates a continuous feedback loop where user-verified labels are ingested by *train* pipelines to create checkpoints for the *infer* pipelines. In Figure 6.2 we show metadata being written via `flor.log` statements, and queried via the `flor.dataframe` API.

While FlorDB supports generic Python pipelines, workflows, and build jobs, it is particularly helpful in managing closed loops, which are prevalent in production environments. Effective management of these closed loops is often challenging and can be a point of failure, making them a primary focus of our case study. The three cases presented next demonstrate instances of this agile development framework. By successfully managing context in the ML lifecycle, ML engineers can close the loop, integrating feedback from later stages at each step and providing each participant with a comprehensive view of the entire lifecycle.

6.4 Case Study

Next, we explore the progressive enhancement of document processing through the integration of AI features. The first case introduces the PDF Parser, a Flask-based web application designed for efficient PDF document processing, including tasks like splitting PDFs, extracting text, and preparing data for analysis using natural language processing (NLP) techniques. Subsequent cases build on this foundation by delving into collaboration. Case 2 examines the collaborative tool QUILT, which performs entity linking, emphasizing the importance of context sharing in collaborative interfaces. Case 3, titled "Merging the Composite," extends the PDF Parser's functionality by incorporating Named Entity Recognition (NER) capabilities from Case 2. This integration shows how functionalities can be shared across projects, and highlights the practical benefits of modular AI in streamlining complex document analysis tasks, particularly in legal and academic applications.

Case 1: Document Segmentation with PDF Parser

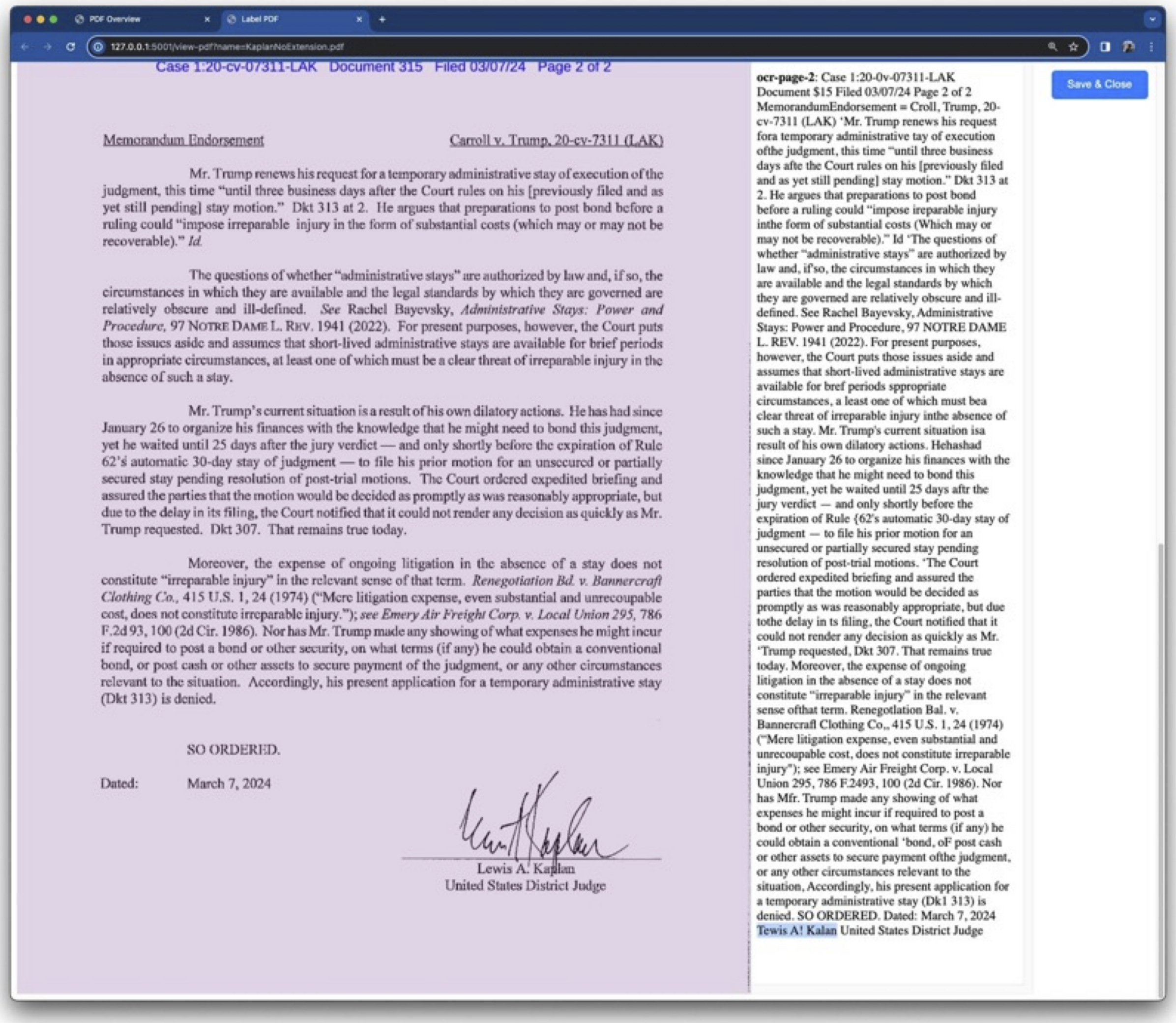
The PDF Parser project[2] is a Flask-based web application designed to simplify how users process and analyze PDF documents. This case explores how the PDF Parser works, highlighting its key features and practical use cases. Users can interact with the parser through a simple web interface, as shown in Figure 6.3. By understanding how the PDF Parser works and its potential applications, users can leverage its features to gain valuable insights from their PDF documents and streamline their workflows. We describe core components next. The focus of the case 1 is simplicity. By seeing how flor is used to implement a real document intelligence application, that this section will serve as a guide for others who are looking to get started with FlorDB.

PDF Extraction & Text Featurization

Once the PDF is converted into text and image formats, ensuring there is one document per page, the process of featurization begins (see lines 1-9 in Figure 6.4). This process typically involves the following steps:

- **Text Extraction**: The `pdf_demux.py` script is used to demultiplex each page of the PDF into separate text and image files. This is crucial for handling documents where text and visual data are intermixed. The text extraction component employs Optical Character Recognition (OCR) to convert any text found in images into machine-readable format.

[2] `https://github.com/ucbepic/pdf_parser`



Case 1:20-cv-07311-LAK Document 315 Filed 03/07/24 Page 2 of 2

Memorandum Endorsement Carroll v. Trump, 20-cv-7311 (LAK)

Mr. Trump renews his request for a temporary administrative stay of execution of the judgment, this time "until three business days after the Court rules on his [previously filed and as yet still pending] stay motion." Dkt 313 at 2. He argues that preparations to post bond before a ruling could "impose irreparable injury in the form of substantial costs (which may or may not be recoverable)." *Id.*

The questions of whether "administrative stays" are authorized by law and, if so, the circumstances in which they are available and the legal standards by which they are governed are relatively obscure and ill-defined. *See* Rachel Bayevsky, *Administrative Stays: Power and Procedure*, 97 NOTRE DAME L. REV. 1941 (2022). For present purposes, however, the Court puts those issues aside and assumes that short-lived administrative stays are available for brief periods in appropriate circumstances, at least one of which must be a clear threat of irreparable injury in the absence of such a stay.

Mr. Trump's current situation is a result of his own dilatory actions. He has had since January 26 to organize his finances with the knowledge that he might need to bond this judgment, yet he waited until 25 days after the jury verdict — and only shortly before the expiration of Rule 62's automatic 30-day stay of judgment — to file his prior motion for an unsecured or partially secured stay pending resolution of post-trial motions. The Court ordered expedited briefing and assured the parties that the motion would be decided as promptly as was reasonably appropriate, but due to the delay in its filing, the Court notified that it could not render any decision as quickly as Mr. Trump requested. Dkt 307. That remains true today.

Moreover, the expense of ongoing litigation in the absence of a stay does not constitute "irreparable injury" in the relevant sense of that term. *Renegotiation Bd. v. Bannercraft Clothing Co.*, 415 U.S. 1, 24 (1974) ("Mere litigation expense, even substantial and unrecoupable cost, does not constitute irreparable injury."); *see Emery Air Freight Corp. v. Local Union 295*, 786 F.2d 93, 100 (2d Cir. 1986). Nor has Mr. Trump made any showing of what expenses he might incur if required to post a bond or other security, on what terms (if any) he could obtain a conventional bond, or post cash or other assets to secure payment of the judgment, or any other circumstances relevant to the situation. Accordingly, his present application for a temporary administrative stay (Dkt 313) is denied.

SO ORDERED.

Dated: March 7, 2024

[illegible]
Lewis A. Kaplan
United States District Judge

ocr-page-2: Case 1:20-0v-07311-LAK Document $15 Filed 03/07/24 Page 2 of 2 MemorandumEndorsement = Croll, Trump, 20-cv-7311 (LAK) 'Mr. Trump renews his request fora temporary administrative tay of execution ofthe judgment, this time "until three business days afte the Court rules on his [previously filed and as yet still pending] stay motion." Dkt 313 at 2. He argues that preparations to post bond before a ruling could "impose ireparable injury inthe form of substantial costs (Which may or may not be recoverable)." Id 'The questions of whether "administrative stays" are authorized by law and, if'so, the circumstances in which they are available and the legal standards by which they are governed are relatively obscure and ill-defined. See Rachel Bayevsky, Administrative Stays: Power and Procedure, 97 NOTRE DAME L. REV. 1941 (2022). For present purposes, however, the Court puts those issues aside and assumes that short-lived administrative stays are available for bref periods sppropriate circumstances, a least one of which must bea clear threat of irreparable injury inthe absence of such a stay. Mr. Trump's current situation isa result of his own dilatory actions. Hehashad since January 26 to organize his finances with the knowledge that he might need to bond this judgment, yet he waited until 25 days aftr the jury verdict — and only shortly before the expiration of Rule {62's automatic 30-day stay of judgment — to file his prior motion for an unsecured or partially secured stay pending resolution of post-trial motions. 'The Court ordered expedited briefing and assured the parties that the motion would be decided as promptly as was reasonably appropriate, but due tothe delay in ts filing, the Court notified that it could not render any decision as quickly as Mr. 'Trump requested, Dkt 307. That remains true today. Moreover, the expense of ongoing litigation in the absence of a stay does not constitute "irreparable injury" in the relevant sense ofthat term. Renegotlation Bal. v. Bannercrafl Clothing Co,, 415 U.S. 1, 24 (1974) ("Mere litigation expense, even substantial and unrecoupable cost, does not constitute irreparable injury"); see Emery Air Freight Corp. v. Local Union 295, 786 F.2493, 100 (2d Cir. 1986). Nor has Mfr. Trump made any showing of what expenses he might incur if required to post a bond or other security, on what terms (if any) he could obtain a conventional 'bond, oF post cash or other assets to secure payment ofthe judgment, or any other circumstances relevant to the situation, Accordingly, his present application for a temporary administrative stay (Dk1 313) is denied. SO ORDERED. Dated: March 7, 2024 Tewis A! Kalan United States District Judge

Figure 6.3: Screenshot of the PDF Parser (Case 1). A scanned document appears on the left pane, it can be scrolled vertically from page to page. A colored layer is overlaid over each page for document segmentation: contiguous pages with the same color belong to the same segment. Extracted text and other features are displayed on the right page, grouped by page. The end-user may correct segmentation errors by clicking on the page overlay to cycle over colors; they may correct OCR errors by editing the text in the right pane directly.

- **Feature Engineering**: The `featurize.py` script takes the cleaned text and applies various natural language processing (NLP) techniques to extract meaningful features from the text (e.g. headers and page numbers, as shown in Figure 6.5). This might include tokenization, stemming, and the creation of n-grams. For images, feature en-

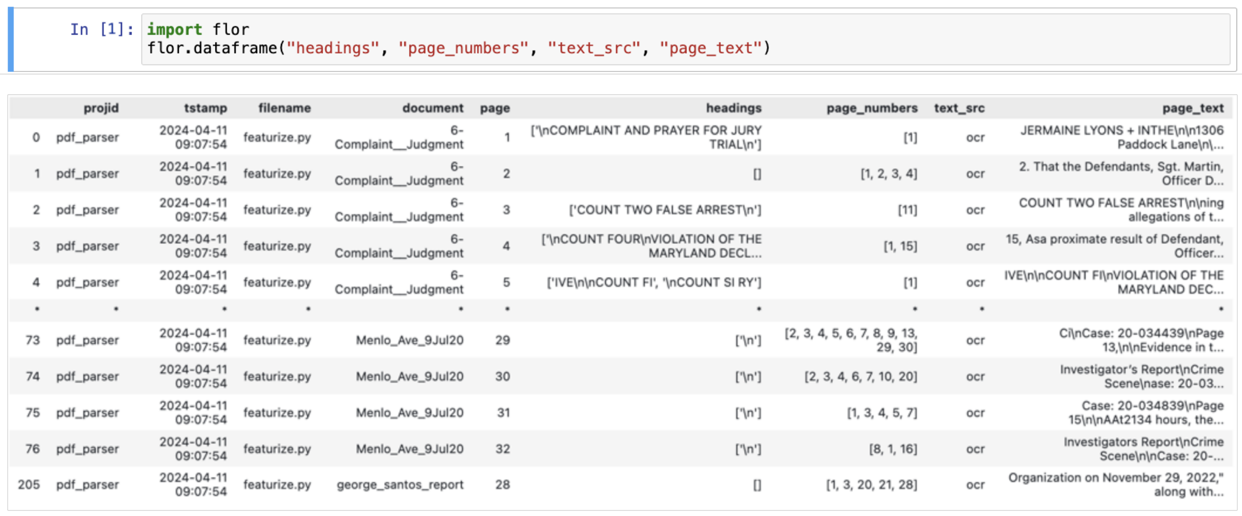
```
process_pdfs: $(PDFS) pdf_demux.py
    @echo "Processing PDF files..."
    @python pdf_demux.py
    @touch process_pdfs

featurize: process_pdfs featurize.py
    @echo "Featurizing Data..."
    @python featurize.py
    @touch featurize

train: featurize hand_label train.py
    @echo "Training..."
    @python train.py

model.pth: train export_ckpt.py
    @echo "Generating model..."
    @python export_ckpt.py

infer: model.pth infer.py
    @echo "Inferencing..."
    @python infer.py
    @touch infer

hand_label: label_by_hand.py
    @echo "Labeling by hand"
    @python label_by_hand.py
    @touch hand_label

run: featurize infer
    @echo "Starting Flask..."
    @flask run
```

Figure 6.4: Makefile of the PDF Parser. Workflow consists of *train* and *infer* pipelines. On *run*, PDF Parser extracts and featurizes documents and feeds them through the inference pipeline to display document segmentation plan to the end-user. Following any corrections and confirmation, the newly labeled data becomes training data for future rounds of training, improving future runs of the *infer* pipeline.

gineering could involve extracting color histograms, texture patterns, or edge statistics which are useful for computer vision models.

- **Vectorization**: The text features are then vectorized into a format suitable for machine learning models. This often involves transforming categorical data into numerical data through techniques like neural embedding, one-hot encoding or the use of TF-IDF (Term Frequency-Inverse Document Frequency) for text.

```
1  for doc_name in flor.loop("document", os.listdir(...)):
2      N = get_num_pages(doc_name)
3      for page in flor.loop("page", range(N)):
4          # text_src is "OCR" or "TXT"
5          text_src, page_text = read_page(doc_name, page)
6          flor.log("text_src", text_src)
7          flor.log("page_text", page_text)
8
9          # Run some featurization
10         headings, page_numbers = analyze_text(page_text)
11         flor.log("headings", headings)
12         flor.log("page_numbers", page_numbers)
```



```
In [1]: import flor
        flor.dataframe("headings", "page_numbers", "text_src", "page_text")
```

	projid	tstamp	filename	document	page	headings	page_numbers	text_src	page_text
0	pdf_parser	2024-04-11 09:07:54	featurize.py	6-Complaint__Judgment	1	['\nCOMPLAINT AND PRAYER FOR JURY TRIAL\n']	[1]	ocr	JERMAINE LYONS + INTHE\n\n1306 Paddock Lane\n\...
1	pdf_parser	2024-04-11 09:07:54	featurize.py	6-Complaint__Judgment	2	[]	[1, 2, 3, 4]	ocr	2. That the Defendants, Sgt. Martin, Officer D...
2	pdf_parser	2024-04-11 09:07:54	featurize.py	6-Complaint__Judgment	3	['COUNT TWO FALSE ARREST\n']	[11]	ocr	COUNT TWO FALSE ARREST\n\ning allegations of t...
3	pdf_parser	2024-04-11 09:07:54	featurize.py	6-Complaint__Judgment	4	['\nCOUNT FOUR\nVIOLATION OF THE MARYLAND DECL...	[1, 15]	ocr	15, Asa proximate result of Defendant, Officer...
4	pdf_parser	2024-04-11 09:07:54	featurize.py	6-Complaint__Judgment	5	['IVE\n\nCOUNT FI', '\nCOUNT SI RY']	[1]	ocr	IVE\n\nCOUNT FI\nVIOLATION OF THE MARYLAND DEC...
•	•	•	•	•	•	•	•	•	•
73	pdf_parser	2024-04-11 09:07:54	featurize.py	Menlo_Ave_9Jul20	29	['\n']	[2, 3, 4, 5, 6, 7, 8, 9, 13, 29, 30]	ocr	Ci\nCase: 20-034439\nPage 13,\n\nEvidence in t...
74	pdf_parser	2024-04-11 09:07:54	featurize.py	Menlo_Ave_9Jul20	30	['\n']	[2, 3, 4, 6, 7, 10, 20]	ocr	Investigator’s Report\nCrime Scene\nase: 20-03...
75	pdf_parser	2024-04-11 09:07:54	featurize.py	Menlo_Ave_9Jul20	31	['\n']	[1, 3, 4, 5, 7]	ocr	Case: 20-034839\nPage 15\n\nAAt2134 hours, the...
76	pdf_parser	2024-04-11 09:07:54	featurize.py	Menlo_Ave_9Jul20	32	['\n']	[8, 1, 16]	ocr	Investigators Report\nCrime Scene\n\nCase: 20-...
205	pdf_parser	2024-04-11 09:07:54	featurize.py	george_santos_report	28	[]	[1, 3, 20, 21, 28]	ocr	Organization on November 29, 2022," along with...

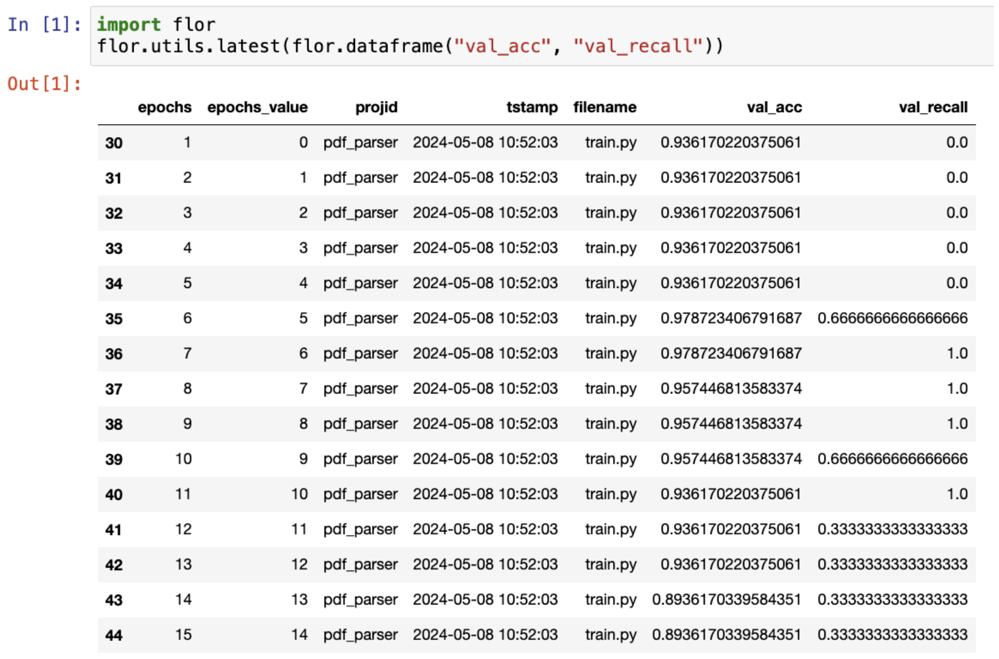
Figure 6.5: PDF Extraction and text featurization. Note that by logging features (e.g. lines 11-12), the developer has effectively populated a feature store (bottom dataframe).

This featurization process is essential for transforming raw PDF data into a structured form that is amenable to analysis and machine learning applications. The described methodology focuses on maximizing the information extracted from each page, ensuring that both textual and visual data contribute to the inferences made on the document.

Inference Pipeline

The inference pipeline automates the processing and analysis of images organized into document-specific folders. The pipeline, contained in `infer.py`, systematically processes each document's pages and images. For each image, it loads the image file, applies standard pre-processing steps such as resizing and normalization, and converts it into a format suit-

```
1  from train import model, transform
2  
3  cols = ["text_src", "headings"]
4  cols += ["page_numbers", "page_text"]
5  features = flor.dataframe(*cols)
6  
7  metrics = flor.dataframe("val_acc", "val_recall")
8  metrics  = flor.utils.latest(metrics)
9  best_ckpt = \
10   metrics[metrics.val_recall == metrics.val_recall.max()]
11 
12 ckpts = {'model': model}
13 with flor.loading(ckpts, best_ckpt):
14     for (doc_name, page), row in features:
15         tensor = transform(row[cols])
16         output = model(tensor)
17         logits, predicted = torch.max(output.data, 1)
18         p = int(predicted.item())
19         flor.log("first_page", 1 if page == 0 else p)
```

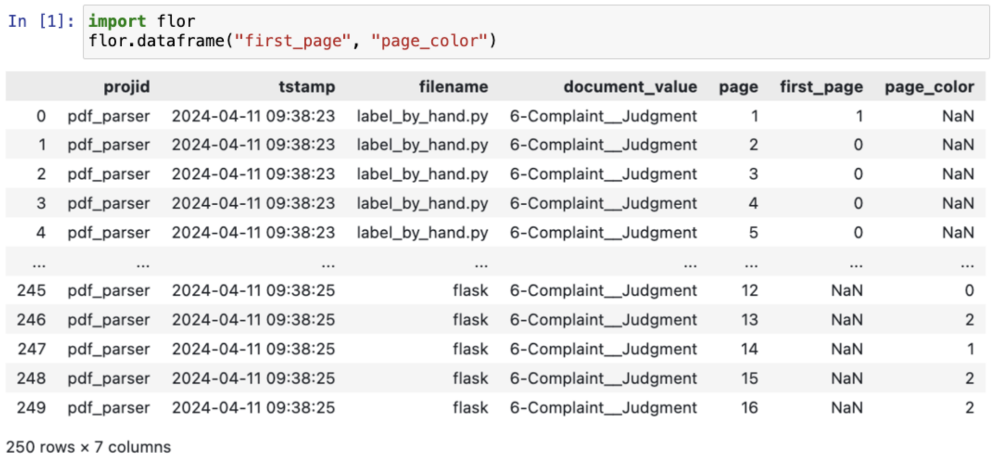


```
In [1]: import flor
        flor.utils.latest(flor.dataframe("val_acc", "val_recall"))
Out[1]:
```

	epochs	epochs_value	projid	tstamp	filename	val_acc	val_recall
30	1	0	pdf_parser	2024-05-08 10:52:03	train.py	0.936170220375061	0.0
31	2	1	pdf_parser	2024-05-08 10:52:03	train.py	0.936170220375061	0.0
32	3	2	pdf_parser	2024-05-08 10:52:03	train.py	0.936170220375061	0.0
33	4	3	pdf_parser	2024-05-08 10:52:03	train.py	0.936170220375061	0.0
34	5	4	pdf_parser	2024-05-08 10:52:03	train.py	0.936170220375061	0.0
35	6	5	pdf_parser	2024-05-08 10:52:03	train.py	0.978723406791687	0.6666666666666666
36	7	6	pdf_parser	2024-05-08 10:52:03	train.py	0.978723406791687	1.0
37	8	7	pdf_parser	2024-05-08 10:52:03	train.py	0.957446813583374	1.0
38	9	8	pdf_parser	2024-05-08 10:52:03	train.py	0.957446813583374	1.0
39	10	9	pdf_parser	2024-05-08 10:52:03	train.py	0.957446813583374	0.6666666666666666
40	11	10	pdf_parser	2024-05-08 10:52:03	train.py	0.936170220375061	1.0
41	12	11	pdf_parser	2024-05-08 10:52:03	train.py	0.936170220375061	0.3333333333333333
42	13	12	pdf_parser	2024-05-08 10:52:03	train.py	0.936170220375061	0.3333333333333333
43	14	13	pdf_parser	2024-05-08 10:52:03	train.py	0.8936170339584351	0.3333333333333333
44	15	14	pdf_parser	2024-05-08 10:52:03	train.py	0.8936170339584351	0.3333333333333333

Figure 6.6: Inference Pipeline. Developer queries a model registry (populated during training) to select the appropriate model for inference (lines 7-10, and bottom dataframe). Selected model is applied on features (lines 3-5) derived by earlier steps in the workflow. Note the versatility of FlorDB, how it can be used at once as a feature store and model registry.

```
 1  @app.route("/")
 2  def home():
 3      return flask.render_template("index.html")
 4
 5  def get_colors():
 6      infer = flor.dataframe("first_page", "page_color")
 7      infer = flor.utils.latest(
 8          infer[infer.document_value == pdf_names[-1]])
 9      if infer.page_color.isna().any():
10          assert infer.first_page.notna().all()
11          color = infer["first_page"].astype(int).cumsum()
12          infer["page_color"] = color - 1
13      return infer["page_color"].to_list()
14
15  @app.route("/save_colors", methods=["POST"])
16  def save_colors():
17      j = request.get_json()
18      colors = j.get("colors", [])
19      pdf_name = pdf_names.pop()
20      pdf_names.clear()
21      with flor.iteration("document", None, pdf_name):
22          for i in flor.loop("page", range(len(colors))):
23              flor.log("page_color", colors[i])
24      flor.commit()
25      return jsonify({"message": "Colors saved"}), 200
```



```
In [1]: import flor
        flor.dataframe("first_page", "page_color")
```

	projid	tstamp	filename	document_value	page	first_page	page_color
0	pdf_parser	2024-04-11 09:38:23	label_by_hand.py	6-Complaint__Judgment	1	1	NaN
1	pdf_parser	2024-04-11 09:38:23	label_by_hand.py	6-Complaint__Judgment	2	0	NaN
2	pdf_parser	2024-04-11 09:38:23	label_by_hand.py	6-Complaint__Judgment	3	0	NaN
3	pdf_parser	2024-04-11 09:38:23	label_by_hand.py	6-Complaint__Judgment	4	0	NaN
4	pdf_parser	2024-04-11 09:38:23	label_by_hand.py	6-Complaint__Judgment	5	0	NaN
...	...	...	...	...	...	...	...
245	pdf_parser	2024-04-11 09:38:25	flask	6-Complaint__Judgment	12	NaN	0
246	pdf_parser	2024-04-11 09:38:25	flask	6-Complaint__Judgment	13	NaN	2
247	pdf_parser	2024-04-11 09:38:25	flask	6-Complaint__Judgment	14	NaN	1
248	pdf_parser	2024-04-11 09:38:25	flask	6-Complaint__Judgment	15	NaN	2
249	pdf_parser	2024-04-11 09:38:25	flask	6-Complaint__Judgment	16	NaN	2

250 rows × 7 columns

Figure 6.7: Methods to read and write page overlay colors in Flask Web App. The query in line 6 is displayed in the bottom dataframe. Page overlays are drawn from column `first_page` or `page_color` depending on whether the label was sourced from crowd-workers (`filename == label_by_hand.py`) or the end-user (`filename == flask`). Note how FlorDB seamlessly manages build context (e.g. label origins).

able for input to the pre-trained model. The model then makes predictions on the images, logging the final inferences (Figure 6.6).

UI: Web Application on Flask

The main Flask script outlines the core functionalities of a web application designed for handling PDF documents and associated image files. It includes routes for displaying and manipulating PDFs and their converted image previews within a web interface. The core of the application is structured around Flask routes that handle web requests:

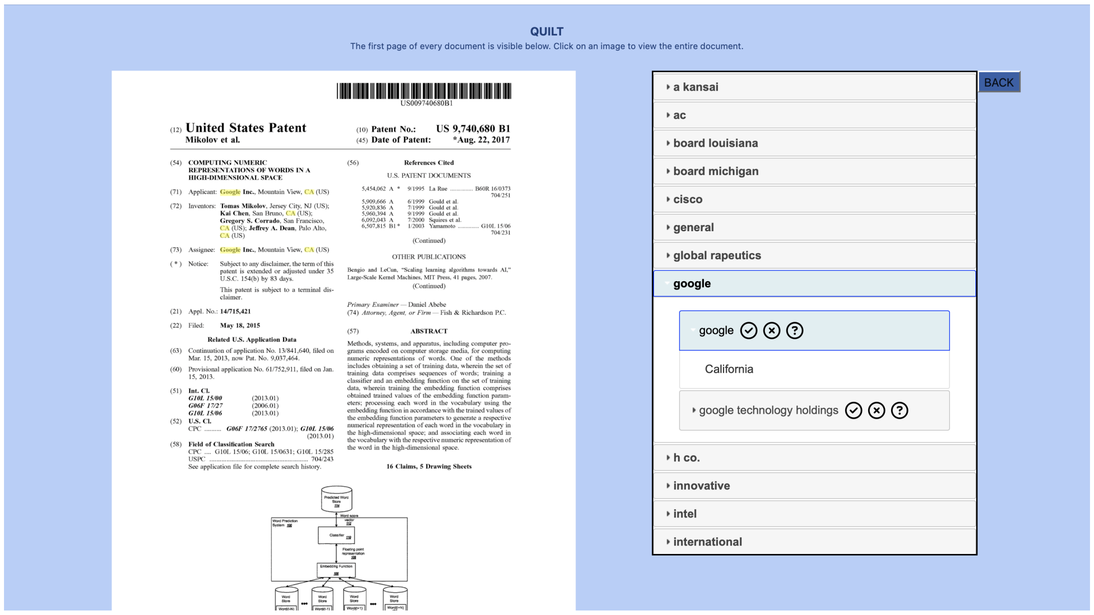
- The root route ("/") displays a home page which lists all the PDF files located in a specified directory. Each PDF file is represented with a preview image, and these images are listed on the webpage using rendered HTML templates.
- The "/view-pdf" route handles requests to view a specific PDF. Depending on user interactions and the file's existence, it can display the document in different modes such as labeled text or named entity recognition (NER) views. This route also handles dynamic user settings that can influence how the document is processed and displayed.
- The "/save_colors" route is a POST endpoint that processes user-submitted data concerning color settings associated with a PDF's pages (lines 15-25 in Figure 6.7). This route captures this data, logs it for tracking, and acknowledges the successful saving of data.

Helper functions such as `get_colors()` fetch color data associated with the pages of a document, integrating latest updates from a dataset (lines 5-13 and bottom dataframe in Figure 6.7). The developer may choose to display labels generated by the model (e.g. `first_page`) or labels entered manually by an expert end-user (e.g. `page_color`).

Training Pipeline

The training pipeline integrates several components of a typical machine learning workflow including data preparation, model definition, training, and validation, specifically customized to handle images extracted from PDF pages. The model used in this script is based on the ResNet-18 architecture, a popular convolutional neural network that is commonly used for image classification tasks. The final layer of the network is modified to suit a binary classification problem, i.e. whether the page in question is a first-page or not. To optimize performance, the script freezes the earlier layers of the model and only allows the final layer to be trained. Data for the model is processed using transformations that standardize and augment the image data, making it suitable for neural network processing. These transformations include resizing, cropping, converting images to tensors, and normalizing their pixel values.

Training and validation processes are conducted in a loop over a specified number of epochs. During each epoch, the model undergoes training and validation phases. In the training phase, the model learns from the training data by adjusting its parameters to minimize the loss function, which is calculated using a weighted cross-entropy loss to handle class imbalance. The validation phase evaluates the model's performance on unseen data, helping monitor its ability to generalize. Throughout the training and validation, performance metrics such as loss, accuracy, and recall are logged using FlorDB.



The first page of every document is visible below. Click on an image to view the entire document.

US009740680B1

(12) **United States Patent**
Mikolov et al.

(10) **Patent No.:** **US 9,740,680 B1**
(45) **Date of Patent:** ***Aug. 22, 2017**

(54) **COMPUTING NUMERIC REPRESENTATIONS OF WORDS IN A HIGH-DIMENSIONAL SPACE**

(71) Applicant: Google **Inc.**, Mountain View, CA (US)

(72) Inventors: **Tomas Mikolov**, Jersey City, NJ (US); **Kai Chen**, San Bruno, CA (US); **Gregory S. Corrado**, San Francisco, CA (US); **Jeffrey A. Dean**, Palo Alto, CA (US)

(73) Assignee: Google **Inc.**, Mountain View, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 83 days.

This patent is subject to a terminal disclaimer.

(21) Appl. No.: **14/715,421**

(22) Filed: **May 18, 2015**

Related U.S. Application Data

(63) Continuation of application No. 13/841,640, filed on Mar. 15, 2013, now Pat. No. 9,037,464.

(60) Provisional application No. 61/752,911, filed on Jan. 15, 2013.

(51) **Int. Cl.**
G10L 15/00 (2013.01)
G06F 17/27 (2006.01)
G10L 15/06 (2013.01)

(52) **U.S. Cl.**
CPC ***G06F 17/2765*** (2013.01); ***G10L 15/06*** (2013.01)

(58) **Field of Classification Search**
CPC G10L 15/06; G10L 15/0631; G10L 15/285
USPC 704/243
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

```
5,454,062 A *  9/1995 La Rue .............. B60R 16/0373
                                                704/251
5,909,666 A    6/1999 Gould et al.
5,920,836 A    7/1999 Gould et al.
5,960,394 A    9/1999 Gould et al.
6,092,043 A    7/2000 Squires et al.
6,507,815 B1 * 1/2003 Yamamoto ............. G10L 15/06
                                                704/231
```

(Continued)

OTHER PUBLICATIONS

Bengio and LeCun, "Scaling learning algorithms towards AI," Large-Scale Kernel Machines, MIT Press, 41 pages, 2007.

(Continued)

Primary Examiner — Daniel Abebe
(74) *Attorney, Agent, or Firm* — Fish & Richardson P.C.

(57) **ABSTRACT**

Methods, systems, and apparatus, including computer programs encoded on computer storage media, for computing numeric representations of words. One of the methods includes obtaining a set of training data, wherein the set of training data comprises sequences of words; training a classifier and an embedding function on the set of training data, wherein training the embedding function comprises obtained trained values of the embedding function parameters; processing each word in the vocabulary using the embedding function in accordance with the trained values of the embedding function parameters to generate a respective numerical representation of each word in the vocabulary in the high-dimensional space; and associating each word in the vocabulary with the respective numeric representation of the word in the high-dimensional space.

16 Claims, 5 Drawing Sheets

Figure 6.8: Screenshot of QUILT (Case 2). A scanned document is displayed on the left pane. Recognized named entities in that document are displayed in a list of dropdowns on the right pane. When the dropdown of a named entity is selected, the mentions of that entity in the document are highlighted, and the dropdown is expanded. The expanded dropdown shows a ranked list of known entities and their known attributes for entity linking.

Case 2: Entity Linking in a Collaborative Context

In the previous sections, we focused on the workflow of a single user interacting with a document intelligence system. However, real-world projects often involve multiple team members with diverse roles and expertise. In this section, we'll expand our exploration to a small team scenario, consisting of: i) the developer, primarily responsible for building and maintaining the technical infrastructure of the entity linking system, and ii) the expert-user, who brings specialized knowledge of the subject matter and provides valuable feedback on the system's accuracy and usefulness. To illustrate, our collaborators at the National Association of Criminal Defense Lawyers (NACDL) presented the following scenario:

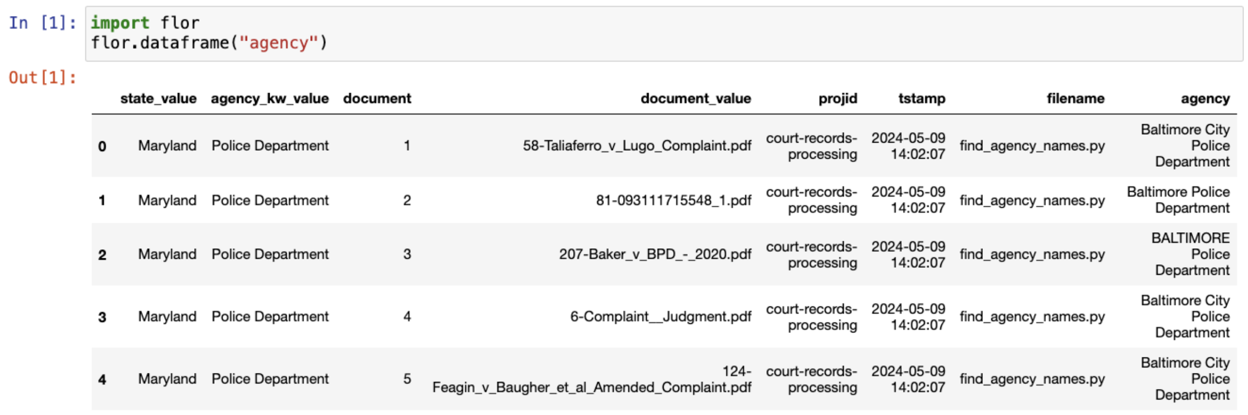
> A team of public defenders wants to track police officers involved in police misconduct cases. They have a collection of court case documents from the last ten

```
# Path Variables
OFF_DIR = Officer_Data
CASE_DIR = Case_File_Processor
AGENCY_DIR = $(CASE_DIR)/Find_Agency_Names

# Targets
case_file_processer:
    @echo "Extracting case files..."
    @cd $(CASE_DIR) && \
        python3 case_file_processer.py && cd ..
    @touch case_file_processer

find_agency_names: case_file_processer
    @echo "Query known agencies..."
    @cd $(AGENCY_DIR) && \
        python3 find_agency_names.py && cd ../../
    @touch find_agency_names

officer_matching: find_agency_names
    @echo "Matching officers to cases..."
    @cd $(OFF_DIR) && \
        python3 officer_matching.py && cd ..
    @touch officer_matching

match_ranking: officer_matching
    @echo "Sorting name matches..."
    @cd $(OFF_DIR) && \
        python3 rank_matches.py && cd ..
    @touch match_ranking

train: match_ranking
    @echo "Calibrating the ranking model..."
    @cd $(OFF_DIR) && \
        python3 train.py && cd ..
    @touch train

run_server: match_ranking
    @cd Server && flask run
```

Figure 6.9: Makefile of QUILT. Scanned documents pass through text extraction and named-entity resolution before being displayed on a web page. Known entities fuzzy-matching named-entities are queried from a production database. Known entities are ranked by similarity score (to the named-entity).

years and a database containing information about police officers, their affiliations, and dates of employment. The defenders aim to create a new dataset that

```
 1 agency_kw = {
 2     "Maryland": ["Police Department"],
 3     "New York": ["PD", "City of"],
 4     ...
 5 }
 6 page_text = flor.dataframe("page_text")
 7 if not page_text.empty:
 8     for s in flor.loop("state", agency_kw):
 9         for kw in flor.loop("agency_kw", agency_kw[s]):
10             documents = page_text.document_value.unique()
11             for doc_name in flor.loop("document", documents):
12                 c = page_text.document_value == doc_name
13                 all_pages = page_text[c]
14                 text = "\n".join(all_pages.page_text)
15                 searchRes = search(text, 2, kw)
16                 pd = checkDept(searchRes, kw)
17                 flor.log("agency", pd)
```



```
In [1]: import flor
        flor.dataframe("agency")
Out[1]:
```

	state_value	agency_kw_value	document	document_value	projid	tstamp	filename	agency
0	Maryland	Police Department	1	58-Taliaferro_v_Lugo_Complaint.pdf	court-records-processing	2024-05-09 14:02:07	find_agency_names.py	Baltimore City Police Department
1	Maryland	Police Department	2	81-093111715548_1.pdf	court-records-processing	2024-05-09 14:02:07	find_agency_names.py	Baltimore Police Department
2	Maryland	Police Department	3	207-Baker_v_BPD_-_2020.pdf	court-records-processing	2024-05-09 14:02:07	find_agency_names.py	BALTIMORE Police Department
3	Maryland	Police Department	4	6-Complaint__Judgment.pdf	court-records-processing	2024-05-09 14:02:07	find_agency_names.py	Baltimore City Police Department
4	Maryland	Police Department	5	124-Feagin_v_Baugher_et_al_Amended_Complaint.pdf	court-records-processing	2024-05-09 14:02:07	find_agency_names.py	Baltimore City Police Department

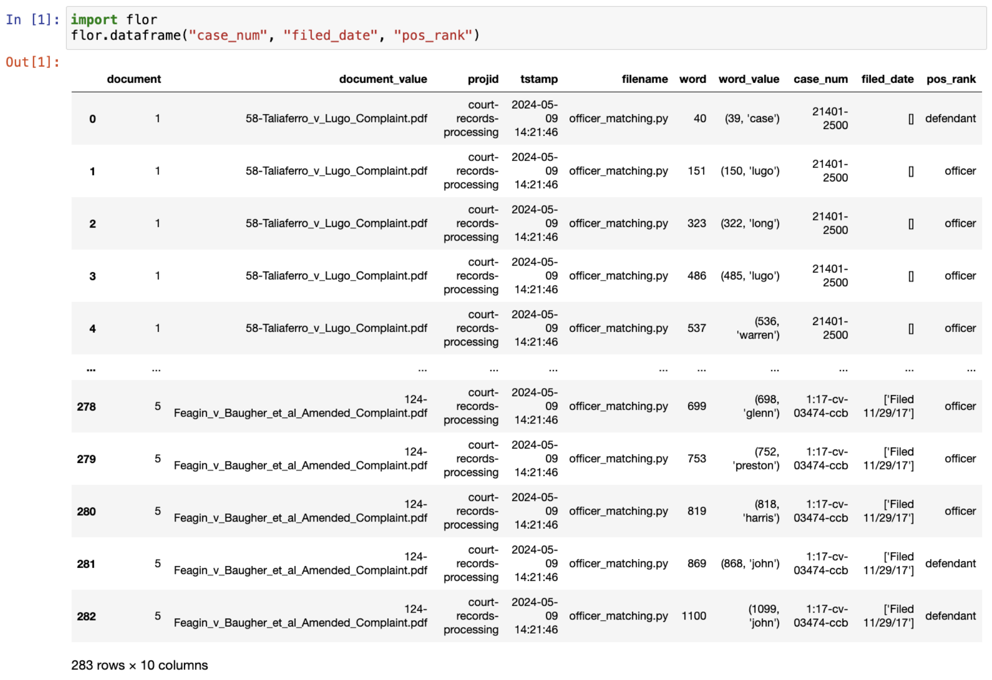
Figure 6.10: Find Agency Names for a group of documents. Note how much *build* (`projid`, `filename`) and *change* metadata (`tstamp`) is included in the `"agency"` dataframe (bottom of this figure), and how the dimensions of the dataframe correspond to `flor` loops (e.g. `"state"` and `"document"` in lines 8-11).

connects these documents with the officers mentioned in them.

In response, we developed QUILT[3], an entity linking tool, through collaboration with NACDL domain experts (Figure 6.8). Over two years, we met weekly with stakeholders from NACDL and a journalist from KQED to discuss needs, identify requirements, iterate on designs, and take feedback. We focused on supporting teams where one or a few members

[3]`https://github.com/ucbepic/court-records-processing`

```python
1  officer_roster = get_officer_roster()
2  page_text = flor.dataframe("page_text")
3
4  docs = page_text.document_value.unique()
5  for doc_name in flor.loop("document", docs):
6      pred = page_text.document_value == doc_name
7      file_text = "\n".join(page_text[pred].page_text)
8      flor.log("case_num", find_case_num(file_text))
9      filed_date, event_range = find_date(file_text)
10     flor.log("filed_date", filed_date)
11     flor.log("event_range", event_range)
12
13     for word in flor.loop("word", file_text.split()):
14         ...
15         if last_name(word, officer_roster.name):
16             flor.log("pos_rank", pos_rank)
17             full_name = first + middle + last
18             flor.log("name_inferred", full_name)
```



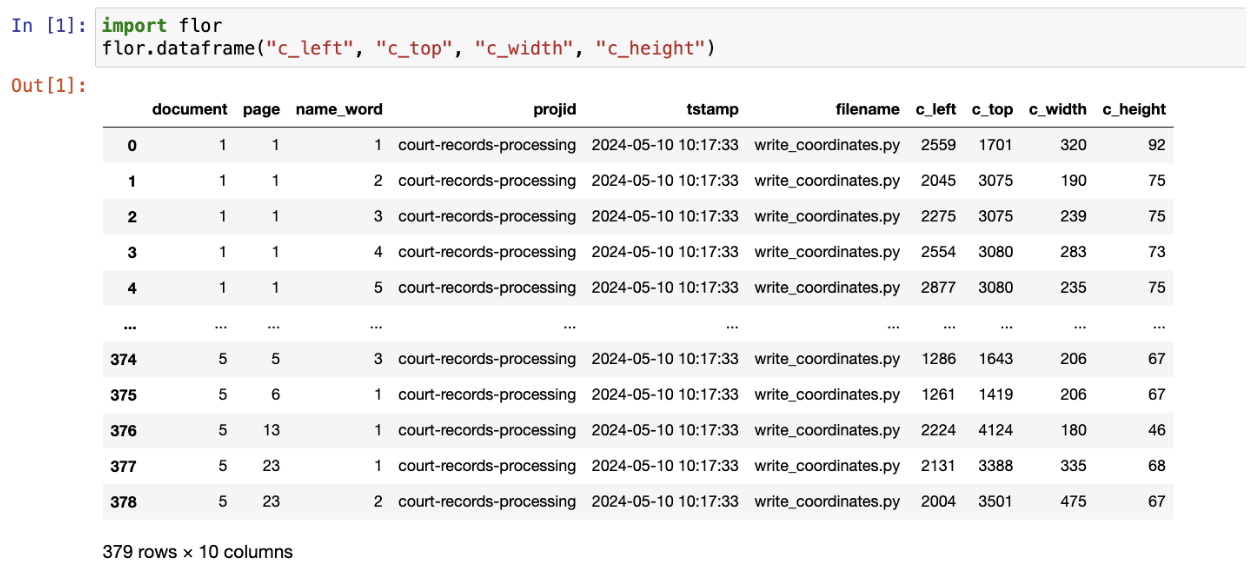
```
In [1]: import flor
        flor.dataframe("case_num", "filed_date", "pos_rank")
Out[1]:
```

	document	document_value	projid	tstamp	filename	word	word_value	case_num	filed_date	pos_rank
0	1	58-Taliaferro_v_Lugo_Complaint.pdf	court-records-processing	2024-05-09 14:21:46	officer_matching.py	40	(39, 'case')	21401-2500	[]	defendant
1	1	58-Taliaferro_v_Lugo_Complaint.pdf	court-records-processing	2024-05-09 14:21:46	officer_matching.py	151	(150, 'lugo')	21401-2500	[]	officer
2	1	58-Taliaferro_v_Lugo_Complaint.pdf	court-records-processing	2024-05-09 14:21:46	officer_matching.py	323	(322, 'long')	21401-2500	[]	officer
3	1	58-Taliaferro_v_Lugo_Complaint.pdf	court-records-processing	2024-05-09 14:21:46	officer_matching.py	486	(485, 'lugo')	21401-2500	[]	officer
4	1	58-Taliaferro_v_Lugo_Complaint.pdf	court-records-processing	2024-05-09 14:21:46	officer_matching.py	537	(536, 'warren')	21401-2500	[]	officer
...	...	...	...	...	...	...	...	...	...	...
278	5	124-Feagin_v_Baugher_et_al_Amended_Complaint.pdf	court-records-processing	2024-05-09 14:21:46	officer_matching.py	699	(698, 'glenn')	1:17-cv-03474-ccb	['Filed 11/29/17']	officer
279	5	124-Feagin_v_Baugher_et_al_Amended_Complaint.pdf	court-records-processing	2024-05-09 14:21:46	officer_matching.py	753	(752, 'preston')	1:17-cv-03474-ccb	['Filed 11/29/17']	officer
280	5	124-Feagin_v_Baugher_et_al_Amended_Complaint.pdf	court-records-processing	2024-05-09 14:21:46	officer_matching.py	819	(818, 'harris')	1:17-cv-03474-ccb	['Filed 11/29/17']	officer
281	5	124-Feagin_v_Baugher_et_al_Amended_Complaint.pdf	court-records-processing	2024-05-09 14:21:46	officer_matching.py	869	(868, 'john')	1:17-cv-03474-ccb	['Filed 11/29/17']	defendant
282	5	124-Feagin_v_Baugher_et_al_Amended_Complaint.pdf	court-records-processing	2024-05-09 14:21:46	officer_matching.py	1100	(1099, 'john')	1:17-cv-03474-ccb	['Filed 11/29/17']	defendant

283 rows × 10 columns

Figure 6.11: Fuzzy-matching named-entities on known entities. Note the visibility added by the bottom dataframe. We can see in the first row that "defendant case" (fourth column from the right) is not person-entity but a case-file, and so further cleaning is needed before logging a hit (lines 13-18).

```
1  @app.route("/")
2  def home():
3      return flask.render_template("index.html")
4
5  @app.route("/coordinates")
6  def coordinates():
7      # assert app.static_folder is not None
8      # coord_file = open(
9      #     os.path.join(app.static_folder,
10     #       "Output_Page_Coordinates.csv"), "r")
11     # reader = csv.reader(coord_file)
12     # data = list(reader)
13     # coord_file.close()
14
15     #  Get the coordinates from flor.dataframe
16     data = flor.dataframe("c_left", "c_top", "c_width", "c_height")
17     return flask.jsonify(data)
```



```
In [1]: import flor
        flor.dataframe("c_left", "c_top", "c_width", "c_height")
```

Out[1]:

	document	page	name_word	projid	tstamp	filename	c_left	c_top	c_width	c_height
0	1	1	1	court-records-processing	2024-05-10 10:17:33	write_coordinates.py	2559	1701	320	92
1	1	1	2	court-records-processing	2024-05-10 10:17:33	write_coordinates.py	2045	3075	190	75
2	1	1	3	court-records-processing	2024-05-10 10:17:33	write_coordinates.py	2275	3075	239	75
3	1	1	4	court-records-processing	2024-05-10 10:17:33	write_coordinates.py	2554	3080	283	73
4	1	1	5	court-records-processing	2024-05-10 10:17:33	write_coordinates.py	2877	3080	235	75
...	...	...	...	...	...	...	...	...	...	...
374	5	5	3	court-records-processing	2024-05-10 10:17:33	write_coordinates.py	1286	1643	206	67
375	5	6	1	court-records-processing	2024-05-10 10:17:33	write_coordinates.py	1261	1419	206	67
376	5	13	1	court-records-processing	2024-05-10 10:17:33	write_coordinates.py	2224	4124	180	46
377	5	23	1	court-records-processing	2024-05-10 10:17:33	write_coordinates.py	2131	3388	335	68
378	5	23	2	court-records-processing	2024-05-10 10:17:33	write_coordinates.py	2004	3501	475	67

379 rows × 10 columns

Figure 6.12: Reading NER coordinates for Web App text highlighting. Note how a single `flor` dataframe (line 16) obviates an entire flask-method worth of ad-hoc data processing code (lines 7-13). Also, note the schema is not readable in the commented version, but is clearly visible in the `flor` dataframe call.

are developers, while the others are non-programmers who primarily review documents. The project's build dependencies are defined in Figure 6.9.

PDF Extraction & Text Featurization

This subsection delves into the techniques and tools employed to extract meaningful text from PDFs and convert it into numerical representations, paving the way for downstream entity linking tasks. Once the PDF is converted into images, the next critical step is to extract textual content from these images. This extraction is performed using the `Tesseract` OCR engine, which interprets the pixel data of the images to produce corresponding text. The script enhances OCR accuracy by pre-processing the images into a format more amenable to OCR, such as converting them to grayscale or re-scaling. Additionally, common OCR errors, such as incorrectly hyphenated text at line breaks, are programmatically corrected to ensure the integrity and continuity of the extracted text. The extracted text is not just collected; it is also transformed by `flor` dataframes into a structured format that can be readily used for data analysis (e.g. "agency" in Figure 6.10) or fed into machine learning algorithms. This involves cleaning and organizing the text into a coherent format free of common scanning errors and artifacts. In addition to raw text extraction, the script logs various attributes of the text, such as its coordinates and size on the page, which can be important for tasks that require understanding the document layout or identifying key sections within the text.

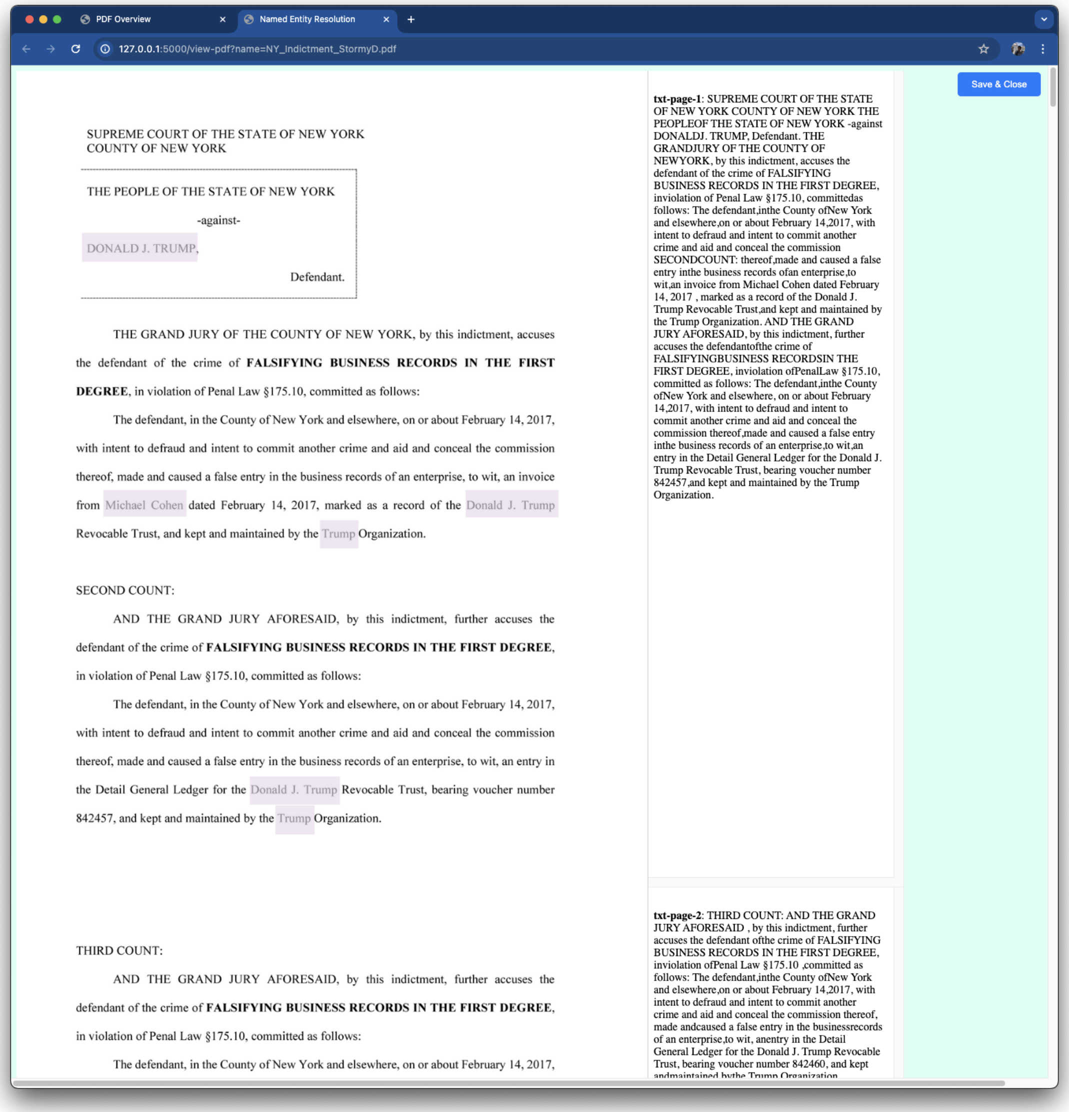
Fuzzy Schema Integration

A core component of this process is fuzzy matching. Given the prevalence of OCR errors and the common occurrence of variations in name spellings, fuzzy matching employs algorithms like Levenshtein distance or Jaro-Winkler distance to quantify the similarity between strings. This enables the identification of potential matches that might otherwise be missed due to minor discrepancies. Once potential matches are identified, the system proceeds to rank them based on a similarity score (Figure 6.11). This score takes into account not only the degree of similarity between strings but also contextual factors, such as the frequency of occurrence of the entities within the documents and their semantic relevance to the surrounding text. The result is a ranked list of potential matches, aiding users in quickly identifying the most relevant known entities for linking.

UI: Web Application on Flask

QUILT uses a Flask-based web application, designed to provide an intuitive and interactive environment for exploring and validating entity recognition and matching results. The web application seamlessly overlays predictions directly over the displayed PDF documents. Recognized named entities are visually highlighted within the documents themselves, while also being presented as a bulleted list in a separate pane. Users can click on names in this list, dynamically updating the PDF view to highlight the selected entity and its mentions, enhancing contextual understanding (Figure 6.12).

Selecting a name from the list reveals a ranked dropdown menu of candidate matches, with the most likely match positioned at the top. The interface offers options to confirm, tentatively accept ("maybe"), or reject matches. Each decision triggers a FlorDB transaction,



SUPREME COURT OF THE STATE OF NEW YORK
COUNTY OF NEW YORK

THE PEOPLE OF THE STATE OF NEW YORK

-against-

DONALD J. TRUMP,

Defendant.

THE GRAND JURY OF THE COUNTY OF NEW YORK, by this indictment, accuses the defendant of the crime of **FALSIFYING BUSINESS RECORDS IN THE FIRST DEGREE**, in violation of Penal Law §175.10, committed as follows:

The defendant, in the County of New York and elsewhere, on or about February 14, 2017, with intent to defraud and intent to commit another crime and aid and conceal the commission thereof, made and caused a false entry in the business records of an enterprise, to wit, an invoice from Michael Cohen dated February 14, 2017, marked as a record of the Donald J. Trump Revocable Trust, and kept and maintained by the Trump Organization.

SECOND COUNT:

AND THE GRAND JURY AFORESAID, by this indictment, further accuses the defendant of the crime of **FALSIFYING BUSINESS RECORDS IN THE FIRST DEGREE**, in violation of Penal Law §175.10, committed as follows:

The defendant, in the County of New York and elsewhere, on or about February 14, 2017, with intent to defraud and intent to commit another crime and aid and conceal the commission thereof, made and caused a false entry in the business records of an enterprise, to wit, an entry in the Detail General Ledger for the Donald J. Trump Revocable Trust, bearing voucher number 842457, and kept and maintained by the Trump Organization.

THIRD COUNT:

AND THE GRAND JURY AFORESAID, by this indictment, further accuses the defendant of the crime of **FALSIFYING BUSINESS RECORDS IN THE FIRST DEGREE**, in violation of Penal Law §175.10, committed as follows:

The defendant, in the County of New York and elsewhere, on or about February 14, 2017,

txt-page-1: SUPREME COURT OF THE STATE OF NEW YORK COUNTY OF NEW YORK THE PEOPLEOF THE STATE OF NEW YORK -against DONALDJ. TRUMP, Defendant. THE GRANDJURY OF THE COUNTY OF NEWYORK, by this indictment, accuses the defendant of the crime of FALSIFYING BUSINESS RECORDS IN THE FIRST DEGREE, inviolation of Penal Law §175.10, committedas follows: The defendant,inthe County ofNew York and elsewhere,on or about February 14,2017, with intent to defraud and intent to commit another crime and aid and conceal the commission SECONDCOUNT: thereof,made and caused a false entry inthe business records ofan enterprise,to wit,an invoice from Michael Cohen dated February 14, 2017 , marked as a record of the Donald J. Trump Revocable Trust,and kept and maintained by the Trump Organization. AND THE GRAND JURY AFORESAID, by this indictment, further accuses the defendantofthe crime of FALSIFYINGBUSINESS RECORDSIN THE FIRST DEGREE, inviolation ofPenalLaw §175.10, committed as follows: The defendant,inthe County ofNew York and elsewhere, on or about February 14,2017, with intent to defraud and intent to commit another crime and aid and conceal the commission thereof,made and caused a false entry inthe business records of an enterprise,to wit,an entry in the Detail General Ledger for the Donald J. Trump Revocable Trust, bearing voucher number 842457,and kept and maintained by the Trump Organization.

txt-page-2: THIRD COUNT: AND THE GRAND JURY AFORESAID , by this indictment, further accuses the defendant ofthe crime of FALSIFYING BUSINESS RECORDS IN THE FIRST DEGREE, inviolation ofPenal Law §175.10 ,committed as follows: The defendant,inthe County ofNew York and elsewhere,on or about February 14,2017, with intent to defraud and intent to commit another crime and aid and conceal the commission thereof, made andcaused a false entry in the businessrecords of an enterprise,to wit, anentry in the Detail General Ledger for the Donald J. Trump Revocable Trust, bearing voucher number 842460, and kept andmaintained bythe Trump Organization.

Figure 6.13: Screenshot of the PDF Parser using NER from QUILT (Case 3). Named-entities are highlighted in purple. Extracted text is displayed on the right pane.

logging the user's choice in the system's backend for potential further analysis. As shown in Figure 6.12, backend logic has been optimized by replacing standard filesystem API calls with `flor.dataframe` queries. This approach significantly simplifies data handling, reduces repetitive database logic, and enhances system efficiency and maintainability, particularly in environments that require rapid prototyping and frequent updates.

```
 1 process_pdfs: $(PDFS) pdf_demux.py
 2     @echo "Processing PDF files..."
 3     @python pdf_demux.py
 4     @touch process_pdfs
 5
 6 featurize: process_pdfs featurize.py
 7     @echo "Featurizing Data..."
 8     @python featurize.py
 9     @touch featurize
10
11 ner_parse:
12     @if [ -f ~/.flor/court-records-processing.db ]; then \
13         mv ~/.flor/court-records-processing.db ~/.flor/court-records-processing.db.bak; \
14     fi
15     @if [ -f ~/.flor/pdf_parser.db ]; then \
16         mv ~/.flor/pdf_parser.db ~/.flor/court-records-processing.db; \
17     fi
18     @if [ -d ../court_records ]; then \
19         mv ../court_records ../court_records.bak; \
20     fi
21     @ln -sf $(realpath app/static/private/pdfs) ../court_records
22
23     @cd ../court-records-processing && \
24         (git checkout flor.pdf_parser$(GIT_COMMIT) || \
25         git checkout -b flor.pdf_parser$(GIT_COMMIT)) && \
26         make case_file_processer
27
28     @rm -f ../court_records
29     @if [ -d ../court_records.bak ]; then \
30         mv ../court_records.bak ../court_records; \
31     fi
32     @mv ~/.flor/court-records-processing.db ~/.flor/pdf_parser.db
33     @if [ -f ~/.flor/court-records-processing.db.bak ]; then \
34         mv ~/.flor/court-records-processing.db.bak ~/.flor/court-records-processing.db; \
35     fi
36     @touch ner_parse
37
38 run: featurize ner_parse
39     @echo "Starting Flask..."
40     @flask run
```

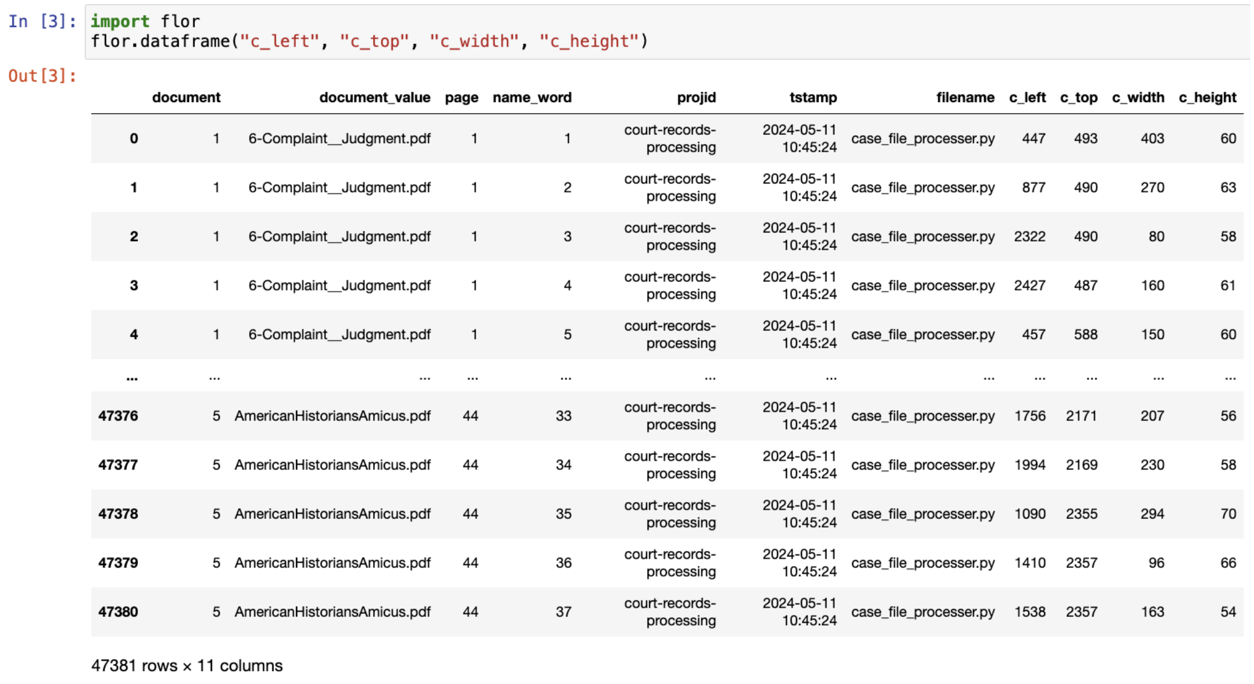
Figure 6.14: The PDF Parser Makefile is modified to set up and cleanup context, calling out to QUILT for NER parsing (lines 11-22).

Case 3: Merging the Composite

In this case, we extend the capabilities of the PDF Parser, originally designed for document segmentation and basic text analysis, by integrating Named Entity Recognition (NER) capabilities from the QUILT system (Figure 6.13). The merge of both cases is managed through modifications to the Makefile of the PDF Parser (Figure 6.14), which controls the file and database operations necessary to leverage NER capabilities. This setup ensures that the PDF Parser can operate seamlessly with the advanced tools developed in Case 2 without extensive manipulation.

As shown in the Makefile snippet (Figure 6.14), the data processing workflow is adapted

```python
@app.route("/")
def home():
    return flask.render_template("index.html")

...

def get_coordinates():
    data = flor.dataframe("c_left", "c_top", "c_width", "c_height")
    return jsonify(data)
```



```
In [3]: import flor
        flor.dataframe("c_left", "c_top", "c_width", "c_height")
Out[3]:
```

	document	document_value	page	name_word	projid	tstamp	filename	c_left	c_top	c_width	c_height
0	1	6-Complaint__Judgment.pdf	1	1	court-records-processing	2024-05-11 10:45:24	case_file_processer.py	447	493	403	60
1	1	6-Complaint__Judgment.pdf	1	2	court-records-processing	2024-05-11 10:45:24	case_file_processer.py	877	490	270	63
2	1	6-Complaint__Judgment.pdf	1	3	court-records-processing	2024-05-11 10:45:24	case_file_processer.py	2322	490	80	58
3	1	6-Complaint__Judgment.pdf	1	4	court-records-processing	2024-05-11 10:45:24	case_file_processer.py	2427	487	160	61
4	1	6-Complaint__Judgment.pdf	1	5	court-records-processing	2024-05-11 10:45:24	case_file_processer.py	457	588	150	60
...	...	...	...	...	...	...	...	...	...	...	...
47376	5	AmericanHistoriansAmicus.pdf	44	33	court-records-processing	2024-05-11 10:45:24	case_file_processer.py	1756	2171	207	56
47377	5	AmericanHistoriansAmicus.pdf	44	34	court-records-processing	2024-05-11 10:45:24	case_file_processer.py	1994	2169	230	58
47378	5	AmericanHistoriansAmicus.pdf	44	35	court-records-processing	2024-05-11 10:45:24	case_file_processer.py	1090	2355	294	70
47379	5	AmericanHistoriansAmicus.pdf	44	36	court-records-processing	2024-05-11 10:45:24	case_file_processer.py	1410	2357	96	66
47380	5	AmericanHistoriansAmicus.pdf	44	37	court-records-processing	2024-05-11 10:45:24	case_file_processer.py	1538	2357	163	54

47381 rows × 11 columns

Figure 6.15: PDF Parser can now get the coordinates for named-entities in its documents, e.g. `document_value == "AmericanHistoriansAmicus.pdf"`.

to include NER by calling the document preparation and processing stages in QUILT. Via the `featurize` build target, the PDF documents are first converted into a text format suitable for NER. Then, during a build of `ner_parse`, the NER models from the QUILT system are applied to the text data to detect and categorize named entities such as people, locations, and dates.

UI: Web Application on Flask

The Flask-based web application serves as a critical interface for the PDF Parser, particularly enhanced to handle Named Entity Recognition (NER) results effectively. With the incorporation of NER capabilities, the application not only displays the processed documents but also enriches them with interactive annotations of identified entities (four right-most columns in Figure 6.15).

6.5 Discussion

In this section, we evaluate FlorDB in the context of modern MLOps principles and best practices. We begin by examining how FlorDB embodies the 3Vs of MLOps as articulated by Shankar & Garcia et al. (2024), demonstrating its effectiveness in enabling velocity, visibility, and versioning throughout the machine learning lifecycle [177]. We then discuss how its design draws inspiration from the Ground data context service and Postel's law of robustness, allowing it to flexibly ingest and organize metadata from diverse systems. Through this analysis, we highlight the system's strengths as a comprehensive platform for streamlining and optimizing MLOps workflows.

MLOps Validation Criteria

Next, we evaluate the characteristics of FlorDB through the lens of the 3Vs of MLOps, as defined by Shankar & Garcia et al. (2024):

- **Velocity**: FlorDB significantly enhances the speed of development, testing, and deployment of machine learning models. By streamlining the integration of new data and models into the existing workflow, FlorDB allows for rapid iteration and quick adaptation to new requirements.
- **Visibility**: The system provides comprehensive logging and monitoring capabilities, which increase the transparency of the ML lifecycle. In the three use-cases, stakeholders were able to track each stage of the model development process, from initial data ingestion to final deployment. This visibility not only facilitated easier debugging and optimization but also enhanced the understanding of model behavior and performance across different operational contexts.
- **Versioning**: FlorDB relies on version control to manage data, models, and code changes. It supports detailed tracking of changes, allowing teams to revert to previous versions and understand the evolution of models over time.

By adhering to these principles, FlorDB addresses critical aspects of the MLOps lifecycle, making it a versatile and powerful tool for organizations aiming to leverage machine learning efficiently and effectively.

Postel's Law of Robustness

FlorDB has been fundamentally informed by the Ground data context service [62]. A key design principle we adopted from Ground is Postel's law of robustness, which can be summarized as:

> Be conservative in what you do, be liberal in what you accept from others.

As discussed earlier, the machine learning lifecycle involves a complex interplay of data, code, and hyper-parameters, each stage utilizing a diverse ecosystem of domain-specific systems with their own APIs. The ability to accept and process this information, regardless of its source, and organize it into a user-friendly format is crucial. We implement this principle through a logging metaphor for writing data and a dataframe metaphor for reading it. Essentially, FlorDB is as easy to write to as NoSQL and as easy to read from as SQL. The various components of contextual information are funneled into what we call a Flor Dataframe. This dataframe acts as a common layer, or narrow gateway, for organizing metadata from interleaved systems and disparate parts, making it both accessible and manageable for the user.

6.6 Related Work

In the realm of AI and ML, managing the lifecycle of machine learning models and their associated data is crucial for maintaining efficient and reproducible workflows. Several systems have been developed to address different aspects of this challenge, each with its unique approach and focus. FlorDB builds upon these ideas, offering a comprehensive context management system that not only logs information but also integrates it into a broader context for streamlined ML lifecycle management.

Experiment Tracking and Version Control: Systems like MLFlow [211], DVC (Data Version Control) [78], and Weights & Biases [17] focus on managing experiments and ensuring reproducibility. MLFlow provides tools for tracking experiments, packaging code into reproducible runs, and sharing and deploying models. DVC manages large files, datasets, machine learning models, and metrics alongside code, emphasizing reproducibility and collaboration. Both systems are crucial for tracking the evolution of models and data, and they can be used alongside FlorDB, but they primarily concentrate on experiment management without deeply integrating hindsight logging (and by extension data context) capabilities.

Model Management and Lineage: ModelDB, Mistique, and Pachyderm emphasize version control and data lineage [193, 194, 77]. ModelDB tracks the lineage of models, capturing relationships between models, their training data, and the code used to produce them. Pachyderm combines data versioning, data pipelines, and lineage on Kubernetes, providing a scalable environment for building reproducible data pipelines. These systems focus on managing the provenance and evolution of models and data, offering a way to query and visualize their history over time. The focus of both systems is more on artifacts and less on process: for example, ModelDB does not automatically version code, and has no way of recovering missing data.

End-to-End ML Workflows: AWS SageMaker and Kubeflow provide comprehensive solutions for building, training, and deploying ML models [105, 30]. SageMaker offers tools for data labeling, model training, and hosting, integrating seamlessly with other AWS services to support scalable ML workflows. Kubeflow, designed to run on Kubernetes, supports the orchestration of complex ML pipelines and integrates with a wide range of ML frameworks. Both platforms emphasize scalability and operational efficiency but primarily focus on the deployment and operational aspects of ML workflows. Either system can be used by FlorDB as a drop-in replacement to Make, the default build system. Other systems in this space include Helix [207] and Motion [174].

Visualization and Monitoring: TensorBoard [53] is a visualization toolkit for TensorFlow that allows users to track and visualize metrics, graphs, and other aspects of machine learning experiments. Because FlorDB does not natively ship with visualization capabilities, surfacing instead the context via Flor Dataframes, TensorBoard is a great option for visualizing FlorDB data.

FlorDB distinguishes itself by addressing the need for comprehensive context management across the entire ML lifecycle. By focusing on the ABCs of context—Application, Build, and Change contexts—FlorDB captures and integrates metadata that spans the full spectrum of data and model management. This includes not only tracking the execution of code and data flow but also managing dependencies, version history, and changes over time. In comparison to these systems, FlorDB's unique contribution ensures that context is preserved and accessible across different stages and by different people across the ML lifecycle, facilitating reproducibility and collaborative development. Furthermore, FlorDB's architecture supports integration with various build and workflow management tools, providing a flexible and extensible solution for managing complex ML workflows.

6.7 Conclusion

In this chapter, we have presented an extended version of FlorDB, a system that extends the Flor family of projects to support the management of context in the machine learning lifecycle. By capturing and integrating comprehensive metadata across the dimensions of Application, Build, and Change (the ABCs of context), FlorDB provides a unified solution for managing the complex and fast-evolving landscape of AI/ML development.

Through a series of cases, we have demonstrated how FlorDB is used to streamline the end-to-end ML lifecycle, from a single user-developer scenario to more complex collaborative environments. These cases highlight the system's ability to facilitate high-velocity development cycles, reliable experimentation, and efficient collaboration by preserving and sharing essential context across team members and stages of the ML workflow. In evaluating FlorDB against the 3Vs of MLOps — Velocity, Visibility, and Versioning — we have shown how the system embodies these principles to optimize the ML lifecycle. FlorDB's design, informed by the Ground data context service allows it to flexibly ingest and organize metadata from heterogeneous sources, serving as a narrow gateway for accessing essential context.

In conclusion, FlorDB represents a significant step forward in managing the complexity of AI/ML development by providing a unified solution for capturing, integrating, and leveraging essential context across the entire ML lifecycle. By managing the ABCs of context FlorDB empowers organizations to accelerate their ML initiatives while ensuring reproducibility, collaboration, and continuous improvement. As such, it has the potential to become an indispensable tool in the MLOps ecosystem, helping teams navigate the challenges and opportunities of AI/ML at scale.

Chapter 7

Conclusion

In conclusion, FlorDB represents a significant advancement in managing the complex and dynamic context inherent in the machine learning lifecycle. By comprehensively capturing and organizing the ABCs of context — Application, Build, and Change — FlorDB empowers machine learning engineers with increased transparency, traceability and control over their ML projects. Through innovations like multiversion hindsight logging, and their integration with build tools and relational data models, FlorDB transforms the scattered and often lost metadata of the ML lifecycle into a structured, queryable dataframe. This unified interface acts as a narrow gateway, channeling disparate information streams into actionable insights. The impact of this holistic context management is significant. It enhances collaboration, enables faster iteration, and critically, closes the feedback loop between model deployment and training. By providing a unified solution for capturing, preserving, and leveraging context, FlorDB empowers ML engineers to work more efficiently, collaboratively, and effectively. As the field of machine learning continues to evolve, the importance of context management will only grow, and FlorDB is poised to play a pivotal role in shaping the future of MLOps. By making FlorDB open-source and readily available, we aim to foster a community-driven approach to further innovation and improvement in the field of context management for machine learning.

7.1 Looking Back

The state of play today for ML context management reflects the complex, multi-faceted nature of modern machine learning workflows. What we observe is a hodgepodge of proprietary systems, each addressing a specific aspect of the ML lifecycle: data catalogs, build systems, feature stores, model registries, label managers, and more. While these tools individually serve important functions, their disaggregated nature has led to a critical challenge in the field. An unfortunate consequence of this fragmented landscape is the lack of a standard mechanism to assemble a collective understanding of the origin, scope, and usage of the data that ML applications manage. Despite the critical nature of this metadata, modern ML applications have struggled to manage these details comprehensively. This gap has significant implications for transparency, reproducibility, and efficiency in ML workflows.

Flor began as a tool for debugging long-running training scripts, but with the introduction of multiversion support it became evident that the applicability was far broader. The key innovation that emerged from this evolution was FlorDB's approach to context capture and management. By providing a familiar *lingua franca* for managing metadata through its logging and dataframe API, FlorDB offers a unified interface that cuts across the disparate tools and systems in the ML ecosystem. This approach transforms the scattered and often lost metadata of the ML lifecycle into a structured, queryable dataframe, providing a single point of reference for the basic information about data and its usage.

Perhaps most significantly, FlorDB's core technology of hindsight logging addresses a longstanding tension in ML workflows: the conflict between agile exploration and well-governed metadata capture. This innovative approach allows context to be captured post-hoc, and even by different individuals than those who build the initial models and pipelines. It effectively cuts the Gordian knot that has long challenged ML engineers, allowing them to work quickly and efficiently while still maintaining long-term organization and ensuring essential context is always captured.

In our vision laid out earlier, Flor streamlines ML engineers' workflows by integrating fragmented metadata from scattered systems within a single, unified, lightweight solution. By providing this narrow gateway — a unified interface for capturing, preserving, and leveraging context — FlorDB empowers ML engineers to work more efficiently, collaboratively, and effectively, all while maintaining the crucial context that underpins their work. As we look back on the development of FlorDB, we see not just a tool, but a paradigm shift in how we approach context management in machine learning. It represents a move from fragmented, ad-hoc solutions to a holistic, integrated "metadata later" approach that recognizes the central role of context in the ML lifecycle, and allows for it to be collected post-hoc.

7.2 Next Steps

As we look to the future of FlorDB, our focus will be on increasing adoption and building a robust open-source community. We plan to develop comprehensive documentation, create case studies, and engage with both academic and industry partners to promote FlorDB's use in diverse AI/ML use-cases. Establishing forums for knowledge sharing and organizing regular events will be crucial in fostering collaboration among users and contributors.

Continuous development will remain a priority, with regular updates based on community feedback and emerging ML trends. As machine learning evolves, so too will the challenges in context management. By maintaining an open, community-driven approach, we're committed to adapting FlorDB to meet the changing needs of ML practitioners and researchers. We're excited to see how FlorDB will shape the future of MLOps and context management in the years to come.

Bibliography

[1] Martín Abadi et al. “Tensorflow: Large-scale machine learning on heterogeneous distributed systems”. In: *arXiv preprint arXiv:1603.04467* (2016).

[2] Deepak Agarwal et al. “LASER: A Scalable Response Prediction Platform for Online Advertising”. In: *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*. WSDM ’14. New York, New York, USA: ACM, 2014, pp. 173–182. ISBN: 978-1-4503-2351-2. DOI: `10.1145/2556195.2556252`. URL: `http://doi.acm.org/10.1145/2556195.2556252`.

[3] Leonel Aguilar et al. “Ease.ML: A Lifecycle Management System for MLDev and MLOps”. In: *Conference on Innovative Data Systems Research (CIDR 2021)*. Jan. 2021. URL: `https://www.microsoft.com/en-us/research/publication/ease-ml-a-lifecycle-management-system-for-mldev-and-mlops/`.

[4] Sridhar Alla and Suman Kalyan Adari. “What is mlops?” In: *Beginning MLOps with MLFlow*. Springer, 2021, pp. 79–124.

[5] Gautam Altekar and Ion Stoica. “ODR: output-deterministic replay for multicore debugging”. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM. 2009, pp. 193–206.

[6] Saleema Amershi et al. “Software Engineering for Machine Learning: A Case Study”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. May 2019, pp. 291–300. DOI: `10.1109/ICSE-SEIP.2019.00042`.

[7] Dario Amodei et al. “Concrete problems in AI safety”. In: *arXiv preprint arXiv:1606.06565* (2016).

[8] Davide Ancona et al. “RPython: a step towards reconciling dynamically and statically typed OO languages”. In: *Proceedings of the 2007 symposium on Dynamic languages*. 2007, pp. 53–64.

[9] Robin Anil et al. “Apache Mahout: Machine learning on distributed dataflow systems”. In: *Journal of Machine Learning Research* 21.127 (2020), pp. 1–6.

[10] Anonymous. *ML Reproducibility Systems: Status and Research Agenda*. 2021. URL: `https://openreview.net/forum?id=v-6XBItNld2`.

[11] *Apache Airflow.* `https://airflow.apache.org/`. Accessed: 2024-02-06. Apache Software Foundation, 2016.

[12] Sotiris Apostolakis et al. "Perspective: A sensible approach to speculative automatic parallelization". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems.* 2020, pp. 351–367.

[13] John Aycock. "Aggressive type inference". In: *language* 1050 (2000), p. 18.

[14] Amitabha Banerjee et al. "Challenges and Experiences with {MLOps} for Performance Diagnostics in {Hybrid-Cloud} Enterprise Software Deployments". In: *2020 USENIX Conference on Operational Machine Learning (OpML 20).* 2020.

[15] Louis Bavoil et al. "Vistrails: Enabling interactive multiple-view visualizations". In: *Proceedings of the IEEE Visualization 2005 (VIS '05).* IEEE. 2005, pp. 135–142.

[16] Denis Baylor et al. "Tfx: A tensorflow-based production-scale machine learning platform". In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* ACM. 2017, pp. 1387–1395.

[17] Lukas Biewald. *Tracking with Weights and Biases www.wandb.com/.* 2020. URL: `https://www.wandb.com/`.

[18] Catherine Billington et al. "A Machine Learning Model Helps Process Interviewer Comments in Computer-assisted Personal Interview Instruments: A Case Study". In: *Field Methods* (2022), p. 1525822X221107053.

[19] Mike Brachmann et al. "Data Debugging and Exploration with Vizier". In: *Proceedings of the 2019 International Conference on Management of Data.* SIGMOD '19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 1877–1880. ISBN: 9781450356435. DOI: `10.1145/3299869.3320246`. URL: `https://doi.org/10.1145/3299869.3320246`.

[20] Eric Breck et al. "Data Validation for Machine Learning". In: *Proceedings of SysML.* 2019. URL: `https://mlsys.org/Conferences/2019/doc/2019/167.pdf`.

[21] Eric Breck et al. "Data Validation for Machine Learning." In: *MLSys.* 2019.

[22] Steven P Callahan et al. "VisTrails: visualization meets data management". In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data.* 2006, pp. 745–747.

[23] Nicolas Carion et al. *End-to-End Object Detection with Transformers.* 2020. arXiv: `2005.12872 [cs.CV]`.

[24] Pete Chapman et al. "The CRISP-DM user guide". In: *4th CRISP-DM SIG Workshop in Brussels in March.* Vol. 1999. sn. 1999.

[25] Sarah Chasins and Rastislav Bodik. "Skip blocks: reusing execution history to accelerate web scripts". In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (2017), pp. 1–28.

[26] Amit Chavan et al. "Towards a unified query language for provenance and versioning". In: *7th {USENIX} Workshop on the Theory and Practice of Provenance (TaPP 15)*. 2015.

[27] Rada Chirkova and Jun Yang. "Materialized views". In: *Databases* 4.4 (2011), pp. 295–405.

[28] Ji Young Cho and Eun-Hee Lee. "Reducing confusion about grounded theory and qualitative content analysis: Similarities and differences." In: *Qualitative report* 19.32 (2014).

[29] David Cohen, Mikael Lindvall, and Patricia Costa. "An introduction to agile methods." In: *Adv. Comput.* 62.03 (2004), pp. 1–66.

[30] Kubeflow Community. *Kubeflow: A Composable, Portable, Scalable ML Platform.* `https://www.kubeflow.org/`. Accessed: 2024-05-31. 2021.

[31] Daniel Crankshaw et al. "Clipper: A {Low-Latency} Online Prediction Serving System". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 2017, pp. 613–627.

[32] Daniel Crankshaw et al. "Clipper: A Low-Latency Online Prediction Serving System". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 613–627. ISBN: 978-1-931971-37-9. URL: `https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw`.

[33] Daniel Crankshaw et al. "The Missing Piece in Complex Analytics: Low Latency, Scalable Model Management and Serving with Velox". In: *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. 2015. URL: `http://cidrdb.org/cidr2015/Papers/CIDR15_Paper19u.pdf`.

[34] John W Creswell and Cheryl N Poth. *Qualitative inquiry and research design: Choosing among five approaches*. Sage publications, 2016.

[35] Weidong Cui et al. "REPT: Reverse Debugging of Failures in Deployed Software". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 17–32.

[36] Rishit Dagli and Ali Mustufa Shaikh. *CPPE-5: Medical Personal Protective Equipment Dataset*. 2021. arXiv: `2112.09569 [cs.CV]`.

[37] Anusha Dandamudi. "Fast Low-Overhead Logging Extending Time". MA thesis. EECS Department, University of California, Berkeley, May 2021. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-117.html`.

[38] 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*. 2008, pp. 1345–1350.

[39] Jeffrey Dean et al. “Large scale distributed deep networks”. In: *Advances in neural information processing systems*. 2012, pp. 1223–1231.

[40] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *CoRR* abs/1810.04805 (2018). arXiv: `1810.04805`. URL: `http://arxiv.org/abs/1810.04805`.

[41] Christof Ebert et al. “DevOps”. In: *Ieee Software* 33.3 (2016), pp. 94–100.

[42] Mihail Eric. *MLOps is a mess but that’s to be expected.* URL: `https://www.mihaileric.com/posts/mlops-is-a-mess/`.

[43] Jean-Rémy Falleri et al. “Fine-grained and accurate source code differencing”. In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 2014, pp. 313–324.

[44] Asbjørn Følstad, Cecilie Bertinussen Nordheim, and Cato Alexander Bjørkli. “What makes users trust a chatbot for customer service? An exploratory interview study”. In: *International conference on internet science*. Springer. 2018, pp. 194–208.

[45] Wikimedia Foundation. *Wikimedia Downloads*. URL: `https://dumps.wikimedia.org`.

[46] Rolando Garcia et al. “Context: The missing piece in the machine learning lifecycle”. In: *CMI*. 2018.

[47] Rolando Garcia et al. “Hindsight Logging for Model Training”. In: *VLDB*. 2021.

[48] Rolando Garcia et al. “Hindsight logging for model training”. In: *Proceedings of the VLDB Endowment* 14.4 (2020), pp. 682–693.

[49] Rolando Garcia et al. “Multiversion Hindsight Logging for Continuous Training”. In: *arXiv preprint arXiv:2310.07898* (2023).

[50] Satvik Garg et al. “On Continuous Integration / Continuous Delivery for Automated Deployment of Machine Learning Models using MLOps”. In: *2021 IEEE Fourth International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*. 2021, pp. 25–28. DOI: `10.1109/AIKE52691.2021.00010`.

[51] Timnit Gebru et al. “Datasheets for datasets”. In: *Communications of the ACM* 64.12 (2021), pp. 86–92.

[52] Samadrita Ghosh. *Mlops challenges and how to face them.* Aug. 2021. URL: `https://neptune.ai/blog/mlops-challenges-and-how-to-face-them`.

[53] Google. *TensorBoard*. `tensorflow.org/tensorboard`. 2020.

[54] Google Cloud. *Vertex AI*. `https://cloud.google.com/vertex-ai/`. Accessed: 2024-02-06. 2021.

[55] *Google PAIR*. `https://research.google/teams/brain/pair/`. 2020.

[56] Tuomas Granlund et al. "MLOps challenges in multi-organization setup: Experiences from two real-world cases". In: *2021 IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI (WAIN)*. IEEE. 2021, pp. 82–88.

[57] Jim Gray and Andreas Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992.

[58] Greg Guest, Arwen Bunce, and Laura Johnson. "How many interviews are enough? An experiment with data saturation and variability". In: *Field methods* 18.1 (2006), pp. 59–82.

[59] Jean-Philippe Thiran Guillaume Jaume Hazim Kemal Ekenel. "FUNSD: A Dataset for Form Understanding in Noisy Scanned Documents". In: *Accepted to ICDAR-OST*. 2019.

[60] Philip J. Guo et al. "Proactive Wrangling: Mixed-Initiative End-User Programming of Data Transformation Scripts". In: *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*. UIST '11. Santa Barbara, California, USA: Association for Computing Machinery, 2011, pp. 65–74. ISBN: 9781450307161. DOI: 10.1145/2047196.2047205. URL: https://doi.org/10.1145/2047196.2047205.

[61] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[62] Joseph M Hellerstein et al. "Ground: A Data Context Service." In: *CIDR*. 2017.

[63] Jonathan Herzig et al. "TaPas: Weakly Supervised Table Parsing via Pre-training". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, July 2020, pp. 4320–4333. DOI: 10.18653/v1/2020.acl-main.398. URL: https://aclanthology.org/2020.acl-main.398.

[64] Charles Hill et al. "Trials and tribulations of developers of intelligent systems: A field study". In: *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2016), pp. 162–170.

[65] Sepp Hochreiter. "The vanishing gradient problem during learning recurrent neural nets and problem solutions". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.02 (1998), pp. 107–116.

[66] Fred Hohman et al. "Understanding and visualizing data iteration in machine learning". In: *Proceedings of the 2020 CHI conference on human factors in computing systems*. 2020, pp. 1–13.

[67] Alex Holkner and James Harland. "Evaluating the dynamic behaviour of Python applications". In: *ACSC*. 2009.

[68] Kenneth Holstein et al. “Improving Fairness in Machine Learning Systems: What Do Industry Practitioners Need?” en. In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems - CHI ’19*. Glasgow, Scotland Uk: ACM Press, 2019, pp. 1–16. ISBN: 978-1-4503-5970-2. DOI: `10.1145/3290605.3300830`. URL: `http://dl.acm.org/citation.cfm?doid=3290605.3300830` (visited on 08/03/2020).

[69] Susan Horwitz. “Identifying the semantic and textual differences between two versions of a program”. In: *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. 1990, pp. 234–245.

[70] Youyang Hou and Dakuo Wang. “Hacking with NPOs: Collaborative Analytics and Broker Roles in Civic Data Hackathons”. In: *Proc. ACM Hum.-Comput. Interact.* 1.CSCW (Dec. 2017). DOI: `10.1145/3134688`. URL: `https://doi.org/10.1145/3134688`.

[71] Jeremy Howard and Sebastian Ruder. “Universal language model fine-tuning for text classification”. In: *arXiv preprint arXiv:1801.06146* (2018).

[72] Silu Huang et al. “OrpheusDB: bolt-on versioning for relational databases”. In: *arXiv preprint arXiv:1703.02475* (2017).

[73] Yupan Huang et al. “LayoutLMv3: Pre-training for Document AI with Unified Text and Image Masking”. In: *Proceedings of the 30th ACM International Conference on Multimedia*. 2022.

[74] Duncan Hull et al. “Taverna: a tool for building and running workflows of services”. In: *Nucleic acids research* 34.suppl_2 (2006), W729–W732.

[75] Chip Huyen. *Machine learning tools landscape V2 (+84 new tools)*. Dec. 2020. URL: `https://huyenchip.com/2020/12/30/mlops-v2.html`.

[76] Forrest N Iandola et al. “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and¡ 0.5 MB model size”. In: *arXiv preprint arXiv:1602.07360* (2016).

[77] Pachyderm Inc. *Pachyderm: Data Versioning, Data Pipelines, and Data Lineage.* `https://www.pachyderm.com/`. Accessed: 2024-05-31. 2021.

[78] Iterative. *DVC: Data Version Control.* `https://dvc.org/`. Accessed: 2024-05-31. 2021.

[79] Christian S Jensen and Richard T Snodgrass. “Temporal data management”. In: *IEEE Transactions on knowledge and data engineering* 11.1 (1999), pp. 36–44.

[80] Meenu Mary John, Helena Holmström Olsson, and Jan Bosch. “Towards mlops: A framework and maturity model”. In: *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2021, pp. 1–8.

[81] Sean Kandel et al. “Enterprise Data Analysis and Visualization: An Interview Study”. In: *IEEE Transactions on Visualization and Computer Graphics* 18.12 (2012), pp. 2917–2926. DOI: `10.1109/TVCG.2012.219`.

[82] Sean Kandel et al. “Wrangler: Interactive visual specification of data transformation scripts”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2011, pp. 3363–3372.

[83] Daniel Kang et al. “Model assertions for debugging machine learning”. In.

[84] Mary Beth Kery, Amber Horvath, and Brad A Myers. “Variolite: Supporting Exploratory Programming by Data Scientists.” In: *CHI*. Vol. 10. 2017, pp. 3025453–3025626.

[85] Miryung Kim et al. “Data scientists in software teams: State of the art and challenges”. In: *IEEE Transactions on Software Engineering* 44.11 (2017), pp. 1024–1038.

[86] Miryung Kim et al. “The Emerging Role of Data Scientists on Software Development Teams”. In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE ’16. Austin, Texas: Association for Computing Machinery, 2016, pp. 96–107. ISBN: 9781450339001. DOI: 10.1145/2884781.2884783. URL: https://doi.org/10.1145/2884781.2884783.

[87] Janis Klaise et al. “Monitoring and explainability of models in production”. In: *ArXiv* abs/2007.06299 (2020).

[88] Johannes Köster and Sven Rahmann. “Snakemake—a scalable bioinformatics workflow engine”. In: *Bioinformatics* 28.19 (2012), pp. 2520–2522.

[89] Dominik Kreuzberger, Niklas Kühl, and Sebastian Hirschl. *Machine Learning Operations (MLOps): Overview, Definition, and Architecture*. 2022. DOI: 10.48550/ARXIV.2205.02302. URL: https://arxiv.org/abs/2205.02302.

[90] Sanjay Krishnan and Eugene Wu. “PALM: Machine Learning Explanations For Iterative Debugging”. en. In: *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics - HILDA’17*. Chicago, IL, USA: ACM Press, 2017, pp. 1–6. ISBN: 978-1-4503-5029-7. DOI: 10.1145/3077257.3077271. URL: http://dl.acm.org/citation.cfm?doid=3077257.3077271 (visited on 08/03/2020).

[91] Sean Kross and Philip Guo. “Orienting, Framing, Bridging, Magic, and Counseling: How Data Scientists Navigate the Outer Loop of Client Collaborations in Industry and Academia”. In: *Proc. ACM Hum.-Comput. Interact.* 5.CSCW2 (Oct. 2021). DOI: 10.1145/3476052. URL: https://doi.org/10.1145/3476052.

[92] Sean Kross and Philip J Guo. “Practitioners teaching data science in industry and academia: Expectations, workflows, and challenges”. In: *Proceedings of the 2019 CHI conference on human factors in computing systems*. 2019, pp. 1–14.

[93] Arun Kumar, Matthias Boehm, and Jun Yang. “Data management in machine learning: Challenges, techniques, and systems”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. 2017, pp. 1717–1722.

[94] Indika Kumara et al. “Requirements and Reference Architecture for MLOps: Insights from Industry”. In: (2022).

[95] Sampo Kuutti et al. “A Survey of Deep Learning Applications to Autonomous Vehicle Control”. In: *IEEE Transactions on Intelligent Transportation Systems* 22 (2021), pp. 712–733.

[96] Ladd. “Semantic diff: A tool for summarizing the effects of modifications”. In: *Proceedings 1994 International Conference on Software Maintenance*. IEEE. 1994, pp. 243–252.

[97] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. “Numba: A llvm-based python jit compiler”. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 2015, pp. 1–6.

[98] Angela Lee et al. *Demystifying a Dark Art: Understanding Real-World Machine Learning Model Development*. 2020. DOI: `10.48550/ARXIV.2005.01520`. URL: `https://arxiv.org/abs/2005.01520`.

[99] Angela Lee et al. “Demystifying a dark art: Understanding real-world machine learning model development”. In: *arXiv preprint arXiv:2005.01520* (2020).

[100] Cheng Han Lee. *3 data careers decoded and what it means for you*. Jan. 2020. URL: `https://www.udacity.com/blog/2014/12/data-analyst-vs-data-scientist-vs-data-engineer.html`.

[101] Yunseong Lee et al. “PRETZEL: Opening the black box of machine learning prediction serving systems”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 611–626.

[102] Viktor Leis et al. “How good are query optimizers, really?” In: *Proceedings of the VLDB Endowment* 9.3 (2015), pp. 204–215.

[103] Leonardo Leite et al. “A survey of DevOps concepts and challenges”. In: *ACM Computing Surveys (CSUR)* 52.6 (2019), pp. 1–35.

[104] Qian Li et al. “R3: Record-Replay-Retroaction for Database-Backed Applications”. In: *Proceedings of the VLDB Endowment* 16.11 (2023), pp. 3085–3097.

[105] Edo Liberty et al. “Elastic Machine Learning Algorithms in Amazon SageMaker”. In: *SIGMOD* (2020), pp. 14–19.

[106] Anderson Lima, Luciano Monteiro, and Ana Paula Furtado. “MLOps: Practices, Maturity Models, Roles, Tools, and Challenges-A Systematic Literature Review.” In: *ICEIS (1)* (2022), pp. 308–320.

[107] Zhiqiu Lin et al. “The CLEAR Benchmark: Continual LEArning on Real-World Imagery”. In: *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*. 2021. URL: `https://openreview.net/forum?id=43mYF598ZDB`.

[108] Eric Liu. “Low Overhead Materialization with Flor”. MA thesis. EECS Department, University of California, Berkeley, May 2020. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-79.html`.

[109] Mengchen Liu et al. "Towards better analysis of deep convolutional neural networks". In: *IEEE transactions on visualization and computer graphics* 23.1 (2016), pp. 91–100.

[110] David B Lomet and Feifei Li. "Improving transaction-time DBMS performance and functionality". In: *2009 IEEE 25th International Conference on Data Engineering*. IEEE. 2009, pp. 581–591.

[111] Mike Loukides. *What is DevOps?* " O'Reilly Media, Inc.", 2012.

[112] Lu Lu et al. "Dying relu and initialization: Theory and numerical examples". In: *arXiv preprint arXiv:1903.06733* (2019).

[113] Scott M Lundberg and Su-In Lee. "A unified approach to interpreting model predictions". In: *Advances in neural information processing systems*. 2017, pp. 4765–4774.

[114] Lucy Ellen Lwakatare et al. "DevOps in practice: A multiple case study of five companies". In: *Information and Software Technology* 114 (2019), pp. 217–230.

[115] Michael A. Madaio et al. "Co-Designing Checklists to Understand Organizational Challenges and Opportunities around Fairness in AI". In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI '20. Honolulu, HI, USA: Association for Computing Machinery, 2020, pp. 1–14. ISBN: 9781450367080. DOI: `10.1145/3313831.3376445`. URL: `https://doi.org/10.1145/3313831.3376445`.

[116] Samuel R Madden et al. "TinyDB: an acquisitional query processing system for sensor networks". In: *ACM Transactions on database systems (TODS)* 30.1 (2005), pp. 122–173.

[117] Michael Maddox et al. "Decibel: The relational dataset branching system". In: *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*. Vol. 9. 9. NIH Public Access. 2016, p. 624.

[118] Sasu Mäkinen et al. "Who needs MLOps: What data scientists seek to accomplish and how can MLOps help?" In: *2021 IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI (WAIN)*. IEEE. 2021, pp. 109–112.

[119] Beatriz MA Matsui and Denise H Goya. "MLOps: five steps to guide its effective implementation". In: *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI*. 2022, pp. 33–34.

[120] Peter Mattson et al. "Mlperf training benchmark". In: *arXiv preprint arXiv:1910.01500* (2019).

[121] Hui Miao, Amit Chavan, and Amol Deshpande. "ProvDB: A System for Lifecycle Management of Collaborative Analysis Workflows". In: *CoRR* abs/1610.04963 (2016). arXiv: `1610.04963`. URL: `http://arxiv.org/abs/1610.04963`.

[122] Hui Miao and Amol Deshpande. "ProvDB: Provenance-enabled Lifecycle Management of Collaborative Data Analysis Workflows." In: *IEEE Data Eng. Bull.* 41.4 (2018), pp. 26–38.

[123] Hui Miao et al. "Modelhub: Deep learning lifecycle management". In: *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE. 2017, pp. 1393–1394.

[124] Margaret Mitchell et al. "Model cards for model reporting". In: *Proceedings of the conference on fairness, accountability, and transparency*. 2019, pp. 220–229.

[125] MLReef. *Global mlops and ML Tools Landscape: Mlreef*. Feb. 2021. URL: `https://about.mlreef.com/blog/global-mlops-and-ml-tools-landscape/`.

[126] Akshay Naresh Modi et al. "TFX: A TensorFlow-Based Production-Scale Machine Learning Platform". In: *KDD 2017*. 2017.

[127] Chandrasekaran Mohan et al. "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging". In: *ACM Transactions on Database Systems (TODS)* 17.1 (1992), pp. 94–162.

[128] Jose G. Moreno-Torres et al. "A unifying view on dataset shift in classification". In: *Pattern Recognition* 45.1 (2012), pp. 521–530. ISSN: 0031-3203. DOI: `https://doi.org/10.1016/j.patcog.2011.06.019`. URL: `https://www.sciencedirect.com/science/article/pii/S0031320311002901`.

[129] Dennis Muiruri et al. "Practices and Infrastructures for ML Systems–An Interview Study in Finnish Organizations". In: (2022).

[130] Michael Muller. "Curiosity, creativity, and surprise as analytic tools: Grounded theory method". In: *Ways of Knowing in HCI*. Springer, 2014, pp. 25–48.

[131] Michael Muller et al. "How data science workers work with data: Discovery, capture, curation, design, creation". In: *Proceedings of the 2019 CHI conference on human factors in computing systems*. 2019, pp. 1–15.

[132] Eugene W Myers. "An O (ND) difference algorithm and its variations". In: *Algorithmica* 1.1-4 (1986), pp. 251–266.

[133] Mohammad Hossein Namaki et al. "Vamsa: Automated provenance tracking in data science scripts". In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2020, pp. 1542–1551.

[134] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. "Fast serializable multi-version concurrency control for main-memory database systems". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 2015, pp. 677–689.

[135] Andrew Ng et al. *Evaluating a model - advice for applying machine learning*. URL: `https://www.coursera.org/lecture/advanced-learning-algorithms/evaluating-a-model-26yGi`.

[136] Jens Nicolay et al. "Detecting function purity in JavaScript". In: *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE. 2015, pp. 101–110.

[137] Chris Olah et al. “The building blocks of interpretability”. In: *Distill* 3.3 (2018), e10.

[138] Yaniv Ovadia et al. “Can You Trust Your Model’s Uncertainty? Evaluating Predictive Uncertainty Under Dataset Shift”. In: *NeurIPS*. 2019.

[139] Cosmin Paduraru et al. “Challenges of Real-World Reinforcement Learning:Definitions, Benchmarks & Analysis”. In: *Machine Learning Journal* (2021).

[140] Andrei Paleyes, Raoul-Gabriel Urma, and Neil D. Lawrence. “Challenges in Deploying Machine Learning: A Survey of Case Studies”. In: *ACM Comput. Surv.* (Apr. 2022). Just Accepted. ISSN: 0360-0300. DOI: `10.1145/3533378`. URL: `https://doi.org/10.1145/3533378`.

[141] Samir Passi and Steven J Jackson. “Trust in data science: Collaboration, translation, and accountability in corporate data science projects”. In: *Proceedings of the ACM on Human-Computer Interaction* 2.CSCW (2018), pp. 1–28.

[142] Samir Passi and Steven J. Jackson. “Data Vision: Learning to See Through Algorithmic Abstraction”. In: *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing* (2017).

[143] Panupong Pasupat and Percy Liang. *Compositional Semantic Parsing on Semi-Structured Tables*. 2015. arXiv: `1508.00305 [cs.CL]`.

[144] Adam Paszke et al. “PyTorch: An imperative style, high-performance deep learning library”. In: *Advances in Neural Information Processing Systems*. 2019, pp. 8024–8035.

[145] Kayur Patel et al. “Investigating statistical machine learning as a tool for software development”. In: *International Conference on Human Factors in Computing Systems*. 2008.

[146] Neoklis Polyzotis et al. “Data Lifecycle Challenges in Production Machine Learning: A Survey”. en. In: *SIGMOD Record* 47.2 (2018), p. 12.

[147] Neoklis Polyzotis et al. “Data management challenges in production machine learning”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. 2017, pp. 1723–1726.

[148] Luisa Pumplun et al. “Adoption of machine learning systems for medical diagnostics in clinics: qualitative interview study”. In: *Journal of Medical Internet Research* 23.10 (2021), e29301.

[149] *PyTorch Documentation*. `pytorch.org/docs`. 2020.

[150] Mark Raasveldt and Hannes Mühleisen. “Duckdb: an embeddable analytical database”. In: *Proceedings of the 2019 International Conference on Management of Data*. 2019, pp. 1981–1984.

[151] Stephan Rabanser, Stephan Günnemann, and Zachary Chase Lipton. “Failing Loudly: An Empirical Study of Methods for Detecting Dataset Shift”. In: *NeurIPS*. 2019.

[152] Alec Radford et al. “Language Models are Unsupervised Multitask Learners”. In: (2019).

[153] Alexander Ratner et al. “Snorkel: Rapid training data creation with weak supervision”. In: *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*. Vol. 11. 3. NIH Public Access. 2017, p. 269.

[154] Alexander Ratner et al. “Sysml: The new frontier of machine learning systems”. In: *arXiv preprint arXiv:1904.03257* (2019).

[155] Paulo E Rauber et al. “Visualizing the hidden activity of artificial neural networks”. In: *IEEE transactions on visualization and computer graphics* 23.1 (2016), pp. 101–110.

[156] Gilberto Recupito et al. “A multivocal literature review of mlops tools and features”. In: *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2022, pp. 84–91.

[157] Cedric Renggli et al. “A data quality-driven view of mlops”. In: *arXiv preprint arXiv:2102.07750* (2021).

[158] Mitchel Resnick et al. “Design principles for tools to support creative thinking”. In: (2005).

[159] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. “”Why should I trust you?” Explaining the predictions of any classifier”. In: *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 2016, pp. 1135–1144.

[160] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. “Anchors: High-precision model-agnostic explanations”. In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[161] Yuji Roh, Geon Heo, and Steven Euijong Whang. “A Survey on Data Collection for Machine Learning: A Big Data - AI Integration Perspective”. In: *IEEE Transactions on Knowledge and Data Engineering* (2019). Conference Name: IEEE Transactions on Knowledge and Data Engineering, pp. 1–1. ISSN: 1558-2191. DOI: `10.1109/TKDE.2019.2946162`.

[162] Philipp Ruf et al. “Demystifying mlops and presenting a recipe for the selection of open-source tools”. In: *Applied Sciences* 11.19 (2021), p. 8861.

[163] Lukas Rupprecht et al. “Improving reproducibility of data science pipelines through transparent provenance capture”. In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 3354–3368.

[164] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: `10.1007/s11263-015-0816-y`.

[165] Nithya Sambasivan et al. "“Everyone wants to do the model work, not the data work”: Data Cascades in High-Stakes AI". In: *proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 2021, pp. 1–15.

[166] Sebastian Schelter et al. "Automating Large-Scale Data Quality Verification". In: *PVLDB'18*. 2018.

[167] Sebastian Schelter et al. "Automatically tracking metadata and provenance of machine learning experiments". In: *Machine Learning Systems Workshop at NIPS*. 2017, pp. 27–29.

[168] D Sculley et al. "Machine learning: The high-interest credit card of technical debt". In: (2014).

[169] D. Sculley et al. "Hidden Technical Debt in Machine Learning Systems". In: *NIPS*. 2015.

[170] Ramprasaath R Selvaraju et al. "Grad-cam: Visual explanations from deep networks via gradient-based localization". In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 618–626.

[171] Koushik Sen et al. "Jalangi: a selective record-replay and dynamic analysis framework for JavaScript". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 2013, pp. 488–498.

[172] Alex Serban et al. "Adoption and Effects of Software Engineering Best Practices in Machine Learning". In: *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ESEM '20. Bari, Italy: Association for Computing Machinery, 2020. ISBN: 9781450375801. DOI: 10.1145/3382494.3410681. URL: https://doi.org/10.1145/3382494.3410681.

[173] Shreya Shankar, Bernease Herman, and Aditya G. Parameswaran. "Rethinking Streaming Machine Learning Evaluation". In: *ArXiv* abs/2205.11473 (2022).

[174] Shreya Shankar and Aditya G Parameswaran. "Building Reactive Large Language Model Pipelines with Motion". In: *Companion of the 2024 International Conference on Management of Data*. 2024, pp. 520–523.

[175] Shreya Shankar and Aditya G. Parameswaran. "Towards Observability for Production Machine Learning Pipelines". In: *ArXiv* abs/2108.13557 (2022).

[176] Shreya Shankar et al. "”We have no idea how models will behave in production until production": How engineers operationalize machine learning". In: *Proceedings of the ACM on Human-Computer Interaction (CSCW)* (2024).

[177] Shreya Shankar et al. "“We have no idea how models will behave in production until production": How engineers operationalize machine learning". In: *Proceedings of the ACM on Human-Computer Interaction (CSCW)* (2024).

[178] Shreya Shankar et al. "Bolt-on, Compact, and Rapid Program Slicing for Notebooks". In: *Proc. VLDB Endow.* (Sept. 2023).

[179] James P Spradley. *The ethnographic interview.* Waveland Press, 2016.

[180] David A Spuler and A. Sayed Muhammed Sajeev. "Compiler detection of function call side effects". In: *Informatica* 18.2 (1994), pp. 219–227.

[181] Steve Nunez. *Why AI investments fail to deliver.* [Online; accessed 15-September-2022]. 2022. URL: `https://www.infoworld.com/article/3639028/why-ai-investments-fail-to-deliver.html`.

[182] Ion Stoica et al. "A Berkeley view of systems challenges for AI". In: *arXiv preprint arXiv:1712.05855* (2017).

[183] Michael Stonebraker. *The design of the Postgres storage system.* Morgan Kaufmann Publishers Burlington, 1987.

[184] Anselm Strauss and Juliet Corbin. "Grounded theory methodology: An overview." In: (1994).

[185] Emma Strubell, Ananya Ganesh, and Andrew McCallum. "Energy and policy considerations for deep learning in NLP". In: *arXiv preprint arXiv:1906.02243* (2019).

[186] Stefan Studer et al. "Towards CRISP-ML (Q): a machine learning process model with quality assurance methodology". In: *Machine learning and knowledge extraction* 3.2 (2021), pp. 392–413.

[187] Masashi Sugiyama et al. "Covariate Shift Adaptation by Importance Weighted Cross Validation". In: *JMLR*. 2007.

[188] RS Sutton. *The Bitter Lesson.* `incompleteideas.net/IncIdeas/BitterLesson.html`. 2019.

[189] Georgios Symeonidis et al. "MLOps - Definitions, Tools and Challenges". In: *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*. 2022, pp. 0453–0460. DOI: `10.1109/CCWC54503.2022.9720902`.

[190] Damian A Tamburri. "Sustainable mlops: Trends and challenges". In: *2020 22nd international symposium on symbolic and numeric algorithms for scientific computing (SYNASC)*. IEEE. 2020, pp. 17–23.

[191] Matteo Testi et al. "MLOps: A taxonomy and a methodology". In: *IEEE Access* 10 (2022), pp. 63606–63618.

[192] F-Y Tzeng and K-L Ma. *Opening the black box-data driven visualization of neural networks.* IEEE, 2005.

[193] Manasi Vartak. "ModelDB: a system for machine learning model management". In: *HILDA '16*. 2016.

[194] Manasi Vartak et al. "Mistique: A system to store and query model intermediates for model diagnosis". In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, pp. 1285–1300.

[195] Manasi Vartak et al. "ModelDB: A System for Machine Learning Model Management". In: *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. HILDA '16. San Francisco, California: ACM, 2016, 14:1–14:3. ISBN: 978-1-4503-4207-0. DOI: `10.1145/2939502.2939516`. URL: `http://doi.acm.org/10.1145/2939502.2939516`.

[196] Manasi Vartak et al. "ModelDB: a system for machine learning model management". In: *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. ACM. 2016, p. 14.

[197] Manasi Vartak et al. "ModelDB: a system for machine learning model management". In: *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. 2016, pp. 1–3.

[198] Michael M Vitousek et al. "Design and evaluation of gradual typing for Python". In: *Proceedings of the 10th ACM Symposium on Dynamic languages*. 2014, pp. 45–56.

[199] Alex Wang et al. "GLUE: A multi-task benchmark and analysis platform for natural language understanding". In: *arXiv preprint arXiv:1804.07461* (2018).

[200] Dakuo Wang et al. "Human-AI Collaboration in Data Science: Exploring Data Scientists' Perceptions of Automated AI". In: *Proc. ACM Hum.-Comput. Interact.* 3.CSCW (Nov. 2019). DOI: `10.1145/3359313`. URL: `https://doi.org/10.1145/3359313`.

[201] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.

[202] Joyce Weiner. "Why AI/data science projects fail: how to avoid project pitfalls". In: *Synthesis Lectures on Computation and Analytics* 1.1 (2020), pp. i–77.

[203] Wikipedia contributors. *MLOps — Wikipedia, The Free Encyclopedia*. [Online; accessed 15-September-2022]. 2022. URL: `https://en.wikipedia.org/w/index.php?title=MLOps&oldid=1109828739`.

[204] Olivia Wiles et al. "A Fine-Grained Analysis on Distribution Shift". In: *ArXiv* abs/2110.11328 (2021).

[205] Kanit Wongsuphasawat, Yang Liu, and Jeffrey Heer. "Goals, Process, and Challenges of Exploratory Data Analysis: An Interview Study". In: *ArXiv* abs/1911.00568 (2019).

[206] Doris Xin et al. "HELIX: Holistic Optimization for Accelerating Iterative Machine Learning". In: *Proceedings of the VLDB Endowment* 12.4 ().

[207] Doris Xin et al. "Helix: Holistic optimization for accelerating iterative machine learning". In: *Proceedings of the VLDB Endowment* 12.4 (2018), pp. 446–460.

[208] Doris Xin et al. "Production machine learning pipelines: Empirical analysis and optimization opportunities". In: *Proceedings of the 2021 International Conference on Management of Data*. 2021, pp. 2639–2652.

[209] Doris Xin et al. "Whither AutoML? Understanding the Role of Automation in Machine Learning Workflows". In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems.* CHI '21. Yokohama, Japan: Association for Computing Machinery, 2021. ISBN: 9781450380966. DOI: `10.1145/3411764.3445306`. URL: `https://doi.org/10.1145/3411764.3445306`.

[210] Ding Yuan et al. "Be conservative: enhancing failure diagnosis with proactive logging". In: *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12).* 2012, pp. 293–306.

[211] M. Zaharia et al. "Accelerating the Machine Learning Lifecycle with MLflow". In: *IEEE Data Eng. Bull.* 41 (2018), pp. 39–45.

[212] Matei Zaharia et al. "Accelerating the machine learning lifecycle with MLflow". In: *Data Engineering* (2018), p. 39.

[213] Amy X Zhang, Michael Muller, and Dakuo Wang. "How do data science workers collaborate? roles, workflows, and tools". In: *Proceedings of the ACM on Human-Computer Interaction* 4.CSCW1 (2020), pp. 1–23.

[214] Ce Zhang, Arun Kumar, and Christopher Ré. "Materialization optimizations for feature selection workloads". In: *ACM Transactions on Database Systems (TODS)* 41.1 (2016), pp. 1–32.

[215] Yu Zhang et al. "OneLabeler: A Flexible System for Building Data Labeling Tools". In: *CHI Conference on Human Factors in Computing Systems.* 2022, pp. 1–22.

[216] Wenting Zheng et al. "Fast databases with fast durability and recovery through multicore parallelism". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14).* 2014, pp. 465–477.