CYMPHONY: TOWARD A CROWDSOURCING PLATFORM FOR DATA INTEGRATION

By

Amanpreet Singh Saini

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2026

Date of final oral examination: 03/06/2026

The dissertation is approved by the following members of the Final Oral Committee:

Suman Banerjee, Professor, Computer Sciences, UW-Madison
AnHai Doan, Professor, Computer Sciences, UW-Madison
Paul Hanson, Professor, Center for Limnology, UW-Madison
Xiangyao Yu, Associate Professor, Computer Sciences, UW-Madison

To the One

Timeless, Formless, and Omnipresent

God

ACKNOWLEDGMENTS

First and foremost, as a Sikh, I would like to thank God, who has been my anchor and support throughout my PhD journey, and who has been working through all the people listed below to make this journey a success.

I would like to thank my advisor, Dr. AnHai Doan, who guided me through my PhD. While I learned many qualities from him, some of the most prominent ones are thinking deeply, working hard consistently, and communicating precisely. Also, in my opinion, this is something that is very hard to teach, but I also learned how to produce meaningful work while keeping things simple.

Next, I would like to thank my committee members Paul Hanson, Suman Banerjee, and Xiangyao Yu. Paul has always seemed to me like a larger-than-life person with a wide variety of talents and interests, who is also grounded and humble. Suman has always seemed like a person who would go out of his way to help others. Xiangyao has always come across as a kind and thoughtful professor. Thank you Paul, Suman, and Xiangyao, for taking the time out of your busy schedules to serve on my committee. I could not have asked for a better trio of committee members for my PhD defense.

I would also like to extend my sincere thanks to Angela Thorp, Cindy Fendrick, and Gigi Mitchell for all administrative support, and for being approachable, helpful, and kind across all of our interactions over the years.

Thank you to my loving and caring family for being there for me through the ups and downs of this journey. Thank you mom, for selflessly having my back, pushing me whenever I felt down, giving great advice whenever I needed it, and often guiding me through this journey like an unofficial advisor. Thank you dad, for being my grounding force, teaching me financial intelligence, and for showing me what courage looks like through your numerous examples in life. Thank you to my wife for always being by my side, showing me what it means to be full of life, teaching me resourcefulness, and caring for our family selflessly. Thank you to my bundle of joy, my son, who has taught me, by his example, that at the end of the day, life is a game that is best played by not taking it too seriously. Thank you to both my sisters for always motivating me, lending me their ear whenever I needed them, selflessly helping me, and being consistently supportive. Thank you to my father-in-law and mother-in-law for being kind, patient, and understanding throughout this journey.

Next, I would like to thank my group mates who made the final part of my journey feel much easier. A special thank you to my long-time colleague, Derek Paulsen, for giving me valuable advice on finalizing my dissertation and preparing for my defense. Thank you Mark Tervo, for helping me customize Cymphony, and in general, for being a hardworking, knowledgeable, and easy-to-work-with colleague. I deeply enjoyed our conversations on a wide range of topics. Thank

you Dev Ahluwalia for being a helpful and dependable peer with a great sense of humor. I greatly appreciated our conversations while pulling other people's legs. Extra thanks to both Mark Tervo and Dev Ahluwalia for helping me with logistics on the day of my defense. Thank you Ting Cai for keeping the room light with your jokes, and for demonstrating the value of balancing hard work with smart work. Thank you Minh Phan, for being a kind, thoughtful, and hardworking peer, and for teaching us some great organization skills. My PhD would not have been a success without your help.

In addition, thank you to all of my friends and colleagues I crossed paths with during my journey here at UW-Madison, and during my internships at Microsoft and Apple.

I would also like to thank my former manager at Goldman Sachs, Ramakrishna Venkataraman, and senior colleagues Narendra Devarasetty and Harshavardhan Arepalli, who nurtured my interest in systems for data management, and later formally recommended me to pursue higher studies at UW-Madison. I would also like to thank my former colleagues Bhopesh Bassi and Shweta Shrivastava, and others for helping me through the process of graduate school applications.

Equally, if not more important, are my thanks to the teachers from my early childhood years to my undergraduate years, who, brick by brick, laid the foundation for not only intellectual skills but also the social skills that are so necessary for being successful in life. My special heartfelt thanks to Vinender Tiwana, Anjli Sharma, Ravita Chaudhary, Sapna Puri, Harpreet Kaur, Divik Jain, Mohit Sardana, Ajay Mittal, Gagan Kaur, Divya Bansal, and Rajesh Bhatia.

The above list is far from complete, but thank you everyone for teaching me, shaping me into the person I am today, and making this success possible.

TABLE OF CONTENTS

Page

LIST OF TABLES vii

LIST OF FIGURES viii

ABSTRACT xi

1 Introduction 1

1.1 The Data Science Pipeline 1
1.2 Crowdsourcing for Data Integration 3
1.3 Existing Crowdsourcing Solutions 6
1.4 Limitations of Existing Solutions 7
1.5 Contributions of the Dissertation 11
1.6 Roadmap 16

2 Designing Cymphony 17

2.1 Defining Crowdsourcing Problems for Data Integration 17
2.2 Challenges in Defining Crowdsourcing Operators and Workflows 19
2.3 The Cymphony Solution 23
2.3.1 Modeling Workflows as DAGs of Operators 23
2.3.2 The 3a_kn Human Operator 25
2.3.3 The 3a_amt Human Operator 26
2.3.4 The sample_random Machine Operator 28
2.3.5 The exec_sql Machine Operator 29
2.3.6 Sample Complex Cymphony Workflow 32
2.4 Executing Cymphony Workflows 33

3 Implementing Cymphony 34

3.1 Overall System Architecture 34
3.2 Account Management Layer 37
3.3 Interface Gateway and API Layer 38
3.3.1 Requester Interface 39
3.3.2 Worker Interface 41
3.3.3 Programmatic API 44

Page

3.4 Data Management Layer . . . 46
3.4.1 Relational Schema Design . . . 46
3.4.2 Filesystem Namespace . . . 49
3.5 Core Orchestration and Execution Layer . . . 50
3.5.1 Requester Manager . . . 50
3.5.2 Parser . . . 51
3.5.3 Optimizer . . . 52
3.5.4 Executor . . . 54
3.5.5 Worker Manager . . . 55
3.6 Crowd Marketplace Integration Layer . . . 56
3.6.1 The `3a_amt` Human Operator . . . 56
3.6.2 AMT Manager . . . 56
3.7 An Illustrative Run . . . 58
3.8 System Maturity and Deployment Status . . . 59

4 Evaluating Cymphony . . . 61

4.1 Applying Cymphony to DI Problems . . . 61
4.1.1 Color Extraction from Product Titles . . . 62
4.1.2 Column Classification in a Data Catalog . . . 67
4.1.3 Entity Matching via Active Learning . . . 72
4.2 Scaling Experiments . . . 78
4.2.1 Target Scale . . . 79
4.2.2 The Simulator . . . 79
4.2.3 Experiments Across Size, Concurrency Levels, and Machines . . . 80
4.2.4 Do We Need Horizontal Scaling? . . . 85

5 Customizing Cymphony for Data Catalog Systems . . . 86

5.1 The Smartcat Data Catalog System . . . 86
5.2 Overall Solution Idea . . . 89
5.3 Realizing the Solution: Design and Implementation . . . 91
5.3.1 Customizing User Registration . . . 91
5.3.2 Adding the 3a_knlm Human Operator . . . 95
5.3.3 Do We Need to Customize the CY Program? . . . 102
5.3.4 The API-driven Cymphony–Smartcat Integration . . . 103
5.3.5 Solution Summary and Lessons Learned . . . 118
5.4 Evaluating the Solution . . . 120
5.4.1 API-Driven Smartcat Client for Evaluation . . . 120
5.4.2 Dataset . . . 123

Page

5.4.3 Experiments Across Multiple Dimensions 124
5.4.4 Conclusion 134

6 Conclusions 137

6.1 Summary of Contributions 137
6.2 Key Lessons and Broader Takeaways 138
6.3 Future Directions 140
6.4 Closing Remarks 141

Bibliography 142

LIST OF TABLES

Table Page

3.1 Programmatic API endpoints exposed by Cymphony. 44

4.1 Person table. 70

4.2 Greenhouse gas table. 71

4.3 Comparison of entity matching performance. 77

4.4 Scaling experiments with varying dataset sizes and worker concurrency. 81

4.5 Scaling experiments with varying dataset sizes and worker concurrency, with red rows indicating additional high-concurrency runs. 83

4.6 Runtime comparison on small vs. big machine under varying dataset sizes and worker concurrency. 83

5.1 Comparison of bulk curation experiments. 127

5.2 Comparison of drive-by and mixed curation experiments. 130

LIST OF FIGURES

Figure Page

1.1 A typical data science pipeline. 2

1.2 Examples of common data integration problems. 2

1.3 Data integration problems are often addressed using crowdsourcing. 3

1.4 Crowdsourcing for data cleaning. 4

1.5 Crowdsourcing for entity matching. 5

1.6 Crowdsourcing for data catalogs. 6

1.7 The interface of Amazon Mechanical Turk (AMT). 8

1.8 An example of a complex crowdsourcing workflow interleaving human input and machine processing. 9

1.9 Stand-alone Cymphony deployment showing end-to-end management of DI workflows. 13

1.10 Cymphony as a backend component within the Smartcat data catalog system. 15

2.1 The input and output for extracting hard disk space from product titles. 18

2.2 A simple Cymphony workflow. 23

2.3 DAG representation of the complex Cymphony program for precision estimation. . . . 23

2.4 Cymphony interacting with Amazon Mechanical Turk (AMT). 26

2.5 Sample complex Cymphony workflow. 32

3.1 The Cymphony system architecture. 35

3.2 The sign up page. 38

Figure Page

3.3 The activation email. 39
3.4 The login screen. 39
3.5 Dashboard displaying user projects. 40
3.6 Manage workflows in a project. 41
3.7 Upload files to a workflow. 41
3.8 Manage runs of a workflow. 42
3.9 Download results from a run. 42
3.10 Worker interface for annotating tasks in a job. 43
3.11 An illustrative complex Cymphony workflow. 58
4.1 Input and expected output for extracting color from product titles. 62
4.2 Color extraction workflow modeled using Cymphony operators. 63
4.3 CY File for Color Extraction (start). 64
4.4 CY File for Color Extraction (end). 65
4.5 Column classification in a data catalog. 68
4.6 Question format presented to workers for column classification. 69
4.7 Input data representation used by Cymphony for column classification. 69
4.8 Example question presented to workers for column classification. 70
4.9 An entity matching example. 72
4.10 An entity matching workflow. 73
4.11 Desired EM workflow using Cymphony. 75
4.12 Python code interacting with Cymphony. 77
4.13 Task shown to worker. 78

Figure Page

4.14 Varying Dataset Size. 82

4.15 Varying the number of workers. 84

5.1 Architecture of the Smartcat (SC) backend. 88

5.2 API-driven integration of Smartcat and Cymphony. 90

5.3 The input and output for extracting hard disk space from product titles. 96

5.4 A catalog curation scenario. 104

5.5 API-driven Smartcat client interacting with Cymphony through bulk and drive-by curation APIs. 122

5.6 Excerpt from the Smartcat-generated EDI column expansion dataset. 123

5.7 Worker Facing Interface. 133

5.8 Short Instructions. 133

5.9 Full Instructions. 134

5.10 Complete View of Full Instructions. 135

ABSTRACT

This dissertation studies crowdsourcing as a fundamental mechanism for data integration (DI) tasks such as information extraction, data cleaning, and entity matching. These tasks often require human judgment to achieve high accuracy. Although both academia and industry have demonstrated the effectiveness of crowdsourcing, the current ecosystem remains fragmented. Academic research typically focuses on narrow algorithmic problems without producing reusable, end-to-end systems, while industrial platforms expose limited abstractions or require substantial financial and operational investment. Consequently, many data science teams still rely on ad-hoc scripts, spreadsheets, and manual coordination to manage human-in-the-loop workflows, resulting in brittle and non-reusable solutions.

We argue that crowdsourcing for data integration should be treated as a first-class systems problem. Rather than viewing human annotation as a peripheral service, we model it as a structured execution substrate with explicit semantics and modular abstractions. The central thesis of this work is that complex human–machine workflows can be expressed declaratively, executed systematically, and integrated cleanly into larger data systems through principled system design.

To realize this vision, we present **Cymphony**, a general-purpose crowdsourcing platform tailored to DI workflows. Cymphony models workflows as directed acyclic graphs (DAGs) of operators over relational artifacts. Human operators encapsulate task assignment, annotation, and aggregation of noisy labels, while machine operators perform deterministic transformations such as sampling and SQL-based processing. This operator abstraction enables users to compose multi-stage workflows that interleave human input with automated computation without exposing low-level

coordination logic. The system provides end-to-end lifecycle support, including task instantiation, worker coordination, data management, orchestration, and integration with external marketplaces such as Amazon Mechanical Turk. All execution artifacts are materialized as relational tables, enabling downstream analysis and seamless integration with traditional data tools.

We evaluate Cymphony along multiple dimensions. End-to-end experiments on representative DI tasks demonstrate that the platform captures real-world human–machine workflows and achieves high-quality labels at reasonable cost. A systematic scalability study shows near-linear scaling under fixed concurrency and predictable performance gains with increased parallelism, demonstrating practicality for real-world workloads.

Finally, we integrate Cymphony into the Smartcat data catalog system via an API-driven architecture and introduce role-aware aggregation semantics to support heterogeneous worker populations. In this setting, Smartcat retains control over user interaction and metadata management while delegating the orchestration of curation workflows to Cymphony. This case study illustrates how Cymphony can function as a reusable human-in-the-loop backend within larger data management systems.

Overall, this dissertation shows that crowdsourcing for data integration can be systematized through explicit execution semantics and modular abstractions, transforming it from isolated task scripts into a scalable and extensible data management platform.

Chapter 1

Introduction

In this dissertation, we study *crowdsourcing (CS)*, the practice of leveraging human workers to solve computational tasks that are difficult to automate reliably [30, 21]. We focus in particular on crowdsourcing as a core mechanism within data science pipelines. We begin by defining the data science pipeline in Section 1.1. Section 1.2 then discusses how crowdsourcing is commonly used within this pipeline. We review existing crowdsourcing solutions in Section 1.3 and analyze their limitations with respect to end-to-end support, generality, and extensibility in Section 1.4. Section 1.5 summarizes the contributions of this dissertation. We conclude with a roadmap of the rest of the dissertation in Section 1.6.

1.1 The Data Science Pipeline

Figure 1.1 illustrates a typical data science pipeline. The process often begins with structured data stored in relational databases, from which relevant tables are extracted. In parallel, unstructured sources such as news articles may be processed to extract structured information, which is likewise represented as tables. These tables are then combined to produce integrated datasets.

Once the data has been integrated, analysts can perform downstream analyses. For example, to study the relationship between geographic location and revenue, an analyst may query the resulting table to evaluate potential correlations. The stages involved in extracting, transforming, and combining data are collectively referred to as *data integration* [17, 23]. In practice, data integration (DI) occupies a substantial portion of the data science pipeline and encompasses a broad range of challenges [22, 18]. In this dissertation we will focus on the crowdsourcing challenges for DI.

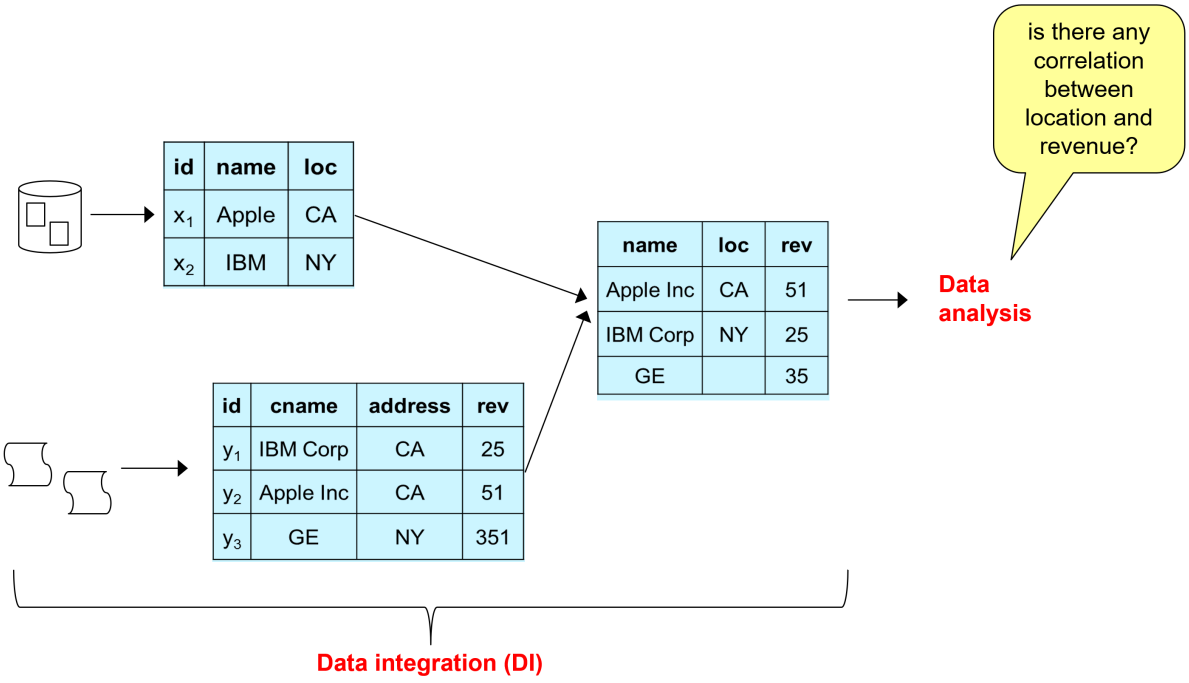

Figure 1.1: A typical data science pipeline.

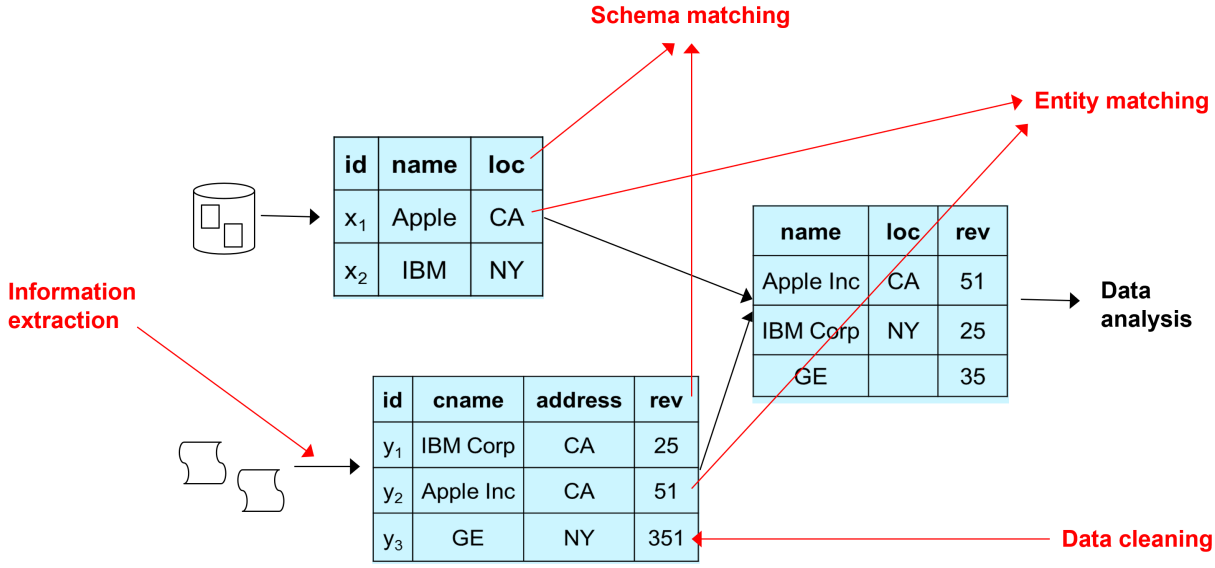

Figure 1.2: Examples of common data integration problems.

Several common data integration challenges are illustrated in Figure 1.2. For instance, transforming unstructured text such as news articles into structured tables is known as *information extraction* [49]. Ensuring the quality of structured data stored in databases is addressed through *data cleaning* [9, 31]; for example, correcting erroneous values such as interpreting an entry of `351` in a revenue column as `35`. Identifying records that refer to the same real-world entity across one or more tables is referred to as *entity matching* [24, 28, 8]. Finally, discovering semantically corresponding columns across different tables or data sources is known as *schema matching* [47, 4].

1.2 Crowdsourcing for Data Integration

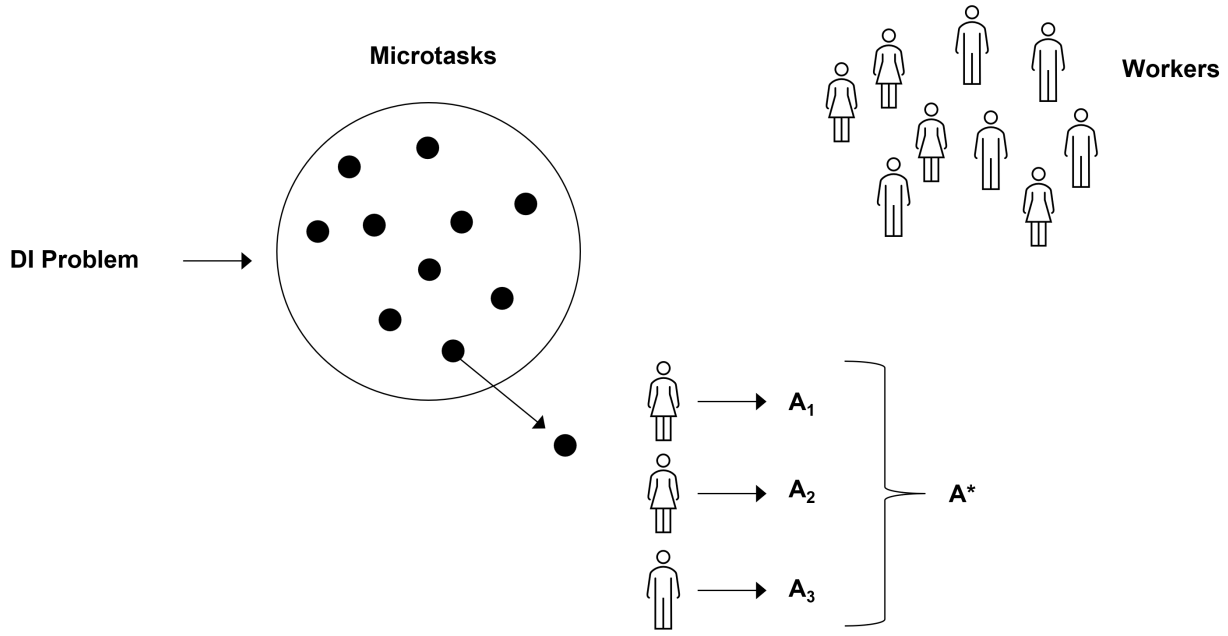

Figure 1.3: Data integration problems are often addressed using crowdsourcing.

These data integration tasks are difficult to solve fully automatically and are therefore commonly addressed using crowdsourcing [1, 11, 21, 36]. As illustrated in Figure 1.3, a data integration problem can often be decomposed into a collection of small, independent microtasks. Given a pool of workers, each microtask can be assigned to one or more workers, whose responses are

then aggregated to produce a final answer for that task. We illustrate this approach using three representative examples.

Example 1.2.1. *As shown in Figure 1.4, consider the task of cleaning a revenue column in a table. Suppose a data science team consists of five members. Each tuple in the table can be assigned to a team member, who is asked to either verify the correctness of the revenue value or correct it if necessary. Consequently, these human judgments yield a cleaned version of the dataset.*

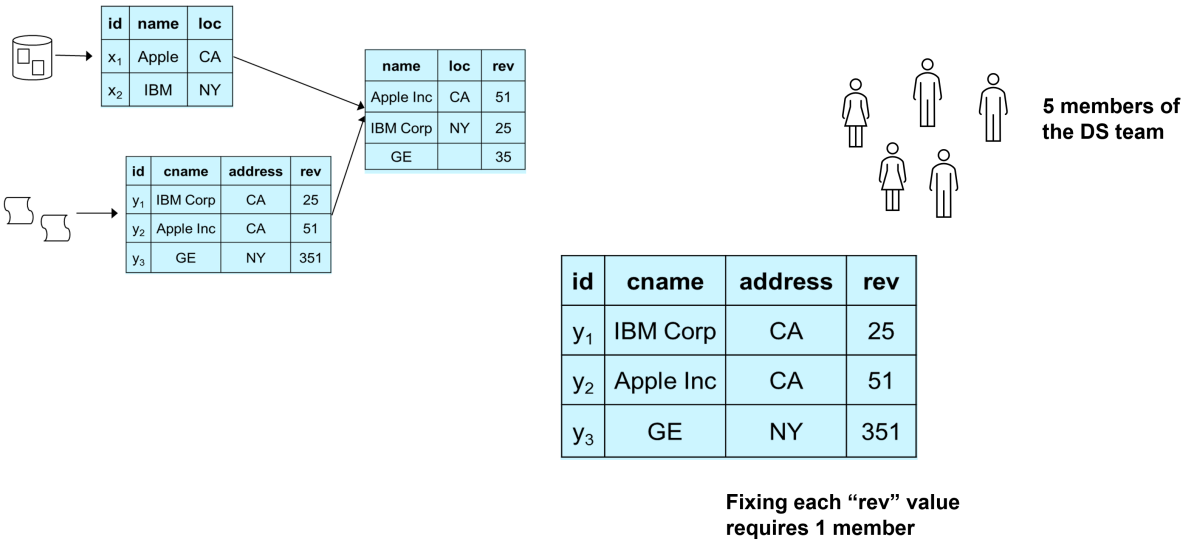

Figure 1.4: Crowdsourcing for data cleaning.

Example 1.2.2. *In entity matching, the goal is to determine whether pairs of tuples from two tables refer to the same real-world entity [24]. As illustrated in Figure 1.5, each microtask presents a worker with a candidate tuple pair and asks whether the two records match. To improve robustness, each microtask can be assigned to multiple workers (e.g., three), and their responses aggregated—typically using majority voting [1]—to produce a final label for that microtask.*

Example 1.2.3. *Crowdsourcing also plays an important role in modern data catalogs [51], as shown in Figure 1.6. Large organizations often maintain hundreds of thousands of tables, making it difficult for data scientists to efficiently discover relevant data assets. To address this challenge,*

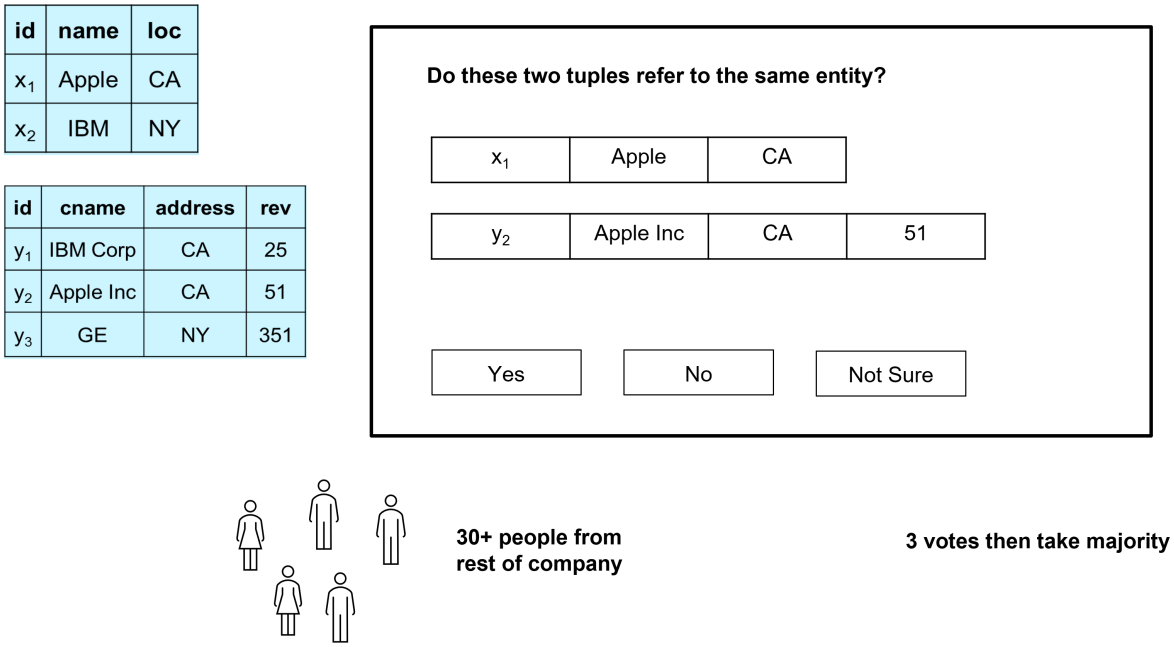

Figure 1.5: Crowdsourcing for entity matching.

data catalogs automatically crawl available tables, extract metadata, and construct a graph representation whose nodes correspond to tables and columns, and whose edges capture structural or semantic relationships.

Once constructed, the catalog enables users to search for relevant tables. However, column and table names are often cryptic (e.g., abbreviated or domain-specific), which limits the effectiveness of keyword-based search. To mitigate this issue, catalogs commonly apply machine learning techniques to expand or normalize column names—for example, mapping a column name such as DTPh to a more descriptive phrase such as "Day Time Phone".

Determining whether such expansions are correct often requires human validation. Crowdsourcing can be used to verify these expanded names by assigning validation tasks to workers. If a task is reviewed by a trusted data steward, a single validation may suffice, whereas validation by general catalog users may require multiple votes that are subsequently aggregated using majority voting.

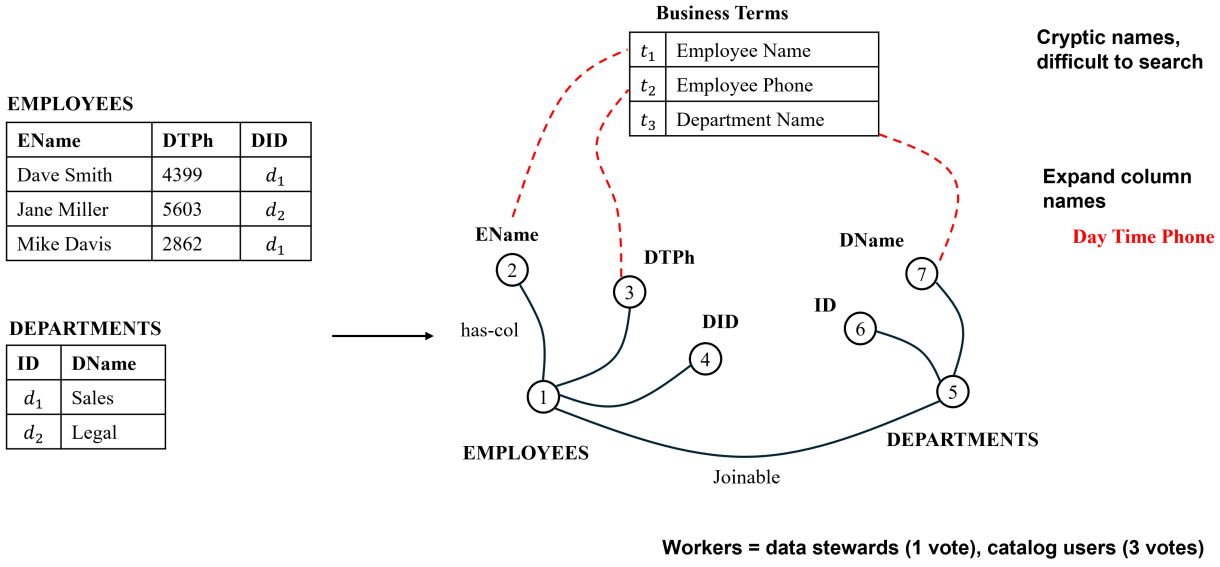

Figure 1.6: Crowdsourcing for data catalogs.

Across these settings, the key commonality is that large data integration problems are decomposed into collections of small, independent microtasks whose execution and aggregation must be carefully coordinated.

1.3 Existing Crowdsourcing Solutions

The previous section illustrated how crowdsourcing is frequently used to address data integration challenges. Despite this widespread use, the current landscape of crowdsourcing solutions remains fragmented and limited, both in academia and in industry.

Academic Approaches: Academic research between approximately 2003 and 2010, including pioneering work from our group [39, 19, 40, 41, 14, 13, 7, 20], helped establish crowdsourcing as a viable technique for data management. This was followed by a period of intense activity from 2011 to 2020 in the broader database community, during which crowdsourcing became a major research focus [16, 36, 11, 37, 6].

This body of work produced a large number of papers, mostly solving narrowly scoped, *point problems*. For example, algorithmic solutions were developed for solving problems such as entity

matching [55, 54, 29, 12, 44], while optimizing specific objectives such as minimizing crowdsourcing cost or maximizing labeling accuracy under constrained budgets [46, 57, 26].

The proposed solutions were often evaluated in isolation and tailored to specific tasks or assumptions. As a result, there has been relatively little emphasis on end-to-end system development, reusable execution frameworks, or open-source platforms that support diverse crowdsourcing workflows [15]. Consequently, many of these techniques remain difficult to adopt or extend in practice.

Industrial Approaches: In contrast, industry has primarily adopted crowdsourcing through either tight integration into larger systems, or cloud hosted platforms, or using contractors.

Cloud hosted platforms include self-service solutions like Amazon Mechanical Turk (AMT) [2], Appen [3], Clickworker [10], Toloka [53], and Microworkers [43]. The most prominent example is AMT, which provides a cloud-based self-service marketplace for crowdsourced tasks [2, 33]. It typically exposes crowdsourcing through simple, flat task abstractions. As illustrated in Figure 1.7, each task consists of a collection of microtasks that are presented to workers independently. Workers are compensated on a per-microtask basis, often at very small monetary increments. While such platforms are flexible and widely used, they provide limited native support for expressing complex workflows, task dependencies, or role-specific aggregation semantics.

Beyond self-service hosted platforms, some organizations rely on specialized contractors that provide managed crowdsourcing services. Contractors such as Sama [48], iMerit [32], and Appen [3] supply dedicated human workforces and often do crowdsourcing tailored to specific business needs of big companies.

1.4 Limitations of Existing Solutions

Because the state of the art is limited, it remains difficult for data science teams to effectively incorporate crowdsourcing into their workflows. While the previous section introduced these limitations, we now discuss them in more detail.

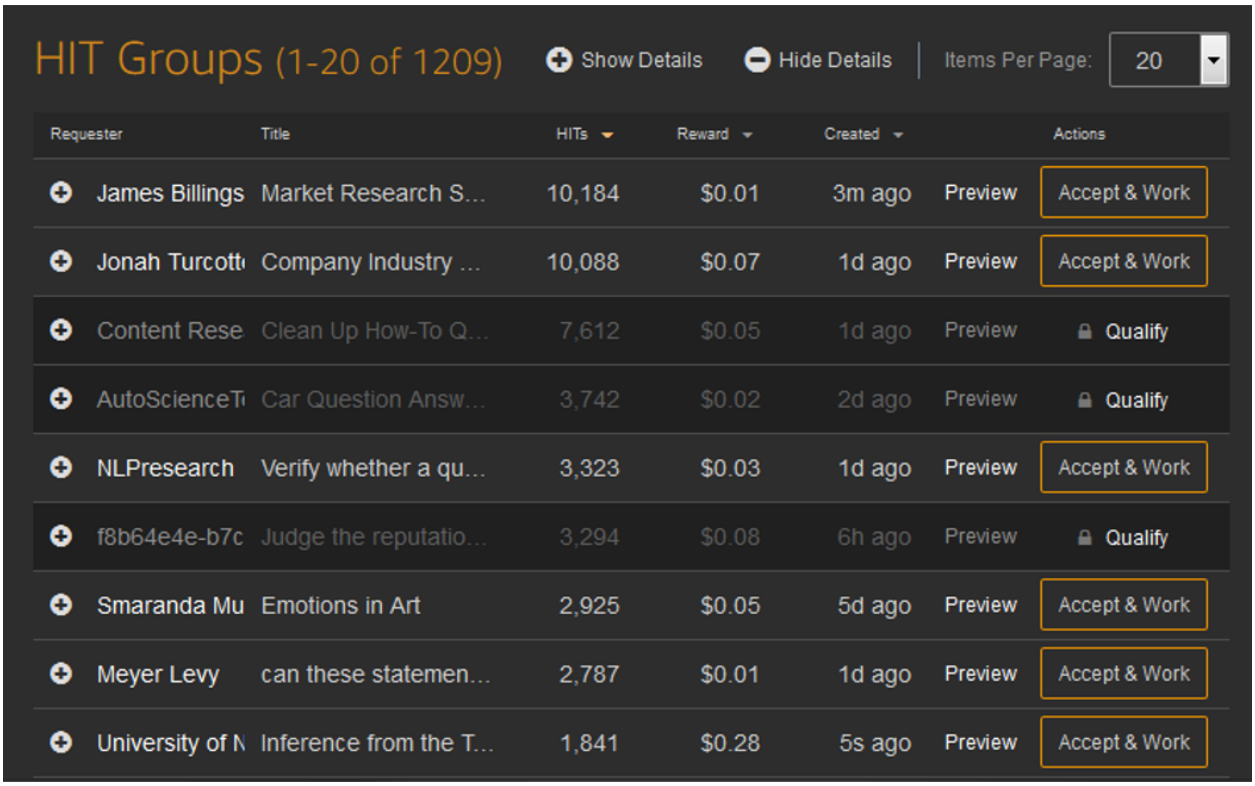

Figure 1.7: The interface of Amazon Mechanical Turk (AMT).

Limitations of Industrial Platforms: First, many teams are unable to rely on industrial, cloud-hosted crowdsourcing services. While platforms such as Amazon Mechanical Turk provide convenient access to a large workforce, they expose only limited abstractions, are difficult to customize or extend, and offer little support for complex, multi-stage workflows [34]. Moreover, organizations are often unwilling or unable to transmit sensitive or proprietary data to external, third-party services, further restricting the applicability of hosted solutions.

Second, contract-based crowdsourcing services present a different set of challenges. Vendors such as Sama [48], iMerit [32], and similar providers typically operate at a scale and cost structure suited to large enterprises. Engagements often require substantial financial commitments—frequently on the order of hundreds of thousands of dollars—and involve long turnaround times. As a result, these solutions are generally accessible only to large organizations such as Google, Microsoft, or Walmart, and are impractical for smaller teams or academic groups.

Limitations of Academic Solutions: On the academic side, despite extensive prior research [36, 11], there is no widely available general-purpose crowdsourcing system that data science teams can readily adopt. Existing research prototypes are rarely designed for deployment, are not maintained as reusable software, and typically address only narrowly scoped tasks [15]. Consequently, there is no academic solution that can be quickly and inexpensively deployed, easily customized or extended, and that abstracts away routine infrastructure concerns.

Overall, in the absence of suitable tools, crowdsourcing is often performed in ad-hoc and cumbersome ways. Teams frequently resort to emailing spreadsheets, manually collecting responses, or writing task-specific scripts to coordinate workers and aggregate results. These approaches are time-consuming, error-prone, and difficult to maintain.

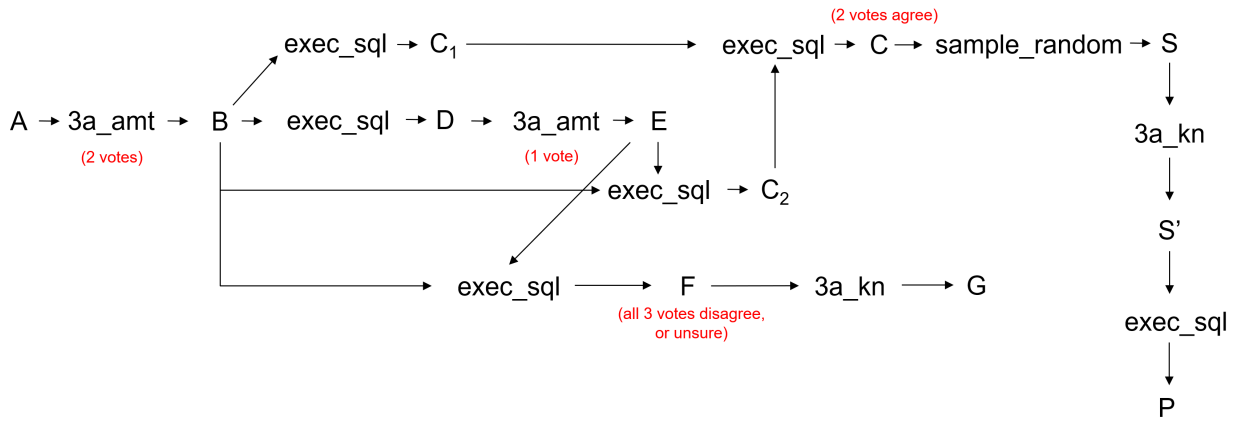

Figure 1.8: An example of a complex crowdsourcing workflow interleaving human input and machine processing.

The problem is exacerbated by the inherent complexity of real-world crowdsourcing workflows. Figure 1.8 shows a workflow for extracting the color attribute from product descriptions. While we describe this example in detail in later chapters, it already illustrates a key challenge: crowdsourcing workflows in data integration are rarely simple. They often involve multiple rounds of human input interleaved with machine processing [7, 52, 12]. For example, a workflow may first crowdsource labels from a large pool of workers, then sample uncertain cases for review by

experts, and finally compute evaluation metrics using SQL queries. The complexity arises not only from the number of steps, but also from the aggregation logic, intricate interactions inherently present when soliciting human votes, and coordination between human and machine-based steps. Implementing such workflows using custom, one-off code imposes significant development overhead and slows iteration, making crowdsourcing impractical at scale.

Together, these limitations motivate the need for a general-purpose end-to-end crowdsourcing system. The goal of this dissertation is to design and build such a crowdsourcing platform specifically for data integration (DI) tasks.

Challenges: Designing a practical crowdsourcing system for data integration raises several fundamental and interrelated challenges. While the high-level goal is to support human-in-the-loop data processing, achieving this goal in a reusable, scalable, and extensible system requires addressing multiple sources of complexity.

- **End-to-end system support:** The first major challenge is to build an end-to-end system that supports the full lifecycle of crowdsourcing workflows. This includes not only task generation and worker interaction, but also workflow orchestration, intermediate state management, aggregation of noisy human inputs [56], and persistent storage of results. In realistic settings, workflows consist of multiple human and automated steps with non-trivial dependencies, require coordination across long-running executions, and must tolerate asynchronous worker behavior. Supporting such workflows demands explicit execution semantics, robust coordination mechanisms, and careful management of execution artifacts, rather than ad-hoc task scripting.

- **General-purpose workflow expressiveness:** A second key challenge is to design the system to be general-purpose, rather than tailored to a single data integration task. Data integration encompasses a diverse set of problems—including data cleaning [9, 31], entity matching [24, 8, 35], information extraction [49], and catalog curation [51]. A practical system must therefore provide abstractions that are expressive enough to capture this diversity, while still offering a unified execution model. Achieving this balance requires decoupling

workflow specification from execution, supporting multiple human steps and aggregation strategies, and enabling the composition of human and automated processing steps within a single workflow.

- **Usability:** Third, the system should be easy to use, while supporting the below two usage patterns. In the first mode, the system should operate as a stand-alone deployment that users can directly interact with. In the second mode, the crowdsourcing system should be able to embed as a component within a larger system that requires human-in-the-loop processing [15].

- **Customization and extensibility:** Finally, the system must be easy to customize, and extend, without requiring users to reimplement core infrastructure. Data scientists and system builders should be able to define new workflows, adjust aggregation semantics, and integrate domain-specific logic without modifying the underlying execution engine. At the same time, the system must expose APIs and internal abstractions that allow it to be embedded into larger systems, such as data catalogs or analytics platforms. Reconciling ease of use with extensibility is challenging, as it requires carefully designed interfaces, stable execution semantics, and schema designs that can evolve without breaking existing workflows.

1.5 Contributions of the Dissertation

This dissertation makes several contributions toward the design, implementation, evaluation, and customization of practical crowdsourcing systems for data integration (DI). Taken together, these contributions demonstrate that crowdsourcing can be treated as a reusable, extensible, and scalable execution substrate, rather than as a collection of task-specific scripts or isolated human annotation pipelines.

End-to-End Crowdsourcing Execution Engine for Data Integration: We present *Cymphony*, a crowdsourcing platform that supports the end-to-end execution of human-in-the-loop data integration workflows. This contribution primarily addresses the challenge of end-to-end system support,

by establishing crowdsourcing as a first-class execution problem in data management systems. Specifically:

- Cymphony manages task instantiation, worker coordination, annotation collection, aggregation, intermediate state materialization, and final output production within a unified execution framework. We explain Figure 1.9 in detail in later chapters, but as a brief illustration, a user provides input data together with a declarative crowdsourcing workflow specification expressed in the Cymphony language. The system then instantiates and executes the workflow, while dynamically presenting microtasks to human workers and coordinating task execution. All execution artifacts, including assignments, annotations, and aggregated outputs, are materialized as relational tables, enabling downstream analysis. The final output of the workflow is exported as a table (e.g., `O.csv`) that can be directly consumed by subsequent data processing or analytics tasks.

- Unlike prior systems that focus on narrow crowdsourcing primitives or single-task pipelines [27, 38], Cymphony supports multi-stage workflows that interleave automated operators with human operators, execute over large datasets, and run under asynchronous and unpredictable worker behavior.

Operator-Based Abstraction for Expressive and Extensible Crowdsourcing Workflows: This dissertation introduces an operator-centric abstraction for crowdsourcing workflows, in which complex human–machine interaction patterns are encapsulated within modular operators. This abstraction addresses the challenges of general-purpose workflow expressiveness and extensibility, specifically:

- Each operator implements a well-defined transformation over relational artifacts while internally managing assignment, annotation, and aggregation logic.

- Workflows are expressed as directed acyclic graphs (DAGs) over operators, allowing users to compose sophisticated multi-stage processes without exposing low-level coordination details.

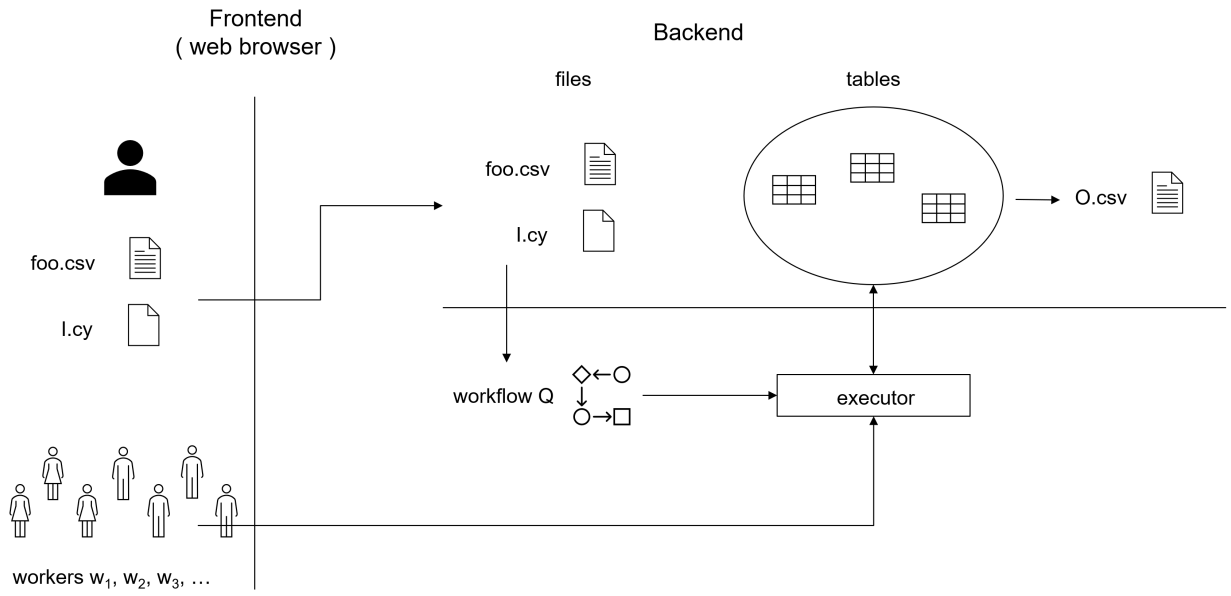

Figure 1.9: Stand-alone Cymphony deployment showing end-to-end management of DI workflows.

- New behaviors can be introduced by defining new operators without modifying the workflow language or the execution engine.

Realistic End-to-End Evaluation on Data Integration Tasks: A major contribution of this dissertation is a comprehensive evaluation of crowdsourcing workflows under realistic data integration scenarios. This evaluation addresses the challenge of usability, specifically:

- We evaluate Cymphony on three representative DI problems—information extraction [49], column classification, and entity matching via active learning [44, 50].

- These experiments demonstrate that Cymphony can faithfully express and execute workflows that closely mirror real-world practice, while achieving high accuracy at reasonable monetary cost using real datasets and real human workers.

- The evaluation goes beyond isolated task accuracy and instead measures end-to-end workflow behavior, including label quality, redundancy, sampling-based quality estimation, and interaction with downstream learning algorithms.

Systematic Scalability Study Under Controlled Workloads: We conduct a detailed scalability evaluation that isolates the runtime behavior of crowdsourcing execution from confounding human factors. This evaluation again addresses the challenge of usability, but this time from a realistic scalability perspective. Specifically:

- We first create a simulator that exercises the full Cymphony execution stack and use it to study system performance across increasing dataset sizes, worker concurrency levels, and machine resources.
- The experiments demonstrate that Cymphony scales linearly with dataset size under fixed concurrency, benefits predictably from increased worker parallelism, and approaches theoretical throughput bounds under modest vertical scaling.
- These results show that a single-node deployment is sufficient for the vast majority of practical DI workloads, providing empirical guidance for system designers and practitioners considering crowdsourcing at scale.

Role-Aware Aggregation with Heterogeneous Trust Models: To support realistic human-in-the-loop scenarios in large data catalog systems, we design and implement role-aware aggregation semantics that allow workers with different trust levels to participate in the same workflow. The contribution addresses the challenges of customization and extensibility by showing that such trust-aware behavior can be introduced as a localized operator extension rather than as a separate system or ad-hoc policy layer. Specifically:

- We introduce the `3a_knlm` operator, which generalizes majority voting by allowing distinct agreement thresholds for regular workers and expert stewards.
- We demonstrate how heterogeneous trust assumptions can be embedded directly within the operator model, enabling expert input to accelerate convergence and improve quality without altering workflow structure or execution semantics.

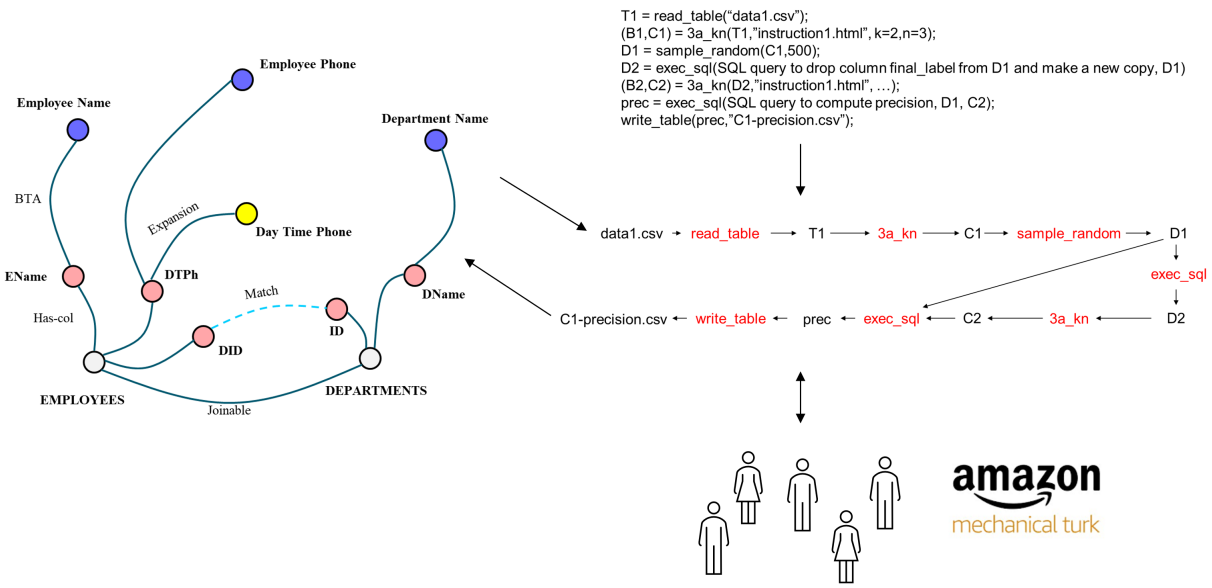

Figure 1.10: Cymphony as a backend component within the Smartcat data catalog system.

API-Driven Integration with a Real Data Catalog System: We demonstrate how Cymphony can be integrated into an existing data catalog system (Smartcat) through an API-driven architecture. The Smartcat case study addresses the challenges of usability, customization, and extensibility. Specifically:

- As shown in Figure 1.10, we show how large external systems such as Smartcat can interact with Cymphony programmatically via its APIs. In this setting, the external system retains control over its user interaction, metadata management, and application-specific logic, while delegating the orchestration of crowdsourcing workflows to the platform.

- We show that the integration supports multiple curation modes that a data catalog system such as Smartcat requires, while coordinating workers with heterogeneous roles.

- We also demonstrate that Cymphony's abstractions and execution semantics generalize beyond standalone experiments and can be embedded into larger data management systems without tight coupling or bespoke integration logic, allowing these systems to evolve independently.

Overall Impact: Together, these contributions demonstrate that crowdsourcing for data integration can be systematized through principled abstractions, execution semantics, and evaluation methodologies. Cymphony bridges the gap between academic crowdsourcing research [36, 16] and practical data systems by providing a reusable, extensible, and end-to-end crowdsourcing platform.

1.6 Roadmap

The remainder of this dissertation is organized as follows. Chapter 2 presents the design of Cymphony, introducing the operator-based workflow model and execution semantics for complex human–machine crowdsourcing pipelines. Chapter 3 describes the system architecture and implementation of Cymphony in detail. Chapter 4 evaluates Cymphony through real-world data integration case studies and controlled scalability experiments, analyzing accuracy, cost, and runtime behavior. Chapter 5 explores customization and integration, showing how Cymphony supports extensible operators, role-aware aggregation, and API-based interaction with external data systems. Chapter 6 concludes the dissertation.

Chapter 2

Designing Cymphony

In this chapter we describe the design of Cymphony. First, we define crowdsourcing problems for data integration using a standard input/output template. Second, we discuss the challenges in defining crowdsourcing operators and workflows. Third, we describe our solution to these challenges, focusing mainly on the Cymphony operators and how they can be stitched together to form complex crowdsourcing workflows. Finally, we discuss the execution of these workflows in Cymphony.

2.1 Defining Crowdsourcing Problems for Data Integration

As we discussed earlier in this dissertation, there is a wide variety of data integration (DI) problems, such as extraction, cleaning, classification, and matching. Each of these problems requires a different form of input and output. Clearly we cannot design Cymphony to accommodate all of these different input/output templates.

Our solution is to unify all of these problems using a single input/output template. Specifically, we propose that the input will be a table I of tuples. The output will be a table O, which is the same as I, but with an extra column at the end called *final-label*, whose cells are empty. The goal of crowdsourcing (CS) is to use crowd workers to fill in this last column. So each tuple in I will form a microtask, and we use workers to solve these tasks and fill in the last column.

Example 2.1.1. *Figure 2.1 shows a tiny example of information extraction. Here workers are required to extract the hard disk space from product titles. The input and output are shown based on the proposed problem template.*

Product Title	Price
”ASUS X205TA 11.6 Inch Laptop (Intel Atom, 2 GB, 32GB SSD, Gold) - Free Upgrade to Windows 10”	$199.00
”Lenovo G50 Entertainment Laptop - Black: DOORBUSTER - Intel Core i7-5500U (2.4GHz / 3.0 GHz Turbo), 8GB RAM, 1TB HDD, 15.6 FHD 1080P Display, DVD Burner, AC-WiFi, USB3.0, HDMI,Bluetooth, Windows 8.1”	$799.77
...	...

(a) Input

Product Title	Price	Final Label
”ASUS X205TA 11.6 Inch Laptop (Intel Atom, 2 GB, 32GB SSD, Gold) - Free Upgrade to Windows 10”	$199.00	32GB
”Lenovo G50 Entertainment Laptop - Black: DOORBUSTER - Intel Core i7-5500U (2.4GHz / 3.0 GHz Turbo), 8GB RAM, 1TB HDD, 15.6 FHD 1080P Display, DVD Burner, AC-WiFi, USB3.0, HDMI,Bluetooth, Windows 8.1”	$799.77	1TB
...	...	...

(b) Output

Figure 2.1: The input and output for extracting hard disk space from product titles.

As described, the above simple input/output template can be used to model a wide variety of DI tasks. For example:

- Consider an information extraction (IE) task where each tuple in a table describes a product. Our goal is to extract the weight from the title of the product, such as *3.5kg* from *Sony radio X67 3.5kg black*, and we have multiple workers that we want to use for this purpose. Here the input I is a table, where each tuple is a product. The last column is *final label* and the workers are supposed to put the weight in there.

- In data cleaning, we may have a column such as *City*, and we want workers to help clean its value, such as *New Yurk* to *New York*. Again, the workers are supposed to enter the clean value into the final column.

- Similarly, we may want workers to help normalize the values of the above column, such as *NY* and *NYC* to *New York City*. Again, workers are supposed to enter the normalized value into the final column.

- Also in data cleaning, an algorithm may detect multiple synonyms for missing values for the column *temperature*, and we may need a set of workers to examine these synonym candidates and select the correct ones.

  Here, the table I will be such that each tuple describes the context information for a missing value candidate V, followed by a cell describing the value V itself. The worker is supposed to enter *yes* or *no* into the last cell (of the final column), indicating if V is indeed a synonym for missing value.

- Suppose we want to use a set of workers to label tuple pairs *yes*/*no* for entity matching. Here the table I is such that each tuple is really a tuple pair, and then the worker must enter *yes*/*no* into the final column to indicate if the pair is a match or not.

2.2 Challenges in Defining Crowdsourcing Operators and Workflows

So far we have modeled a variety of DI problems with a clean crowdsourcing problem template: given an input table I, crowdsource it to produce an output table O, which is table I with an extra column at the end to capture the crowdsourcing output.

We envision creating a workflow W that consists of many operators, including some crowdsourcing operators, then executing workflow W to go from table I to table O. However, it turns out that it is non-trivial to design such workflows and operators, as we discuss below.

Consider the following simple CS problem: given a table T, output T with a new column *final_annotation*. For each tuple x in T, get 3 annotations from 3 workers, then take majority vote as the final annotation for that tuple.

Intuitively, we can create the following three operators:

- $x =$ `assign`(T, w, A): If a worker w arrives, then selects a tuple x from the table T and assigns it to worker w. Uses table A for storing this assignment.

- $a =$ `annotate`(x, w): Collects an annotation a from the worker w for the tuple x and store this annotation in a table B.

- $a^* = \texttt{aggregate}(x, B)$: Computes a final label a^* based on all collected annotations for x present in table B, and store this final label in a table C.

Thus, these operators use three tables: $A(x, w)$ for tracking assignments, $B(x, w, a)$ for storing raw annotations, and $C(x, a^*)$ for storing a final aggregated label corresponding to each tuple.

Unfortunately, it turns out that we cannot define a clean workflow, say a DAG (directed acyclic graph), involving these operators. This is because when we use these operators to solve the above problem, there are complex interactions among these operators that we cannot capture using a simple DAG.

Specifically, Algorithm 2.1 shows how we can use these three operators to solve the above CS problem. To explain this algorithm, Intuitively, at the start we have the input table T and three empty tables A, B, C. The stopping condition is that all tuples in T have final annotations $a*$ in C.

Then while the stopping condition is not yet true and a new worker w arrives, we execute the following steps:

- Execute $\texttt{assign}(T, w, A)$ to select a tuple $x \in T$ to assign to worker w. Remove x from T and add (x, w) to A.

- Execute $\texttt{annotate}(x, w)$ to obtain an annotation a for x. If $a = NULL$, meaning we fail to obtain an annotation (e.g., if the worker goes to lunch and abandons the task), then we need to put tuple x back to table T. Otherwise we add (x, w, a) to table B.

- If we manage to obtain an annotation a for x, then we now execute $\texttt{aggregate}(x, B)$ to obtain the aggregated annotation, a.k.a. label $a*$, for x. If $a* = NULL$, meaning we fail to obtain the aggregated label (e.g., because we have obtained only two labels for x, and they are different, so we cannot yet obtain a majority vote), then we must put x back into table T (so that eventually we can obtain another label for x). Otherwise we have the aggregated label $a*$, so we add $(x, a*)$ to table C.

As described, while we can use the above three operators to help solve the target problem, we need to use them in a complex algorithm, with interactions going back and forth among the

operators. As such, we cannot cleanly model this complex algorithm with a DAG involving just these operators.

Thus, the problem of at which level to define the operators and how to define the workflow involving these operators is highly non-trivial.

Algorithm 2.1 Complex Interactions Among Three Operators

Input: Tuples table T, worker pool W
Data: Assignments table A, Annotations table B, Final aggregates table C
Initialize $A, B, C \leftarrow \emptyset$
while there exists $x \in T$ such that $x \notin C$ **do**
    if new worker $w \in W$ arrives **then**
        $x \leftarrow \text{assign}(T, w, A)$ ▷ Assign tuple and track in A
        $T \leftarrow T \setminus \{x\}$ ▷ Remove x from T
        $A \leftarrow A \cup \{(x, w)\}$ ▷ Record assignment
        $a \leftarrow \text{annotate}(x, w)$ ▷ Collect worker annotation
        if a is **null then**
            $T \leftarrow T \cup \{x\}$ ▷ Reclaim tuple on abandonment
        else
            $B \leftarrow B \cup \{(x, w, a)\}$ ▷ Record annotation
            $a^* \leftarrow \text{aggregate}(x, B)$ ▷ Compute consensus
            if a^* is **not null then**
                $C \leftarrow C \cup \{(x, a^*)\}$ ▷ Add to final table
            else
                $T \leftarrow T \cup \{x\}$ ▷ Insufficient agreement; reclaim x
            end if
        end if
    end if
end while
return C

In addition to the above modeling challenge, we also face the following challenges that a robust crowdsourcing platform must resolve to be effective for data integration.

Integration with Hosted Services: The above solution assumes an in-house worker pool. However, many data integration tasks require scaling to external hosted services, such as Amazon Mechanical Turk (AMT). Integrating these services raises significant modeling questions: How do we abstract the communication with external APIs within the same operator framework? How do we handle service-specific constraints, such as worker qualifications?

Modeling Complex, Hybrid Workflows: Real-world DI workflows are rarely composed of a single crowdsourcing step. They often involve complex sequences that interleave human operations with automated machine operations. Consider a typical quality-assurance pipeline:

1. **Crowdsourcing:** A large table T is crowdsourced using 30 workers to produce a labeled table C.

2. **Sampling:** A machine operator takes a random sample of C to produce a smaller subset S.

3. **Expert Review:** The sample S is crowdsourced again, but this time using two highly-trusted data stewards to produce a *gold standard* table D.

4. **Computation:** A SQL query is executed over tables C and D to calculate the precision of table C (the initial crowdsourcing effort).

Our solution should enable users to quickly construct such complex CS worflows.

Developer Customization and Extensibility: Finally, for a platform to be actually useful in a research or industrial environment, it must be extensible. Developers need the ability to easily customize existing operators or define new ones without modifying the core system architecture. The challenge lies in providing a modular interface that allows these extensions to be *plugged in* and stitched together into the broader workflow seamlessly.

2.3 The Cymphony Solution

We now describe the Cymphony solution that addresses the above challenges.

2.3.1 Modeling Workflows as DAGs of Operators

We will model a Cymphony workflow as a DAG where nodes represent operators (or jobs) and edges represent the flow of data. We define an operator H to be $X = H(Y)$, where X and Y refer to tables, files, or atomic values such as integers, reals, strings, etc. This functional signature allows for a wide variety of operations, from human-centric crowdsourcing to automated operations like SQL-based transformations.

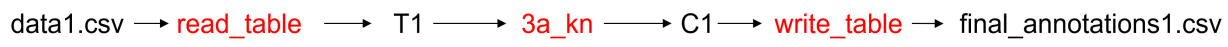

Figure 2.2: A simple Cymphony workflow.

A Cymphony workflow is specified declaratively as a Cymphony program. For example, the simple workflow in Figure 2.2 can be expressed as the following Cymphony program:

```
T1 = read_table("data1.csv");
(B1, C1) = 3a_kn(T1, "instruction1.html", k = 2, n = 3);
write_table(C1, "final_annotations1.csv");
```

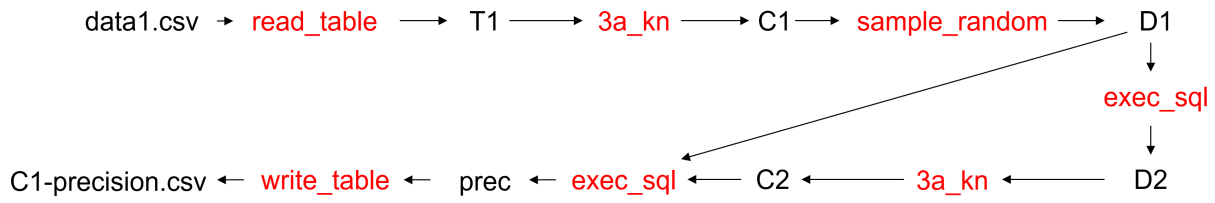

Figure 2.3: DAG representation of the complex Cymphony program for precision estimation.

As another example, Figure 2.3 describes the workflow that estimates the precision of crowdsourcing (described earlier in this chapter). The following program specifies that workflow:

```
T1 = read_table("data1.csv");
(B1, C1) = 3a_kn(T1, "instruction1.html", k = 2, n = 3);
D1 = sample_random(C1, 500);
D2 = exec_sql(
    SQL query to drop column final_label from D1 and make a copy,
    D1
);
(B2,C2) = 3a_kn(D2, "instruction1.html", k = 1, n = 1);
prec = exec_sql(SQL query to compute precision, D1, C2);
write_table(prec, "C1-precision.csv");
```

The current Cymphony system provides the following built-in operators:

- **Human Operators:** These operators take a table I, crowdsource, and produce a table O. We provide two human operators: `3a_kn` and `3a_amt`.
- **Machine Operators:** These operators automatically transform table/files and other value types. They do not involve any crowd workers. We provide two machine operators: `sample_random` and `exec_sql`.

These four operators already enable us to create very complex CS workflows, as illustrated later in this chapter (and see also the evaluation chapter). Developers however can easily customize existing operators or develop new ones using the operator template described earlier. In what follows we describe the above four operators in detail.

But before doing so, we note an important point. Observe that the above operators are defined at a higher level than that of the procedures `assign`, `annotate`, and `aggregate` described earlier. In fact, we will show below that an operator such as `3a_kn` is implemented using these procedures.

2.3.2 The 3a_kn Human Operator

The `3a_kn` operator is the core built-in human operator of Cymphony. It implements a majority vote strategy to fill the `final_label` column for a given input table. Specifically, for each micro-task t in the input table, it solicits up to n votes. If at least k votes agree, then it returns that as the *aggregated annotation*. Otherwise, it returns *null (undecided)*. This strategy allows us to capture many common crowdsourcing scenarios, such as getting up to 3 votes then taking the majority: $k = 2$, $n = 3$, or using 1 trustworthy data steward: $k = 1$, $n = 1$.

This operator has the form $(B, C) = 3a\_kn(I, Instr, k, n)$, where

- Input table I is a table where each tuple $x \in I$ represents a single microtask.
- Instruction file $Instr$ is an HTML file that defines the task completion instructions that are presented to the worker.
- Threshold k is the minimum number of workers who must agree on the same label for a tuple to be considered *completed*.
- Limit n is the maximum number of total annotations the system will collect for a single tuple before giving up on reaching a consensus.
- Annotation table B is an output table storing every individual raw annotation, with the schema *(tuple_id, worker_id, annotation)* where *tuple_id* represents a tuple $x \in I$.
- Aggregated table C is a table containing the final aggregated results. For each tuple $x \in I$, the system provides a final label a^* if at least k workers agreed; otherwise, the label is `NULL`.

Example 2.3.1. *To illustrate the `3a_kn` operator, consider an entity matching task where we set $k = 2$ and $n = 3$. For a tuple-pair $x \in I$:*

- *If Worker 1 says Yes and Worker 2 says Yes, the task finishes early with a Yes label for x.*
- *If Worker 1 says Yes, Worker 2 says No, and Worker 3 says No, the task finishes with a No label for x (reaching k = 2 at the last possible vote).*

- *If Worker 1 says Yes, Worker 2 says No, and Worker 3 says Cannot Determine, the task finishes with a* `NULL` *for* x *because no label achieved k = 2 agreements within n = 3 attempts.*

We implement this operator using Algorithm 2.1. Recall that this algorithm in turn uses the procedures `assign`, `annotate`, and `aggregate` discussed earlier. Thus, the *3a* in the name `3a_kn` comes from `assign`, `annotate`, and `aggregate`. The latter part *kn* comes from the majority voting parameters.

2.3.3 The 3a_amt Human Operator

The `3a_amt` operator is a specialized human operator designed to interface with external crowdsourcing marketplaces, specifically *Amazon Mechanical Turk* (AMT). See Figure 2.4. This operator allows Cymphony to scale beyond in-house worker pools by programmatically creating Human Intelligence Tasks (HITs) on AMT. It maintains a similar majority voting strategy to `3a_kn` where for each microtask t in the input table, it solicits n votes from the AMT workers. If at least k votes agree, then it returns that as the *aggregated annotation*. Otherwise, it returns *null (undecided)*. In addition to the majority voting strategy, it introduces parameters necessary for external service configuration and worker quality control.

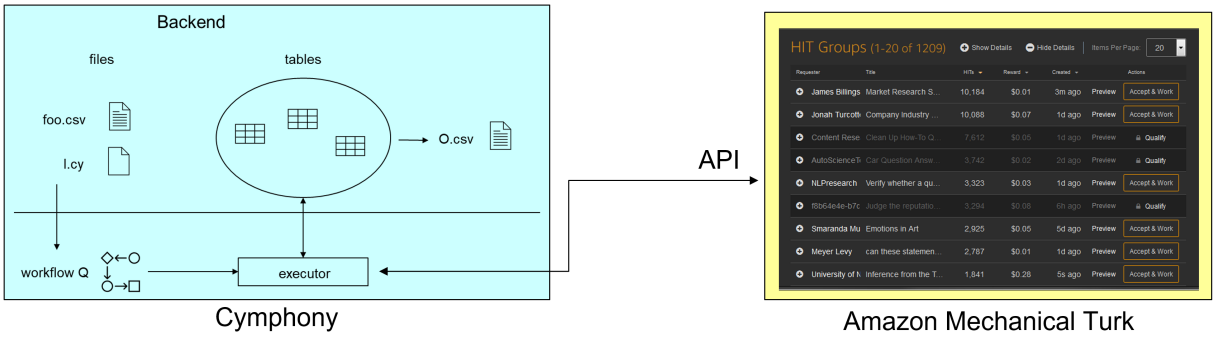

Figure 2.4: Cymphony interacting with Amazon Mechanical Turk (AMT).

This operator has the form $(B, C) = 3a\_amt(I, Instr, k, n, Qual, Price)$, where

- Input table I is a table where each tuple $x \in I$ represents a single microtask.

- Instruction file $Instr$ is an HTML file that defines the task completion instructions that are presented to the AMT workers.

- Threshold k is the minimum number of workers who must agree on the same label for a tuple to be considered *completed*.

- Limit n is the number of total annotations the system will collect for a single tuple before giving up on reaching a consensus.

  There is a minor difference in this parameter as compared to `3a_kn`. We collected *up to* n votes in `3a_kn`, where the system may stop early once k identical annotations are observed. In contrast, here in `3a_amt`, AMT always returns *exactly* n annotations per tuple, after which majority voting is applied. The reason is that there is no direct way to tell AMT to stop collecting more votes for a tuple if the consensus has been reached before hitting the limit.

- Qualification $Qual$ is a list of requirements that workers must meet to see the task (*e.g.*, a minimum HIT approval rate or a specific geographic location).

- The quantity $Price$ is the monetary reward (in USD) offered to each worker for each completed annotation.

- Annotation table B is an output table storing all individual raw annotations collected from AMT, with the schema *(tuple_id, amt_worker_id, annotation)* where *tuple_id* represents a tuple $x \in I$ and *amt_worker_id* is an *ID* given by AMT to the worker.

- Aggregated table C is a table containing the final aggregated results. For each tuple $x \in I$, the system provides a final label a^* if at least k workers agreed; otherwise, the label is `NULL`.

Example 2.3.2. *To illustrate the `3a_amt` operator, consider an entity matching task where we set $k = 2$, $n = 3$, and $geographic\_location = [\text{"}US - CA\text{"}, \text{"}IN\text{"}]$. When Cymphony executes this operator (called a job), only workers belonging to California and India will be able to view this job on their respective AMT dashboards. Let us say workers $w1$, $w2$, and $w3$ belong to these locations and start working on the job. Then, for a tuple-pair $x \in I$:*

- *If $w1$ says Yes and $w2$ also says Yes, the task does not finish early since $n = 3$ here implies exactly 3 votes will be collected by AMT. Say $w3$ then says Yes, then the task finishes with a Yes label for x (reaching k = 3 and n = 3).*
- *If $w1$ says Yes, $w2$ says No, and $w3$ says No, the task finishes with a No label for x (reaching k = 2 and n = 3).*
- *If $w1$ says Yes, $w2$ says No, and $w3$ says Cannot Determine, the task finishes with a `NULL` for x because no label achieved k = 2 agreements within n = 3 attempts.*

We also implement this operator using Algorithm 2.1, but with a modified *assign* and *annotate* phase. Instead of waiting for a worker to log into Cymphony, the system pushes the task to AMT via its API. Cymphony then periodically polls AMT to retrieve annotations, updating tables B and C as responses arrive. Since we maintain, albeit loosely, the `assign`, `annotate`, and `aggregate` lifecycle, we keep the *3a* in the name of the `3a_amt` operator.

2.3.4 The sample_random Machine Operator

The `sample_random` operator is a machine operator used to generate a representative subset of an input table. In DI workflows, this is frequently used to create a small set of tuples for quality assurance or to obtain a *gold standard* through expert review, as discussed in the precision-estimation example earlier in Figure 2.3.

This operator has the form $S = sample\_random(I, k)$, where

- Input table I is the source table from which tuples will be selected.
- Sample size k is an integer specifying the exact number of tuples to be randomly selected from the input table.
- Sampled table S is the output table containing exactly k tuples chosen uniformly at random from I. The schema of S is identical to the schema of I.

We implement this operator using a straightforward procedure. The system selects k rows at random from I. Then the selected tuples are copied into a new physical table S and stored in the

database. Once the table is materialized, the operator is marked as *completed*. If $k \geq |I|$, the system will just return the contents of table I as the output, without sampling it.

2.3.5 The exec_sql Machine Operator

The `exec_sql` operator is a flexible machine operator. It allows developers to execute arbitrary SQL queries to process the data, such as joining multiple tables or calculating accuracy metrics. Unlike traditional database queries, this operator is integrated into the Cymphony DAG, allowing it to take multiple input tables and map resulting database tables back to Cymphony variables for downstream use.

The operator has the form $(O_1, O_2, \ldots, O_n) = exec\_sql(I_1, \ldots, I_m, queries, mapping)$, where

- Input tables $I_1, \ldots, I_m$ are Cymphony tables that are referenced within the SQL queries.
- A string value $queries$ contains one or more SQL statements (e.g., `CREATE`, `SELECT`, `ALTER`).
- A mapping list $mapping$ maps internal tables created during the operator's execution to the specified output tables.
- Output tables $O_1, \ldots, O_n$ represent one or more tables generated by the SQL execution. The schema and content of each table are determined by the specific statements in the query block.

Example 2.3.3. *In many DI workflows, human operators may not always reach consensus on every tuple. For example, after applying a `3a_kn` operator, some tuples may be confidently labeled, while others may remain* undecided *due to insufficient agreement among workers. It is often necessary to explicitly separate these two groups so that downstream operators can process them differently (e.g., finalize the agreed-upon tuples while re-routing undecided tuples for further annotation). To illustrate the `exec_sql` operator and how it provides a convenient mechanism to express such logic using SQL, consider the following operator invocation:*

```
/* attach labels and separate out tuples that need more annotations. */
(
```

```
    AGREEMENTS_WITH_LABELS,
    DISAGREEMENTS_WITHOUT_LABELS
) = exec_sql(
    ORIGINAL_DATA,
    C_1,
    queries = "
        CREATE TABLE temp AS (
            SELECT _id, product_title, label
            FROM ORIGINAL_DATA
            INNER JOIN C_1
            USING (_id)
        );
        CREATE TABLE agreements AS (
            SELECT * FROM temp WHERE label != 'undecided'
        );
        CREATE TABLE disagreements AS (
            SELECT * FROM temp WHERE label = 'undecided'
        );
        ALTER TABLE disagreements DROP COLUMN label;
    ",
    mapping_to_output_variables = [
        "temp: None",
        "agreements: AGREEMENTS_WITH_LABELS",
        "disagreements: DISAGREEMENTS_WITHOUT_LABELS"
    ]
);
```

In this example, $ORIGINAL\_DATA$ contains the raw input tuples, while C_1 contains the aggregated labels produced by a prior human operator. The first SQL statement creates a temporary table $temp$ by joining these two tables on their shared tuple identifier $\_id$, thereby attaching the corresponding label to each tuple.

The subsequent statements partition $temp$ into two tables:

- *$agreements$, which contains all tuples whose label is not `undecided`, representing tuples for which consensus has been reached.*
- *$disagreements$, which contains only tuples labeled as `undecided`, representing those that require additional processing or further human annotation. The `ALTER TABLE` statement removes the $label$ column from $disagreements$, reflecting that these tuples are being forwarded without a final label.*

Finally, the `mapping_to_output_variables` list specifies how the internal SQL tables are mapped to Cymphony variables. The temporary table $temp$ is discarded, while $agreements$ and $disagreements$ are materialized as the output variables $AGREEMENTS\_WITH\_LABELS$ and $DISAGREEMENTS\_WITHOUT\_LABELS$, respectively.

This example illustrates how `exec_sql` enables developers to embed rich relational transformations within the Cymphony workflow DAG, allowing complex data partitioning and restructuring logic to be expressed declaratively and composed with both human and machine operators.

We implement this operator as follows. Assuming that all input tables are already materialized in the database, and that the SQL statements are valid, Cymphony first executes the block of SQL statements within the database context. It then parses the *mapping* list to identify which resulting tables should be registered as Cymphony variables ($O_1, \ldots, O_n$) and which should be treated as temporary (*None*). Once the output tables have been materialized using the mapping, the operator is marked as *completed*.

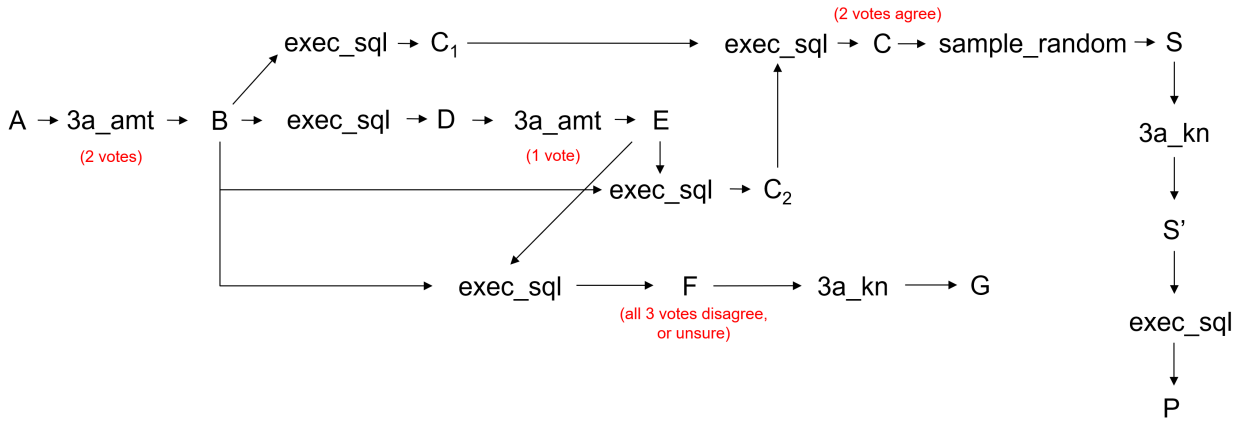

Figure 2.5: Sample complex Cymphony workflow.

2.3.6 Sample Complex Cymphony Workflow

The four operators just described are already sufficient to create quite complex CS workflows. In this section we describe one such workflow. Figure 2.5 shows the DAG of this workflow.

In this figure, table A contains 100 product descriptions. The task is to extract color out of each of those product descriptions. To start off, we feed table A into a $3a\_amt$ operator that will gather 2 votes per tuple, resulting in a table B. At this point, we execute SQL queries to separate out those tuples that got disagreeing votes (table D), from those that agreed on both the votes (table C_1).

For these tuples with disagreeing votes i.e. table D, we execute $3a\_amt$ once more, this time getting 1 vote per tuple, resulting in a table E. We join table E with table B on the tuple id via two $exec\_sql$ operations. One captures those tuples in whom 2 out of the 3 total collected votes agree, resulting in a table C_2. The other captures those tuples in whom the aggregated label is $undecided$ (all the 3 collected votes disagree) or is $unsure$ (i.e., Cannot determine), resulting in the table F.

At this point, the tuples that did receive valid aggregated labels reside in tables C_1 and C_2, so we union them via an $exec\_sql$ operation to get the table C. To figure out the precision of these crowdsourced results, we decide to take a sample out of table C via a $sample\_random$ operator to get a table S, and further pass that into a $3a\_kn$ operator to get gold labels from our in-house

domain experts, resulting in a table S'. We then compare S to S' to estimate precision on the crowdsourced turker results i.e. table C.

Since the tuples in F still do not have any valid labels, we crowdsource them using in-house experts via a $3a\_kn$ operation, resulting in a table G.

Therefore, not only do we have an estimate of the precision of crowdsourced results coming from turkers, but we have also collected labels for all the tuples that we originally had in table A. This final output table is: $(C \setminus S) \cup S' \cup G$.

2.4 Executing Cymphony Workflows

So far we have defined Cymphony workflows. We now describe how such workflows should be executed. We begin by defining the notion of *executable operators*. We say a machine operator is executable if all of its inputs are available. We say a human operator is executable if its non-human inputs (e.g., tables, files) are available.

Then we view time as happening in epoches (e.g., seconds). For each epoch:

- We execute all machine operators that are executable.
- If a worker w arrives, then we assign w (or let w choose) an executable human operator, then execute that human operator.

We terminate when all operators in the workflow have been executed.

Chapter 3

Implementing Cymphony

In this chapter we describe the implementation of Cymphony, which reflects the design of Cymphony discussed in the last chapter.

3.1 Overall System Architecture

We begin by defining a set of terms that will be used throughout the remainder of this dissertation:

- **Task:** The smallest unit of human work, corresponding to the annotation of a single tuple.
- **Worker:** A human participant, either in-house or external, who performs annotation tasks.
- **Requester:** An end user who specifies, submits, and manages collaborative workflows (e.g., for data cleaning, labeling, or integration).
- **User:** A system identity that may act as a requester, a worker, or both, depending on context.
- **Workflow:** A directed acyclic graph of automatic and/or human operators that specifies a collaborative application. Executing a workflow corresponds to executing the associated application.
- **Run:** A concrete execution instance of a workflow.
- **Job:** The smallest schedulable execution unit within a run, corresponding to a single operator node in the workflow DAG.

- **Annotation:** A raw label or data value produced by a worker for a specific tuple within a human job. Annotations are persisted and later aggregated to form final outputs.

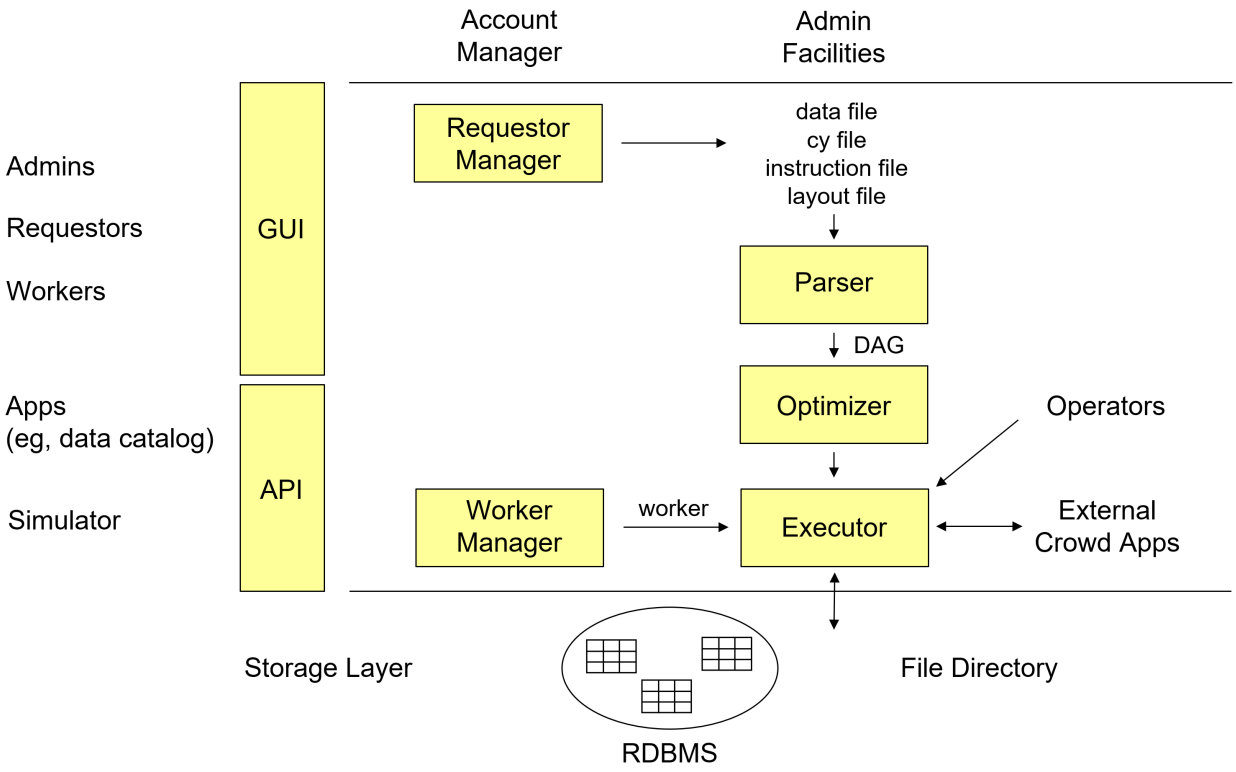

Figure 3.1: The Cymphony system architecture.

Having introduced the relevant terminology, we now describe the system architecture of Cymphony, illustrated in Figure 3.1. At a high level, Cymphony is implemented as a modular web-based platform built atop a relational database. To accommodate heterogeneous user roles, multiple interaction modalities, and the complex interplay between automatic and human-in-the-loop computation, the system is organized into five primary architectural layers:

1. **Account Management Layer:** Manages user accounts, including registration, activation, and authentication. See the part labeled "Account Manager" at the top of Figure 3.1.

2. **Interface Gateway and API Layer:** Provides graphical and programmatic interfaces for workflow specification, run management, and result retrieval, and hosts a reactive dashboard

for in-house workers. See the two boxes labeled “GUI” and “API” on the left side of the above figure.

3. **Data Management Layer:** Maintains persistent execution state and physical data artifacts using a hybrid relational–filesystem design. The relational database supports concurrent workflow executions and simultaneous worker participation within human jobs, while the filesystem organizes workflow artifacts and materialized outputs. See the parts labeled “Storage Layer”, “RDBMS”, and “File Directory” at the bottom of the figure.

4. **Core Orchestration and Execution Layer:** Implements the core workflow runtime, including components such as the Requester Manager, Parser, Optimizer, Executor, and Worker Manager. See the parts labeled “Requestor Manager”, “Worker Manager”, “Parser”, “Optimizer”, and “Executor” in the middle of the figure.

5. **Crowd Marketplace Integration Layer:** Enables integration with external crowd platforms—such as Amazon Mechanical Turk—allowing workflows to scale beyond the in-house worker pool. See the part labeled “External Crowd Apps” at the bottom right part of the figure.

Cymphony is implemented in Python using the Django web framework, with PostgreSQL as the underlying relational database, and is deployed behind the Gunicorn application server. We chose Python for its mature ecosystem and rapid development workflow, which is particularly useful for integrating heterogeneous components such as web services, database logic, and external marketplace APIs. Django provides a structured foundation for building the web application, including built-in support for authentication, request routing, and modular organization of the codebase. PostgreSQL serves as the primary persistent store for Cymphony’s metadata and tables, providing transactional guarantees needed to coordinate concurrent workflow runs and worker interactions. Finally, Gunicorn provides a lightweight WSGI-compliant server for running the Django application in production.

Now that we have looked at the overall system architecture, let us look closely at each of the five constituent layers.

3.2 Account Management Layer

In this section we briefly describe the account management layer of Cymphony.

Users, Accounts, and Authentication: Cymphony adopts a unified user model. A **user** represents a single authenticated identity in the system and may act as a *requester* or as a *worker* depending on the user's choice. That is, the same user account may create and execute workflows through the requester interface and also contribute annotations through the worker interface. We talk about these interfaces in detail in Section 3.3. A user **account** is associated with a unique username, password, and email address. Cymphony uses session-based **authentication** to maintain identity and execution context across user interactions. A session begins when a user successfully authenticates and ends when the user logs out or when the session expires. During a session, an opaque session identifier is exchanged between the client and server for each request. The session identifier allows the server to retrieve user identity and execution state maintained on the server.

Signup: As shown in Figure 3.2, a user can create an account by providing their name, username, password, and email address. Upon submission, the system creates the user account in an inactive state. As illustrated in Figure 3.3, an activation email is automatically sent to the supplied email address. This email contains a unique activation link that the user must visit to verify their account. Once the activation link is accessed, the account becomes active, and the user may subsequently authenticate using their username and password.

Login: As shown in Figure 3.4, a user authenticates by providing their username and password. If the credentials are valid and the account has been activated, the system establishes a persistent authenticated session and directs the user to their dashboard.

Design Rationale: Cymphony implements account creation and login using the `django-registration` package together with Django's built-in `django.contrib.auth` backend and `SessionMiddleware`. Underneath, they rely on two tables to support authentication and session state. Specifically, `auth_user(id, username, password, first_name, last_name, email)` stores user credentials and profile information used during authentication,

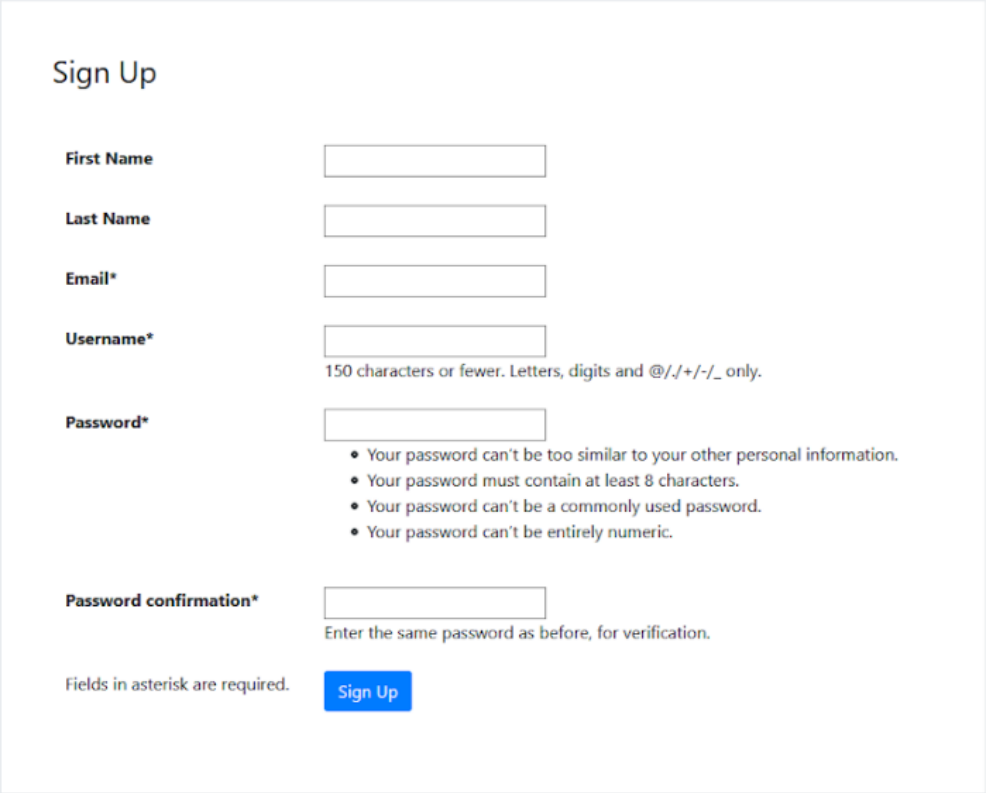

Figure 3.2: The sign up page.

while `django_session(session_key, session_data, expire_date)` persists server-side session state for authenticated users. Using Django's native authentication stack in combination with `django-registration` provides cryptographically secure credential storage, enforced email verification, and minimal persistent state beyond standard Django tables, allowing for not only security, but also easier long term maintenance.

3.3 Interface Gateway and API Layer

Now that we have talked about how accounts are created for end users, let us look at the interface of Cymphony that provides all external access points through which users, applications, and external systems can interact with the platform. To this end, Cymphony exposes a web-based

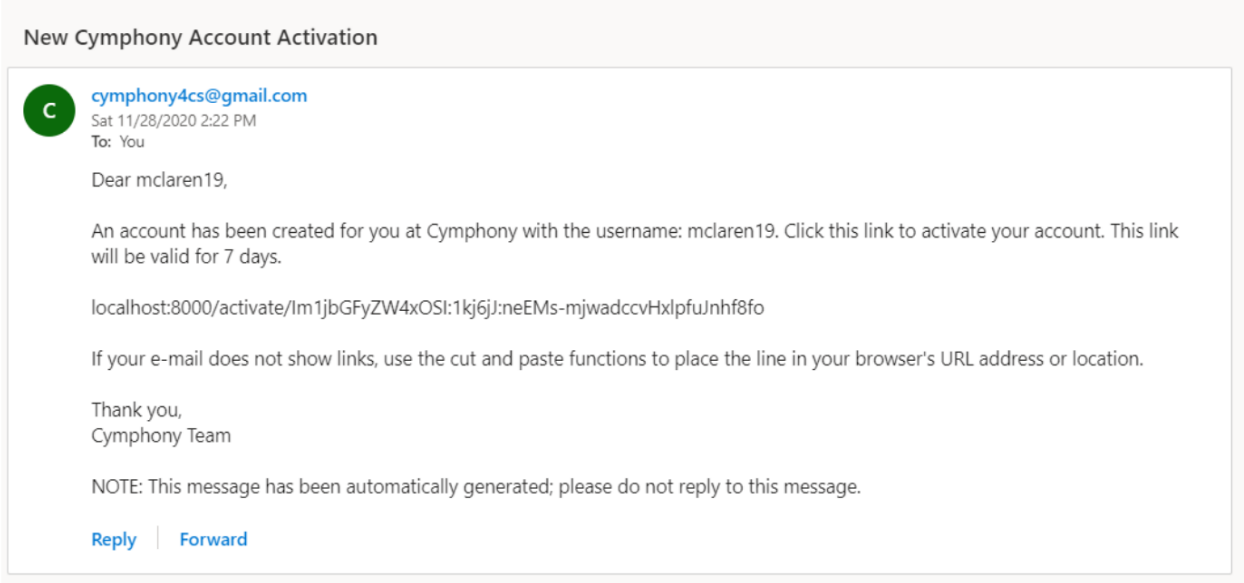

Figure 3.3: The activation email.

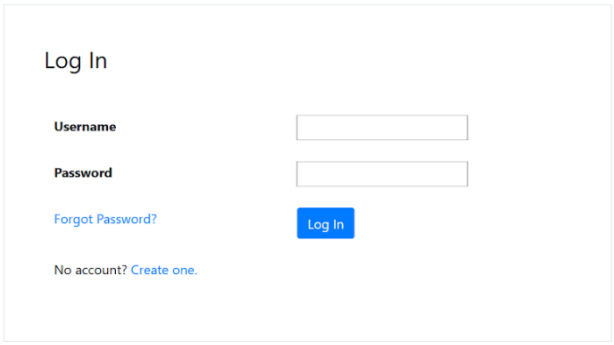

Figure 3.4: The login screen.

graphical user interface (GUI), and a REST-style programmatic API. We first describe the GUI-based interface pertaining to the requester. Second, we describe the GUI-based interface for the worker. Finally, we describe Cymphony's programmatic API.

3.3.1 Requester Interface

We describe how the **requester interface** supports end users in defining, managing, and executing collaborative workflows.

Project Creation Interface: All requester activities are organized under *projects*, which serve as the primary organizational unit in Cymphony. Therefore, a requester begins by creating a *project*. The requester specifies each project by supplying a project name and description. A project groups together workflows, datasets, and execution runs that may belong to the same overall data integration task. There could be multiple such projects under a requester, as shown in Figure 3.5.

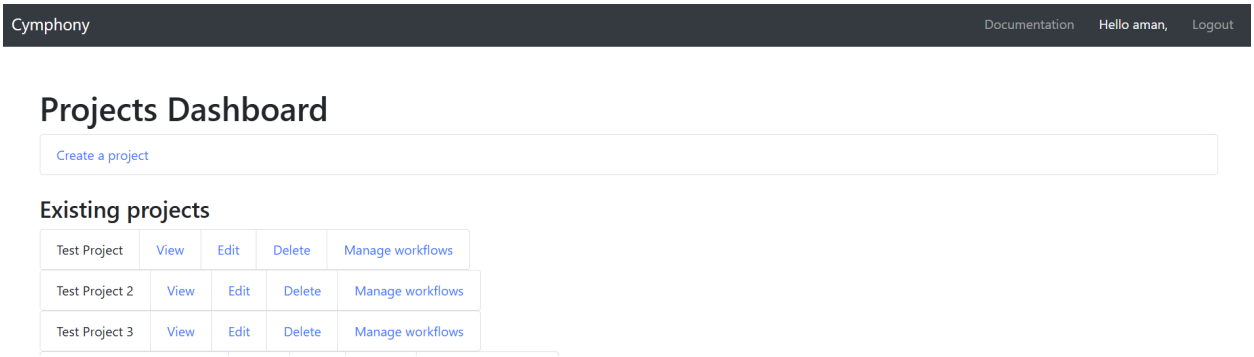

Figure 3.5: Dashboard displaying user projects.

Workflow Creation and Configuration Interface: As shown in Figure 3.6, a requester can have one or more workflows within a project. The requester specifies each workflow by supplying a workflow name and description. As shown in Figure 3.7, the requester also uploads workflow artifacts like the workflow program file (containing the CY program), data file(s), instruction file(s), and/or layout file(s). These artifacts collectively define a *workflow specification*, which is persisted by the system and becomes available for repeated execution.

Run Management Interface: The requester may initiate one or more *runs* for each workflow, as shown in Figure 3.8. For each run, the requester specifies a run name, and description. The requester may also download all materialized output tables and intermediate results from the run, as shown in Figure 3.9.

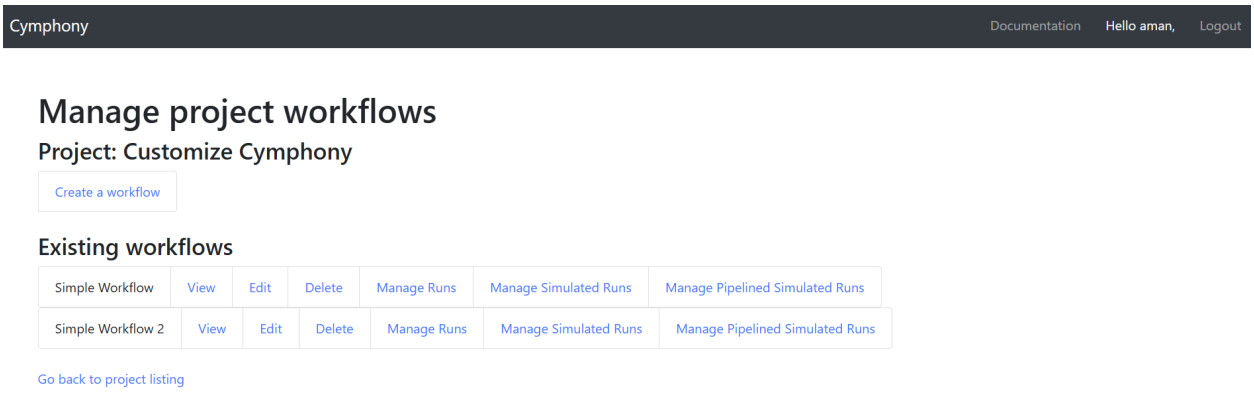

Figure 3.6: Manage workflows in a project.

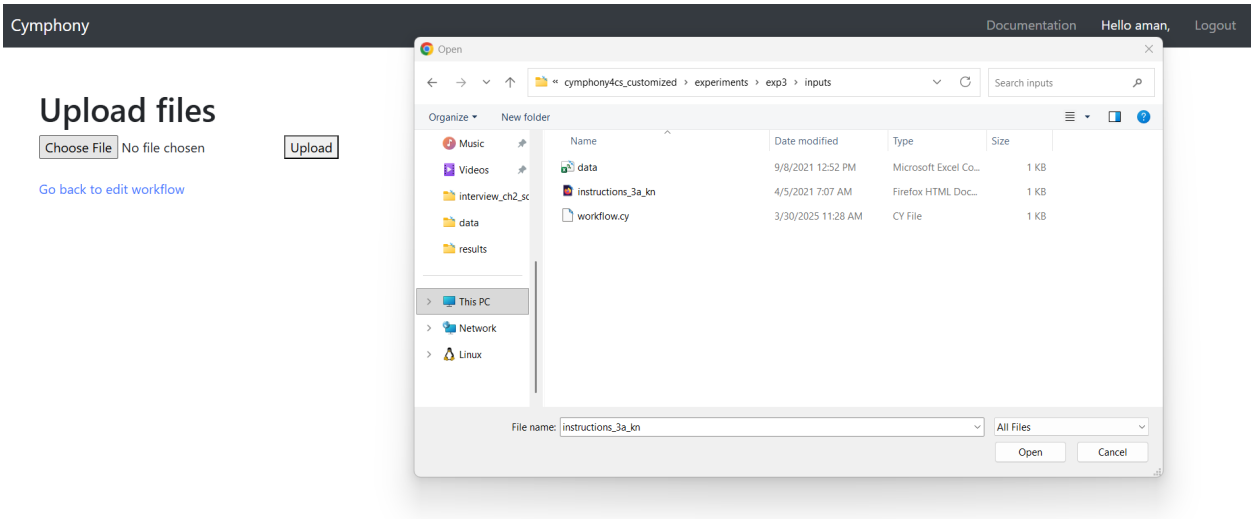

Figure 3.7: Upload files to a workflow.

3.3.2 Worker Interface

The **worker interface** provides the entry point through which human workers interact with Cymphony to complete tasks generated by human jobs in a run. To this end, we describe how workers work on tasks of a human job to bring it to completion.

The user begins by creating an account and authenticating into the system. Upon successful authentication, the user chooses to work as a worker to access the worker interface. The user or let us say worker w, is then directed to a dashboard that displays currently available human jobs to work on. Each job consists of tasks corresponding to a human operator in an active workflow

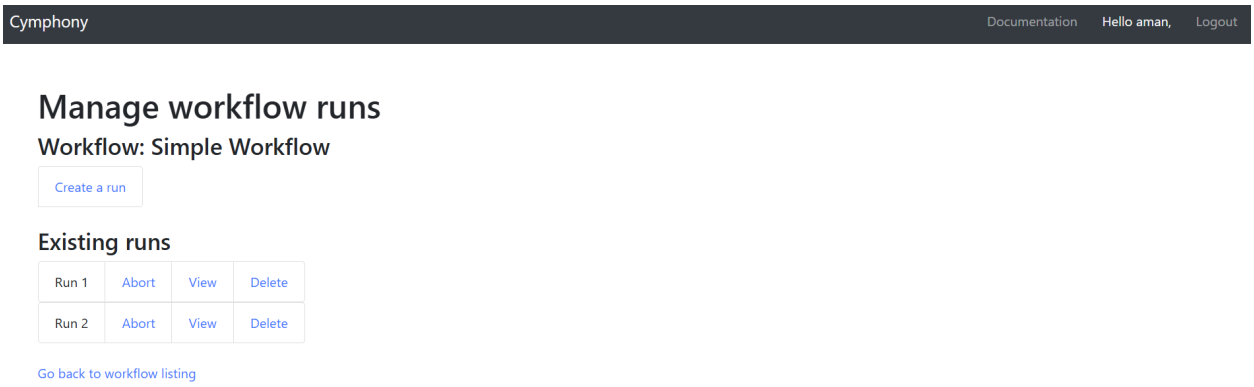

Figure 3.8: Manage runs of a workflow.

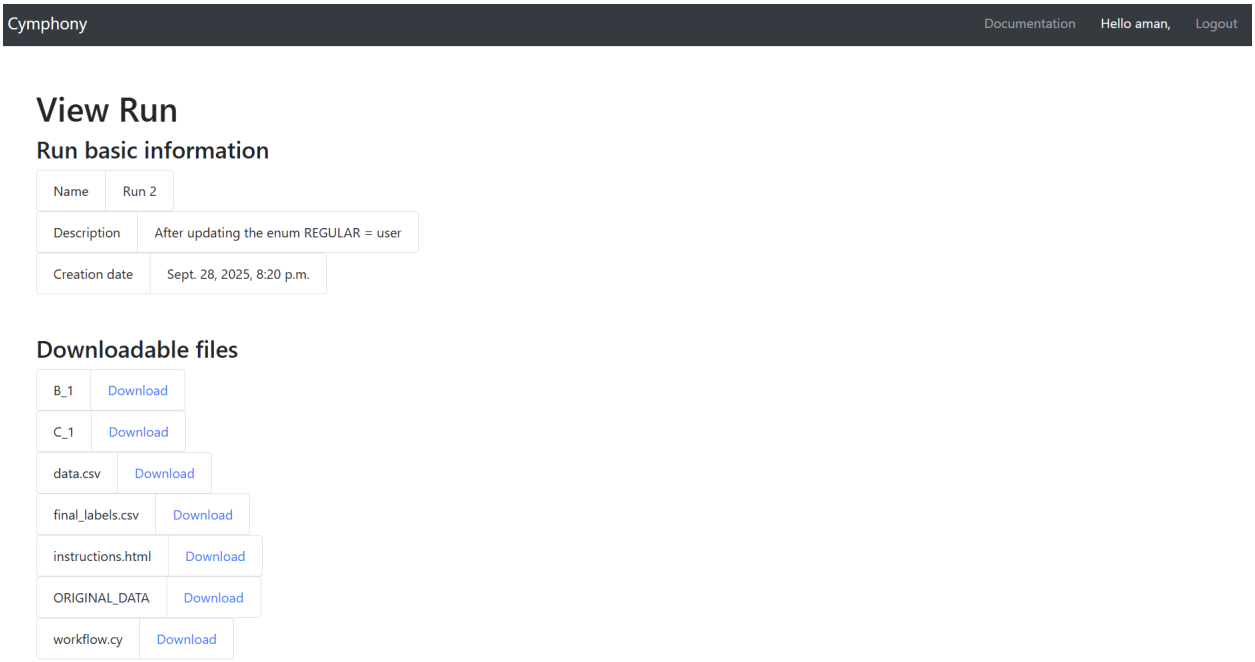

Figure 3.9: Download results from a run.

run. For each job, the dashboard displays the title of the human job as given by its requester, the associated job description, and an option to work on the job.

When w selects a job to work on, the system assigns a task to the worker where each task corresponds to a single tuple in the data file of that human operator. The system then presents an interface similar to Figure 3.10 that displays the tuple or tuple-pair associated with the task, the instruction template provided by the requester, and input widgets for submitting annotations.

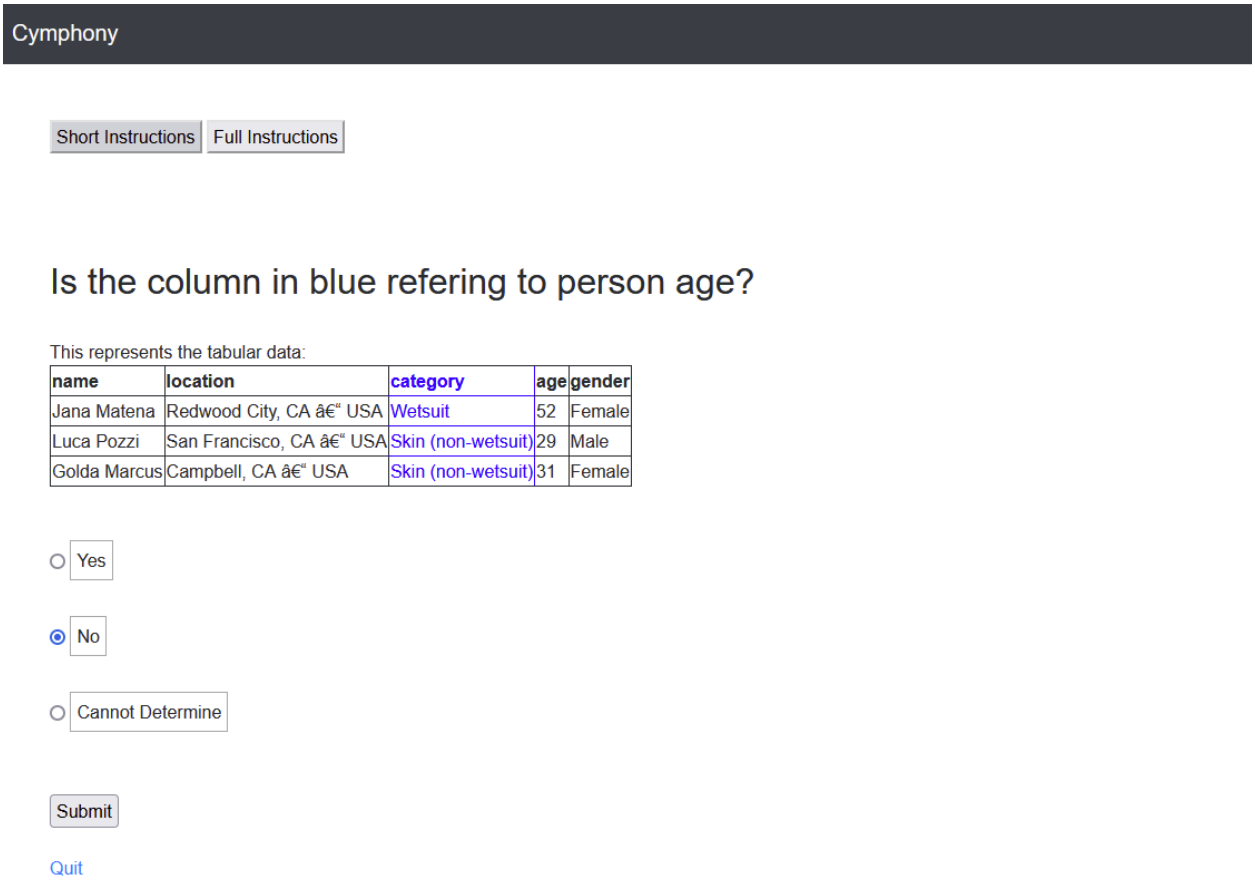

Figure 3.10: Worker interface for annotating tasks in a job.

After completing the task, w submits an annotation through the interface. A new task is then assigned to w if available. There can be two reasons for a task to be unavailable to w. Either all the remaining tasks got picked up by other workers, leaving a temporarily unavailability of tasks for w, or the job is already completed. Depending on the reason, the interface provides a notification to w. In this fashion, these workers keep annotating the tasks belonging to this human job. Eventually, the human job will be complete, and no longer available to work on. Along the way, a worker is also free to skip tasks, or quit working on the human job – in which case they can rejoin the job at a later time if it remains available.

3.3.3 Programmatic API

The **programmatic API** exposes Cymphony's control plane to external applications, enabling third-party systems and scripts to interact with the platform as requesters without relying on the graphical user interface. Through this API, clients can programmatically manage projects and workflows, upload workflow artifacts (e.g., CY programs and input files), initiate and abort workflow runs, and retrieve materialized outputs. To present this interface, we first introduce the endpoint notation used through this section. Second, we summarize the available endpoints in tabular form. Finally, we discuss how these APIs facilitate the integration of Cymphony into larger data processing pipelines and external systems.

Endpoint Notation: We denote the base path of the Cymphony control plane as: `BASE` $\equiv$ `/controller/`. Furthermore, we use the shorthand: `CTRL(category, action, params)` $\equiv$ `BASE/?category=category&action=action` $\|$ `params`, where

1. `category` identifies the target resource class (i.e., `project`, `workflow`, or `run`);
2. `action` specifies the operation to be performed (e.g., create, upload, view, or download);
3. `params` denotes the parameters required to complete the request, such as <`project_id`>, <`workflow_id`>, <`run_id`>, and <`file_name`>.

.

API Endpoints: Table 3.1 summarizes the API endpoints exposed by Cymphony, expressed relative to `BASE` and using the shorthand notation where applicable.

Table 3.1: Programmatic API endpoints exposed by Cymphony.

Method	**Endpoint**	**Functional Description**
POST	`one_step_register/`	Registers a new user without requiring activation.
POST	`login/`	Authenticates a user and initializes a session.

Method	Endpoint	Functional Description
POST	`CTRL(project, create)`	Creates a new project.
POST	`CTRL(workflow, create, pid=<project_id>)`	Creates a workflow under the project identified by `<project_id>`.
POST	`CTRL(workflow, upload_file, pid=<project_id>, wid=<workflow_id>)`	Uploads a file to an existing workflow, identified by the combination of `<project_id>` and `<workflow_id>`.
POST	`CTRL(run, create, pid=<project_id>, wid=<workflow_id>)`	Creates a workflow run, corresponding to the workflow identified by the combination of `<project_id>` and `<workflow_id>`.
GET	`CTRL(run, view, pid=<project_id>, wid=<workflow_id>, rid=<run_id>)`	Retrieves the output file list of a run, identified by the combination of `<project_id>`, `<workflow_id>`, and `<run_id>`.
GET	`CTRL(run, download_file, pid=<project_id>, wid=<workflow_id>, rid=<run_id>, fname=<file_name>)`	Downloads a file corresponding to a run, identified by the combination of `<project_id>`, `<workflow_id>`, `<run_id>`, and `<file_name>`.

Usefulness of the Cymphony API: These endpoints enable Cymphony to be embedded as a programmable component in two primary settings:

1. **Data integration and learning pipelines.** Cymphony can be invoked as a human-in-the-loop module within larger pipelines, such as the active learning workflow described in Section 4.1.3, where entity matching tasks are orchestrated programmatically via the API.

2. **External data processing systems.** The API allows Cymphony to be integrated into third-party platforms and custom data processing systems, enabling external applications to delegate complex crowdsourced data operations to Cymphony (see Chapter 5).

3.4 Data Management Layer

In the previous section, we introduced the interface layer and the logical entities of projects, workflows, runs, and jobs. In this section, we present Cymphony's data management layer, which provides the persistent coordination state and physical storage mechanisms required to execute and manage these entities. We first formalize the execution model that relates projects, workflows, runs, and jobs, and then use these relationships to derive the relational schema that governs workflow execution and worker coordination. Second, we describe the corresponding filesystem namespace used to physically organize workflow artifacts and execution outputs. Finally, we summarize how this hybrid relational–filesystem design enables reproducible, concurrent, and long-running human-in-the-loop workflow executions.

3.4.1 Relational Schema Design

Entity Relationships and Design Motivation: As already introduced, a **project** serves primarily as an organizational unit that groups together a set of workflows.

A **workflow** is analogous to a program or a SQL query — it is a static specification that can be created, edited, and deleted. A user first creates and edits a workflow F. A workflow F is a static object that contains a unique workflow identifier, name, and description. It also contains a CY program defining the workflow DAG, along with the associated CSV input files and instruction templates.

A **run** is analogous to an operating system process or a database query execution — it is a concrete instantiation of a workflow that produces results. At any time after creating the workflow F, the user may execute F, thereby instantiating a new *run*. Each run is associated with metadata

such as a run ID, name, description, status, and creation timestamps. During execution, runs progress through the states `RUNNING`, `COMPLETED`, and `ABORTED`.

Multiple runs may be created from the same workflow and may execute concurrently.

Further, each operator node in the workflow DAG is instantiated as a **job** during a run. Jobs correspond to individual operator executions within a run and progress through the states `IDLE`, `RUNNING`, `COMPLETED`, and `ABORTED`. To support concurrent workflow executions and concurrent worker participation within human jobs, Cymphony's relational catalog is partitioned into global, run-scoped, and job-scoped namespaces.

Global Catalog Tables: This information associated with projects, workflows, runs, and jobs can be naturally captured using a collection of *global catalog tables*. These tables serve as master relations that define the core system entities and maintain their essential metadata. They typically record identifiers, names, descriptions, creation timestamps, and hierarchical parent-child relationships.

- `all_projects(p_id, u_id, p_name, p_desc, date_creation)`
- `all_workflows(w_id, p_id, u_id, w_name, w_desc, date_creation)`
- `workflow_input_files(input_file_path, f_id, w_id, p_id, u_id)`
- `workflow_inst_files(inst_file_path, f_id, w_id, p_id, u_id)`
- `workflow_layout_files(layout_file_path, f_id, w_id, p_id, u_id)`
- `workflow_cy_files(cy_file_path, f_id, w_id, p_id, u_id)`
- `all_runs(r_id, w_id, p_id, u_id, r_name, r_desc, status, date_creation)`
- `all_jobs(j_id, r_id, w_id, p_id, u_id, j_name, j_type, status, date_creation)`

Project- and Workflow-Scoped Tables: We have just described the global catalog tables. Going down to the project level, Cymphony currently does not maintain any project-scoped relations, as

projects serve primarily as lightweight organizational containers for workflows and do not require additional project-specific state. Similarly, no workflow-scoped relations are maintained, since workflow identifiers, metadata, and associated artifacts are already fully captured by the global catalog tables.

Run-Scoped Tables: Going down another level, each workflow execution is assigned its own run-scoped relational namespace. Specifically, each run owns a dedicated namespace prefixed by `ui_pj_wk_rm_`, where `ui`, `pj`, `wk`, and `rm` denote the user, project, workflow, and run identifiers, respectively. This namespace stores the compiled workflow DAG and all variable materializations generated during execution, including:

- Storing DAG metadata in `ui_pj_wk_rm_nodes(n_id, n_name, n_type)`, and `ui_pj_wk_rm_edges(src_id, dst_id)`;
- Storing variable materialization tables, which persist operator outputs at the run scope. For example, executing $(B, C) = \texttt{3a\_kn}(T, \texttt{"instruction1.html"}, k = 2, n = 3)$ materializes the tables `ui_pj_wk_rm_b(task_id, worker_id, annotation)` and `ui_pj_wk_rm_c(task_id, label)`.

Job-Scoped Tables: While the outputs of a completed job are materialized as run-scoped tables, the execution of an individual job requires additional state at a finer level to coordinate concurrent worker interactions and aggregation. Cymphony therefore isolates each job execution using a set of job-scoped relational tables. For example, during the execution of a `3a_kn` job, the system materializes the following job-scoped tables:

- `ui_pj_wk_rm_jn_config_parameters(key, value, value_data_type)`, which stores operator configuration parameters (e.g., $k = 2$, $n = 3$);
- `ui_pj_wk_rm_jn_tasks(task_id, total_assigned, abandoned, pending_annotations, done)`, which stores task-level metadata and supports coordination of concurrent worker arrivals;

- `ui_pj_wk_rm_jn_assignments(task_id, worker_id, status)`, which records task assignments to workers;
- `ui_pj_wk_rm_jn_outputs(task_id, worker_id, annotation)`, which persists raw annotations submitted by workers;
- `ui_pj_wk_rm_jn_final_labels(task_id, label)`, which stores aggregated labels produced so far.

Relational Schema Summary and Identifier Conventions: The persistent coordination state maintained in Cymphony's relational catalog enables the system to pause workflow executions for human input, to coordinate concurrent worker annotations on the same job, and to resume execution deterministically. By isolating each run into its own relational namespace, the schema also supports concurrent workflow executions while preventing coordination hotspots.

Cymphony follows hierarchical identifier conventions rather than globally unique identifiers. Specifically, projects, workflows, runs, and jobs are identified within the context of their parent entities. For example, a run is uniquely identified by the composite key (u_id, p_id, w_id, r_id), which encodes that the run belongs to a specific workflow within a specific project created by a specific user. This design keeps execution contexts self-contained, minimizes cross-component dependencies, and simplifies code portability, comprehension, and long-term maintainability and transferability.

3.4.2 Filesystem Namespace

Having described the relational namespace organization and underlying table schemas, we now describe how these abstractions are reflected in the filesystem.

Cymphony mirrors the relational namespace into a directory hierarchy of the form: `$HOME/u<u_id>/p<p_id>/w<w_id>/r<r_id>/`. Files uploaded by the requester, such as input CSV files and instruction templates (e.g., `data1.csv`, `instruction1.html`), are stored at the workflow level. When the requester initiates a workflow execution, these files are copied into the corresponding run directory. Files generated during workflow execution, such as materialized

intermediate tables and final output files (e.g., `final_labels1.csv`), are created within the run-level directory. Upon completion of a run, all files in the run directory are made available to the requester for inspection and download through the run dashboard.

This file-based organization provides a lightweight mechanism for isolating concurrent workflow executions and for durably persisting intermediate and final results across long-running runs.

Summary: Together, Cymphony's relational catalog and run-scoped filesystem namespaces form a hybrid data management layer that provides durable coordination state and isolated physical storage for each workflow execution. This design enables reproducible executions, supports long-running human-in-the-loop workflows with pause–resume semantics, and allows multiple concurrent runs and concurrent worker annotations to proceed safely and deterministically.

3.5 Core Orchestration and Execution Layer

This section describes the core runtime components that implement the logic to create, orchestrate, and execute crowdsourcing workflows in Cymphony. We first describe the *Requester Manager*, which manages workflow lifecycle and execution initialization. Second, we describe the *Parser*, which compiles workflow specifications into DAG representations. Third, we present the *Optimizer*, which linearizes and prepares workflows for execution. Fourth, we describe the *Executor*, which drives runtime execution. Finally, we describe the *Worker Manager*, which coordinates human task execution and annotation aggregation.

3.5.1 Requester Manager

The **Requester Manager** implements the logic for requester-driven workflow lifecycle management. It acts as the execution coordinator, mediating between the requester-facing interfaces and the internal workflow execution engine.

Upon a project creation request issued through the requester interface, the Requester Manager allocates a project identifier, inserts a new record into the `all_projects` table, and creates a corresponding project directory at `$HOME/u<u_id>/p<p_id>/`.

For each workflow created under a project, the Requester Manager allocates a workflow identifier, inserts a record into the `all_workflows` table, registers the uploaded workflow artifacts into `workflow_cy_files`, `workflow_input_files`, and `workflow_inst_files`, while uploading them under a newly created workflow directory at `$HOME/u<u_id>/p<p_id>/w<w_id>/`.

For each run initiation request, the Requester Manager allocates a run identifier, inserts a row into the `all_runs` table, creates a run directory at `$HOME/u<u_id>/p<p_id>/w<w_id>/r<r_id>/`, copies the workflow-level artifacts into the run directory, and invokes the Parser to compile the CY program into a DAG.

Within the codebase, business logic and data access operations are physically separated to improve modularity and maintainability. Together, these steps ensure that each workflow specification and each of its executions are durably registered in the catalog and isolated in a deterministic filesystem namespace before execution begins.

3.5.2 Parser

The **Parser** is Cymphony's workflow compiler. It translates a requester-provided workflow specification into a validated, persisted, and executable internal representation. Specifically, it parses a CY program and produces a directed acyclic graph (DAG) whose nodes represent operator instances and whose edges encode dataflow dependencies.

At the implementation level, the Parser accepts a workflow specification copied into the run directory—consisting of a CY program file and associated artifacts—and produces a validated workflow DAG $G = (V, E)$, where each $v \in V$ corresponds to an operator instance and each $(u, v) \in E$ denotes a data dependency.

The CY program is first tokenized and parsed into an intermediate representation, which is subsequently validated for syntactic correctness, supported operator types (e.g., `exec_sql`, `3a_kn`, `3a_amt`), valid variable references, and valid references to run-scoped artifacts such as CSV and instruction files. Operator-specific parameter constraints are also validated at this stage.

Each operator declaration is then mapped to a unique operator instance (job) and assigned a node identifier. Directed edges are created between operator nodes based on declared input and output variables, yielding a directed acyclic graph that captures the workflow's dataflow semantics.

The resulting DAG is materialized into run-scoped catalog tables `ui_pj_wk_rm_nodes` and `ui_pj_wk_rm_edges`. The `nodes` table stores one tuple per operator instance (job), while the `edges` table encodes directed dataflow dependencies between operator instances. Upon successful compilation, the Parser returns control to the Requester Manager, which forwards the compiled DAG to the Optimizer.

3.5.3 Optimizer

The **Optimizer** transforms the logical workflow DAG produced by the Parser into a concrete execution plan consumable by the runtime engine. It derives a deterministic, job-level schedule that specifies the order in which operator instances are to be instantiated and executed. Formally, given a workflow DAG $G = (V, E)$, the Optimizer produces a linear execution plan $L = \langle j_1, j_2, \ldots, j_{|V|} \rangle$, where each $j_i \in V$ is a job corresponding to an operator instance, and the ordering respects all data dependencies encoded by E.

Since a workflow DAG may admit multiple valid topological orderings, the Optimizer selects one such ordering and materializes it as the execution plan. Formally, the produced ordering L satisfies the property that for every edge $(u, v) \in E$, the job corresponding to u appears before the job corresponding to v in L.

At the implementation level, Cymphony employs Kahn's algorithm to derive a topological ordering of the workflow DAG, as shown in Algorithm 3.1. The Optimizer maintains a set of *root nodes*—operators with no incoming dependencies—repeatedly selects and removes one such node, appends it to the linearized schedule, and deletes its outgoing edges. Whenever removing an edge causes a downstream node to have no remaining incoming edges, that node becomes eligible and is added to the root set. If edges remain after the process terminates, the graph contains a cycle and the workflow specification is rejected. Given constant-time access to adjacency lists and indegree maintenance, the linearization runs in $O(|V| + |E|)$ time.

Algorithm 3.1 Linearize a DAG using Kahn's algorithm (Optimizer)

Input : Directed graph $G = (V, E)$

Output : A topological ordering L of V (or **error** if G contains a cycle)

```
procedure KahnTopologicalSort(G = (V, E))
    L ← [ ]
    R ← { v ∈ V | indegree(v) = 0 }                    ▷ root nodes
    while R ≠ ∅ do
        remove any node n from R
        append n to L
        for m such that (n, m) ∈ E do
            remove edge (n, m) from E
            if indegree(m) = 0 then
                add m to R
            end if
        end for
    end while
    if E ≠ ∅ then
        error                                          ▷ G contains a cycle
    end if
    return L
end procedure
```

The Optimizer materializes the derived linear execution plan in a run-scoped catalog table named `ui_pj_wk_rm_nodes_execution_order(n_id, position)`. This relation stores, for each job node, its execution position within the linearized schedule. Here, `position` denotes the total order index assigned by the Optimizer, with smaller values indicating earlier execution. Persisting the execution order allows the Executor to resume and deterministically coordinate long-running workflow executions using database state. The resulting execution plan L, materialized in `ui_pj_wk_rm_nodes_execution_order`, serves as the authoritative static schedule consumed by the Executor during runtime.

3.5.4 Executor

The **Executor** interprets the persisted linear execution plan produced by the Optimizer and coordinates the execution of both automatic and human jobs while maintaining durable run- and job-level execution state. For a given run, the Executor loads the execution plan L from the run-scoped table `ui_pj_wk_rm_nodes_execution_order` and traverses jobs sequentially according to their assigned `position`. Each job j_i transitions through the state sequence `IDLE` $\rightarrow$ `RUNNING` $\rightarrow$ `COMPLETED` (or `ABORTED` upon failure).

If j_i corresponds to an automatic operator (e.g., `exec_sql`, `sample_random`), the Executor updates the job state in `all_jobs` to `RUNNING`, executes the corresponding operator logic, materializes its output as run-scoped relational tables and run-directory files, and then updates the job state to `COMPLETED`.

If, however, j_i corresponds to a human operator (e.g., `3a_kn`, `3a_amt`), the Executor initializes job-scoped coordination tables, publishes tasks via the Worker Manager or external marketplace integration layer, updates the job state in `all_jobs` to `RUNNING`, and suspends traversal of L until the human job completes (see Section 3.5.5). Upon job completion, the Executor materializes run-scoped output tables and updates the job state to `COMPLETED`.

While a human job is in the `RUNNING` state, the Executor does not advance to subsequent jobs. Once the job completes, traversal resumes from the next job in the persisted execution order. This design enables long-running, pause–resume workflow execution while preserving deterministic DAG semantics.

If all jobs in L complete successfully, the Executor updates the run state in `all_runs` to `COMPLETED`. If any job fails, the Executor marks that job and all remaining `IDLE` or `RUNNING` jobs as `ABORTED` and updates the run state to `ABORTED`.

Together, these mechanisms allow Cymphony to reliably execute hybrid workflows that interleave automatic computation and human-in-the-loop processing while maintaining deterministic and durable execution semantics.

3.5.5 Worker Manager

The **Worker Manager** implements Cymphony's in-house human computation runtime. It coordinates all interactions between the Executor and authenticated human workers, including task publication, assignment, annotation collection, aggregation, and job completion.

The Worker Manager accepts execution requests for human jobs from the Executor, as well as annotation submissions and task abandonment events from workers. It produces persisted per-worker annotation records, aggregated task labels, and job completion notifications that are forwarded back to the Executor.

For each active human job, the Worker Manager first decomposes the job's input table into one task per tuple. It then materializes job-scoped coordination tables in the namespace `ui_pj_wk_rm_jn_` and initializes the task metadata relation `ui_pj_wk_rm_jn_tasks(task_id, total_assigned, abandoned, pending_annotations, done)`. It then exposes available tasks via the worker interface, where it assigns tasks to authenticated workers while recording assignments in `ui_pj_wk_rm_jn_assignments(task_id, worker_id, status)`.

When a worker submits an annotation, the Worker Manager persists the raw annotation in `ui_pj_wk_rm_jn_outputs(task_id, worker_id, annotation)`, updates counters in `ui_pj_wk_rm_jn_tasks`, and triggers aggregation checks.

For each task, the Worker Manager accumulates annotations until the operator-defined consensus criteria are satisfied (e.g., k agreeing votes out of n total submissions). Once satisfied, the Worker Manager computes and persists the aggregated label in `ui_pj_wk_rm_jn_final_labels(task_id, label)` and marks the task as `COMPLETED` in `ui_pj_wk_rm_jn_tasks`.

A human job is considered `COMPLETED` once all of its tasks have produced final aggregated labels (including undecided outcomes when consensus is not reached). Upon job completion, the Worker Manager updates the job state in `all_jobs` and notifies the Executor. The Executor then resumes traversal of the linear execution plan L.

3.6 Crowd Marketplace Integration Layer

This section describes how Cymphony integrates with external crowd marketplaces to obtain human annotations beyond the in-house worker pool. This layer embeds third-party marketplaces directly into the workflow runtime while preserving Cymphony's native assign–annotate–aggregate execution semantics for human jobs. Cymphony currently supports Amazon Mechanical Turk (AMT) through a dedicated human operator, `3a_amt`, and an integration manager, which together enable workflows to scale to large and diverse external worker populations. We first review the `3a_amt` operator and then describe the AMT Integration Manager (AMT Manager).

3.6.1 The `3a_amt` Human Operator

AMT integration is realized by introducing the specialized human operator `3a_amt` into the workflow language (see Chapter 2). The operator is semantically analogous to the in-house `3a_kn` operator but delegates task assignment and annotation to AMT. The operator has the form $(B, C) = 3a\_amt(I, Instr, k, n, Qual, Price)$, where I is the input table, $Instr$ is the instruction template shown to AMT workers, k and n specify majority-voting parameters, and $(Qual, Price)$ configure AMT-specific worker qualifications and compensation. Raw worker annotations are materialized into table B, and final aggregated labels are written to table C.

3.6.2 AMT Manager

The **AMT Manager** is Cymphony's external human computation runtime. It is invoked by the Executor whenever a `3a_amt` job becomes active and implements the same task decomposition and annotation persistence pipeline as the in-house Worker Manager, but executes them through the AMT API, and adopts slightly different aggregation semantics due to the bounded nature of AMT. For each task t, the AMT Manager solicits *exactly* n annotations. If at least k of these annotations agree, that value is returned as the aggregated label. Otherwise, the task is assigned the special value `null` (undecided).

When the Executor encounters a `3a_amt` job, it delegates execution to the AMT Manager, which proceeds as follows. The job's input table is decomposed into one task per tuple, after which job-scoped coordination tables are initialized. Tasks are translated into AMT HITs and published using the AMT API. The AMT Manager periodically polls AMT for completed assignments. Retrieved worker annotations are persisted into job-scoped output tables, after which aggregation is applied using the above semantics. When all tasks have produced aggregated final labels, the AMT Manager marks the job as `COMPLETED` and notifies the Executor.

For each `3a_amt` job, the AMT Manager materializes a job-scoped relational namespace that mirrors the in-house Worker Manager's coordination schema while enabling asynchronous population through the AMT API:

- `ui_pj_wk_rm_jn_amt_config_parameters(key, value, value_data_type)`, which stores the configuration parameters supplied in the `3a_amt` operator specification.
- `ui_pj_wk_rm_jn_amt_tasks(task_id, hit_id, hit_group_id, num_responses_submitted, max_responses, done)`, which tracks individual tasks belonging to the job and their AMT-level execution state.
- `ui_pj_wk_rm_jn_amt_assignments(task_id, amt_worker_id, amt_assignment_id)`, which persists task assignments to AMT workers.
- `ui_pj_wk_rm_jn_amt_outputs(task_id, amt_worker_id, annotation)`, which persists raw annotations submitted by AMT workers.
- `ui_pj_wk_rm_jn_amt_final_labels(task_id, label)`, which stores final aggregated labels for each task in the job.

By embedding the AMT Manager into the Executor's scheduling loop, Cymphony supports workflows that seamlessly interleave in-house and external human computation within a single unified execution plan.

3.7 An Illustrative Run

Having described each architectural layer in isolation, we now present an illustrative end-to-end execution of a complex Cymphony workflow to demonstrate how these components interact during runtime.

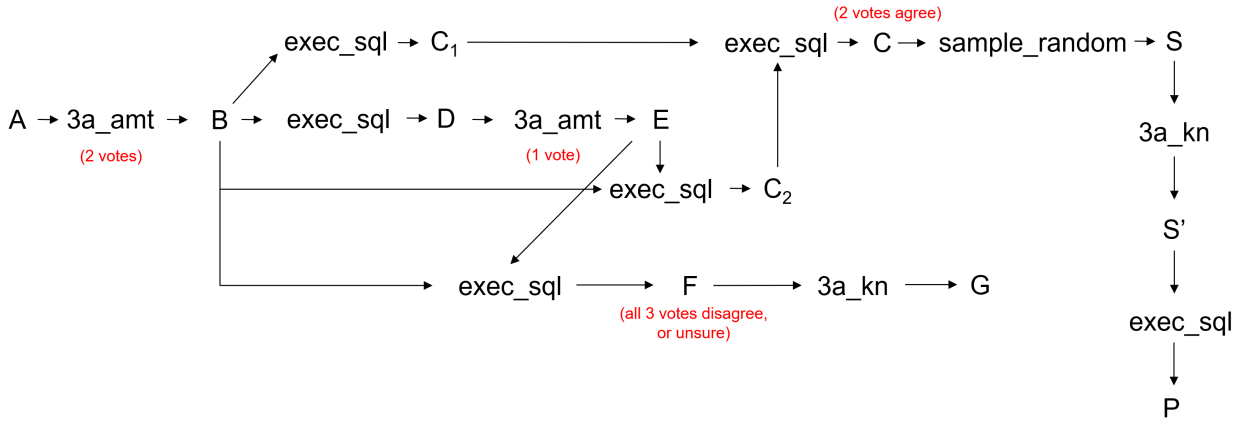

Figure 3.11: An illustrative complex Cymphony workflow.

Figure 3.11 shows a representative workflow consisting of both automatic and human operators. It corresponds to the color-extraction workflow with precision estimation previously introduced in Chapter 2, but here we focus on the implementation semantics rather than the conceptual details.

A requester specifies the workflow as a Cymphony program and submits an execution request to the system. Cymphony creates a new `Run` instance to represent this execution and initializes a dedicated run directory to isolate all generated artifacts. The program is parsed into a workflow DAG, which is persisted in run-scoped catalog tables. The Optimizer then computes and persists a linear execution order over the DAG nodes. The Executor loads this persisted execution order and begins deterministic traversal of the plan.

At the start of execution, only the first `3a_amt` operator is eligible for execution. The Executor delegates this job to the AMT Manager, suspending traversal of the execution plan while tasks are published to AMT and worker annotations are asynchronously collected. Once each tuple in

table A has accumulated the required number of annotations, the AMT Manager aggregates the results according to the operator's k/n policy, materializes the aggregated output, and produces run-scoped table B. It then marks the job as `COMPLETED`, and notifies the Executor. The Executor then resumes traversal from the next job in the persisted execution order.

Subsequent automatic operators (e.g., `exec_sql`) are executed immediately, with their outputs materialized as run-scoped tables. When a subsequent human operator is encountered—either `3a_amt` or `3a_kn`—the Executor again suspends traversal and delegates coordination to the appropriate human computation runtime (AMT Manager or Worker Manager). In the case of `3a_kn`, arriving workers are dynamically assigned tasks and annotations are collected and aggregated until all tuples have produced final labels.

This process continues until all jobs in the persisted execution order have completed. At that point, the Executor marks the run as `COMPLETED`, making all final outputs available to the requester.

3.8 System Maturity and Deployment Status

Having described Cymphony's architecture and end-to-end execution semantics, we briefly summarize the system's implementation maturity, codebase scale, and deployment history. This context helps position Cymphony not merely as a conceptual research prototype, but as a fully implemented and deployed crowdsourcing platform.

Cymphony is implemented as a modular, web-based system using Python and the Django framework. At the time of writing, the implementation comprises approximately **16,816 lines of Python code** spanning the interface layer, workflow compiler, execution engine, human computation runtimes, and data management layer described in this chapter. In total, the codebase contains **69,716 lines of code**, including Python, HTML, JavaScript, CSS, and Markdown files. The system is organized into logically separated modules, with clear separation between interface components, business logic, and persistent data access, facilitating extensibility and long-term maintenance.

Cymphony is released as an open-source system to support reproducibility, extensibility, and community-driven development. The full source code, along with documentation, example workflows, and deployment instructions, is publicly available at `https://github.com/`

anhaidgroup/cymphony. In addition to open-source release, Cymphony has been deployed in production at the Qatar Computing Research Institute (QCRI) since April 2022, where it has been used to support ongoing data integration and curation research.

Beyond this initial deployment, Cymphony has also served as the basis for subsequent system extensions. As discussed in Chapter 5, the platform has been customized to support Smartcat, a data catalog system requiring role-aware curation, drive-by annotation, and API-based integration. This customized variant is publicly available at https://github.com/saini5/cymphony4cs_customized and has been used in academic research settings.

Together, these deployments and extensions demonstrate that Cymphony is a mature, extensible system that has evolved beyond a single-use research prototype into a reusable crowdsourcing execution platform.

Chapter 4

Evaluating Cymphony

In this chapter, we evaluate Cymphony along three dimensions: (i) its ability to express and execute realistic crowdsourcing workflows for data integration (DI), (ii) the accuracy and cost achieved on real datasets with real workers, and (iii) the scalability of the runtime as dataset size, worker parallelism, and machine resources increase.

We organize the evaluation in two parts. First, we present three real DI case studies: information extraction from text, column classification in a data catalog, and entity matching via active learning. Second, we present a suite of simulated scaling experiments using synthetic workers to stress-test the execution engine under controlled workloads.

4.1 Applying Cymphony to DI Problems

A key motivation for Cymphony is that practical crowdsourcing solutions for data integration (DI) often require *multi-stage* control flow, including obtaining redundant labels, branching on agreement or disagreement, sampling for quality estimation, escalating ambiguous cases to expert workers, and materializing intermediate results for downstream processing.

This section demonstrates the versability of Cymphony by showing that it can naturally capture and execute such complex crowdsourcing workflows. We further show that Cymphony achieves these capabilities while saving significant user effort and attaining state-of-the-art accuracy at a reasonable cost.

To this end, we present three experiments, across which we report workflow decomposition into jobs and tasks, crowdsourcing payments, the total number of labels collected, and task-level accuracy (or precision/recall).

4.1.1 Color Extraction from Product Titles

We begin with a color extraction DI task that illustrates how Cymphony can be used to implement and execute a realistic multi-stage crowdsourcing workflow.

Color Extraction as a Crowdsourcing Workflow: The color extraction task is defined as follows: given a product title, the goal is to extract the color of the product (e.g., *Gray*, *Blue*), as illustrated in Figure 4.1.

Product Title
”Details about iPhone 6 Plus – 64GB- Gray Smartphone”@en
”Details about Huawei Google Nexus 6P H1512 64GB 5.7 inch – Blue”@en
...

(a) Input

Product Title	Final Label
”Details about iPhone 6 Plus – 64GB- Gray Smartphone”@en	Gray
”Details about Huawei Google Nexus 6P H1512 64GB 5.7 inch – Blue”@en	Blue
...	...

(b) Expected Output

Figure 4.1: Input and expected output for extracting color from product titles.

Based on these requirements, we design the following multi-stage crowdsourcing workflow:

1. For each product title, collect two independent color annotations from AMT workers.
2. If the two annotations disagree, request an additional annotation from AMT.
3. Let C denote the set of titles for which at least two annotations agree.
4. Let F denote the set of remaining titles for which all three annotations disagree.
5. Sample a subset $S \subseteq C$ and send these titles to data stewards for re-annotation, producing S', which is used to estimate the labeling precision over C.
6. Send all titles in F to data stewards for resolution, producing the set G.

7. Form the final output by combining: (i) titles in $C \setminus S$, (ii) steward-labeled titles in S', and (iii) steward-resolved titles in G.

Using Cymphony to Solve Color Extraction: We now describe how the color extraction workflow is implemented in Cymphony. Figure 4.2 shows the workflow expressed using Cymphony operators. This workflow is then specified as a CY program, excerpts of which are shown in Figures 4.3 and 4.4.

At a high level, Figure 4.3 implements the initial annotation and disagreement resolution stages, while Figure 4.4 implements the final result assembly. Since the complete CY program consists of approximately 300 lines, we omit the middle portion for brevity and instead summarize the program's semantics below.

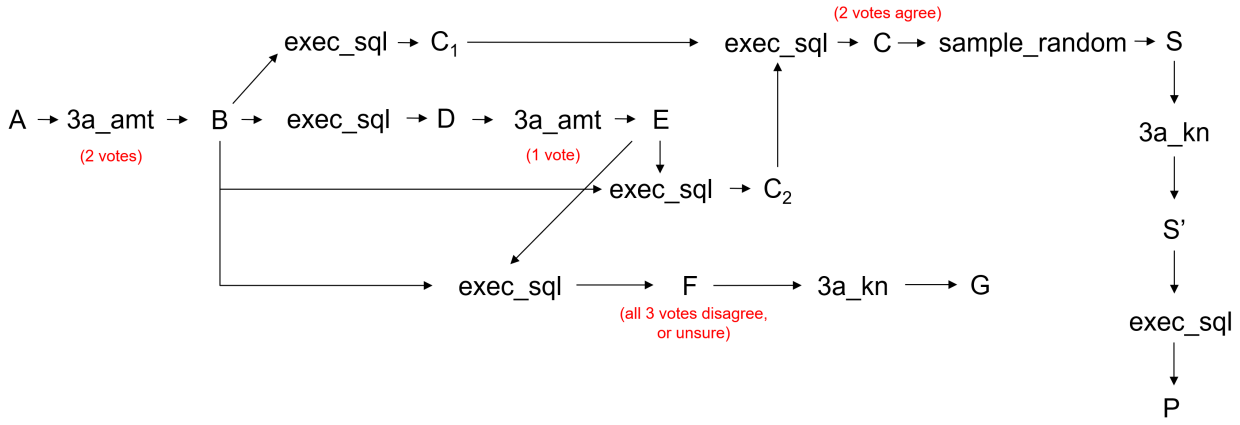

Figure 4.2: Color extraction workflow modeled using Cymphony operators.

1. The input dataset of product titles is denoted as `ORIGINAL_DATA`.

2. `ORIGINAL_DATA` is submitted as an AMT job that collects two annotations per tuple, producing tables `B_1` and `C_1`.

3. Based on `C_1`, tuples are partitioned into:

workflow.cy

```
/*
    real world workflow involving color extraction via free_text response.
*/

/* read data */
ORIGINAL_DATA = read_table("data.csv");

/* take two annotations per tuple from amt workers */
(B_1, C_1) = 3a_amt(
    ORIGINAL_DATA,
    "instructions.html",
    title="Identify product color from the product title",
    description="Data provided in tabular format represents the title of a product. Identify the color from the product title.",
    keywords=["phone", "mobile", "classification", "extraction", "color", "product", "table", "text"],
    lifetime=2592000,
    question="Look at the tabular data and please consult instructions. Type the color as it appears in the product title.",
    answers="free_text",
    k = 2,
    n = 2,
    annotation_time_limit = 3600,
    reward_per_hit=0.03,
    auto_approve_and_pay_workers_in=172800,
    tasks_per_hit=2,
    publish_to_sandbox=False,
    workers_are_masters=True
);

/* queries to attach tuples to the labeled ids and separate out tuples that need more annotations */
(
    AGREEMENTS_WITH_LABELS,
    DISAGREEMENTS_WITHOUT_LABELS
) = exec_sql(
    ORIGINAL_DATA,
    C_1,
    queries = "
        CREATE TABLE temp as (
            SELECT _id,product_title,label
            FROM ORIGINAL_DATA
            INNER JOIN C_1
            USING(_id)
        );
        CREATE TABLE agreements as (
            select * from temp where label!='undecided'
        );
        CREATE TABLE disagreements as (
            select * from temp where label='undecided'
        );
```

Figure 4.3: CY File for Color Extraction (start).

   (a) `DISAGREEMENTS_WITHOUT_LABELS`, containing tuples whose aggregated label is `undecided`;

   (b) `AGREEMENTS_WITH_LABELS`, containing tuples whose aggregated label is determined.

4. `DISAGREEMENTS_WITHOUT_LABELS` is resubmitted to AMT to collect one additional annotation per tuple, producing `B_2` and `C_2`.

5. `B_1` contains two annotations per tuple, while `B_2` contains a third annotation for tuples that previously disagreed.

6. For each tuple x in `B_2`:

```
(E) = exec_sql(
    TITLES_WITH_COLORS,
    S_WITH_AMT_LABELS,
    S_WITH_CYMPHONY_LABELS,
    D_WITH_CYMPHONY_LABELS,
    queries="
        CREATE TABLE amt_labeled_titles as (
            SELECT
                _id,product_title, label
            FROM
                TITLES_WITH_COLORS
            EXCEPT
            SELECT
                _id,product_title, label
            FROM
                S_WITH_AMT_LABELS
        );
        CREATE TABLE output as (
            SELECT
                _id,product_title, label
            FROM
                amt_labeled_titles
            UNION
            SELECT
                _id,product_title, label
            FROM
                S_WITH_CYMPHONY_LABELS
            UNION
            SELECT
                _id,product_title, label
            FROM
                D_WITH_CYMPHONY_LABELS
        );
    ",
    mapping_to_output_variables=[
    "amt_labeled_titles:None",
    "output:E"
    ]
);

/* write output to file */
write_table(E, file="final_labels.csv");
```

Figure 4.4: CY File for Color Extraction (end).

(a) If the annotation in B_2(x) matches either annotation in B_1(x), that value is selected as the final label;

(b) Otherwise, if all three annotations differ, the label for x is set to `undecided`.

7. The resulting fully labeled dataset is denoted as DATA_WITH_AMT_LABELS.

8. This dataset is partitioned into:

   - TITLES_WITH_COLORS, containing tuples with determined color labels;
   - TITLES_WITHOUT_COLORS, containing tuples labeled `undecided` or `Cannot Determine`.

9. From TITLES_WITH_COLORS, a sample of 100 tuples (S_WITH_AMT_LABELS) is selected for quality estimation:
   (a) Labels are removed to form S_WITHOUT_LABELS.
   (b) These tuples are re-annotated by trusted stewards with exactly one annotation per tuple via the 3a_kn operator, producing S_WITH_CYMPHONY_LABELS.
   (c) AMT precision is computed by comparing S_WITH_AMT_LABELS and S_WITH_CYMPHONY_LABELS.
10. All tuples in TITLES_WITHOUT_COLORS are sent to trusted stewards for exactly one annotation per tuple via 3a_kn, producing D_WITH_CYMPHONY_LABELS.
11. The final result table E is formed as: E = D_WITH_CYMPHONY_LABELS $\cup$ S_WITH_CYMPHONY_LABELS $\cup$ (TITLES_WITH_COLORS $\setminus$ S_WITH_AMT_LABELS).
12. E is written to final_labels.csv and constitutes the complete labeled dataset.

Dataset Preparation: We begin with the *GS for Product Matching and Product Extraction* dataset available at: http://data.dws.informatik.uni-mannheim.de/productcrawl/product-fextraction-gold-standard/data/all_labelled_entities.json. We preprocess this dataset by extracting product titles from the JSON source, yielding an initial set of 460 records. We then apply the following cleaning steps/transformations:

1. We normalize entries containing inch symbols (e.g., converting *5.5"* to *5.5 inch*) to avoid CSV parsing errors caused by embedded quotation marks.
2. We remove nested quotation marks to prevent malformed CSV rows. For example, *"Details about Apple iPhone 4S 16GB "Factory Unlocked" Black and White Smartphone"@en* is converted to *"Details about Apple iPhone 4S 16GB Factory Unlocked Black and White Smartphone"@en*.
3. We split lines containing multiple product titles into separate records.

The cleaned dataset consists of 522 product titles spanning categories such as mobile phones, headphones, and televisions.

Cost and Accuracy: We collect two AMT annotations for each of the 522 product titles. Among these, only 86 titles receive disagreeing annotations, requiring one additional AMT annotation each. This results in a total of $2 \times 522 + 86 = 1130$ AMT labels. At a rate of \$0.03 per label, the total AMT cost is \$33.90.

For in-house validation, we request one annotation per title for 105 titles: 100 sampled titles for quality estimation and 5 titles (less than 1% of all titles) whose AMT annotations were undecided. Thus, a total of 105 in-house annotations are collected.

The workflow achieves high labeling accuracy at modest cost:

- On the 100-title sample, AMT labels agree with in-house steward labels on 99 titles.
- Using this sample to estimate AMT performance over all 522 titles, the estimated AMT precision is 99%, and the estimated recall is $517 \times 0.99/522 \approx 98\%$.
- The estimated end-to-end precision/recall of Cymphony over all 522 titles is $\frac{417 \times 0.99 + 105}{522} \approx 99.2\%$.

4.1.2 Column Classification in a Data Catalog

The column classification task is defined as follows: given a relational table and a target semantic label L (e.g., *person_age*), the goal is to identify all columns that can be assigned label L. This task reflects a common data catalog operation in which organizations classify columns to improve data understanding, discovery, and governance processes, as illustrated in Figure 4.5.

The CS workflow follows the same multi-stage structure as the color extraction pipeline. It collects redundant crowd judgments (*Yes/No/Unsure*), applies majority agreement, estimates labeling precision via expert sampling, and escalates ambiguous cases to experts for resolution.

We now describe how the column classification workflow is implemented in Cymphony. Although the underlying workflow structure remains the same as in the color extraction experiment,

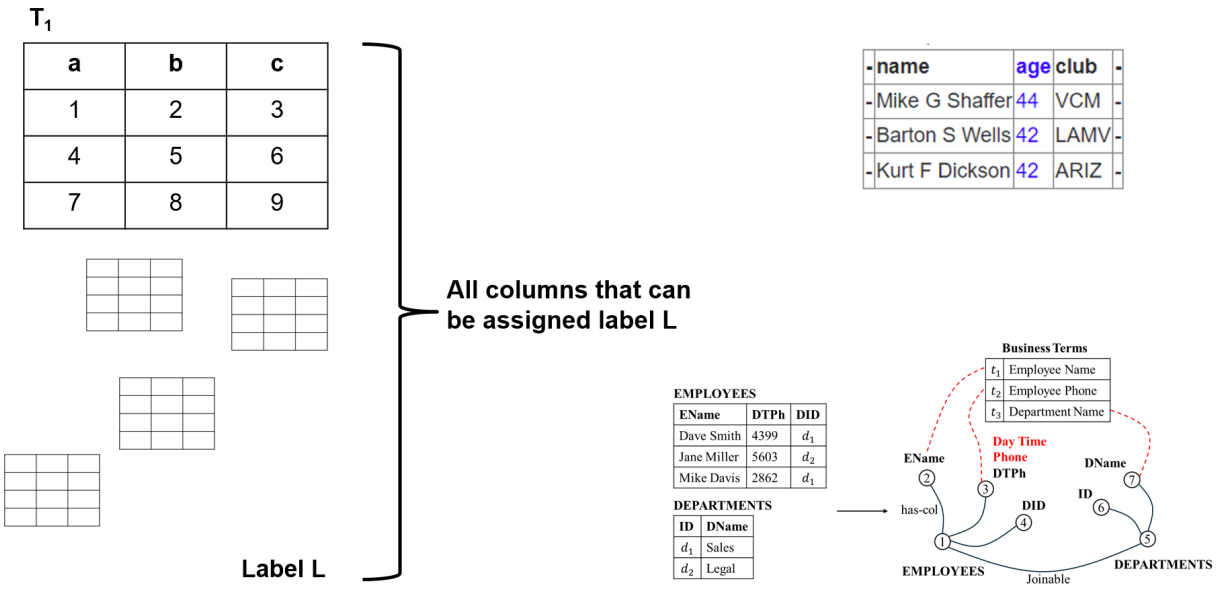

Figure 4.5: Column classification in a data catalog.

the presentation of individual tasks must be adapted to the column classification setting. Accordingly, we first explain how tasks are presented to workers, then describe the corresponding input representation used by Cymphony, and finally show how this input is integrated into our multi-stage workflow.

Figure 4.6 shows the task interface presented to workers. Each task asks a worker to determine whether a target semantic label L (e.g., *person age*) can be assigned to a highlighted target column b. To provide contextual information, surrounding columns are also displayed so that workers can reason about the broader table semantics.

To support this worker experience, the input data to Cymphony is structured as shown in Figure 4.7. Each input tuple represents a table instance similar to the one shown above. The schema encodes the target column together with its immediate left and right neighbors:

- `tc_name`, `tc_v1`, `tc_v2`: name and example values of the target column;
- `lc_name`, `lc_v1`, `lc_v2`: name and example values of the left neighboring column;
- `rc_name`, `rc_v1`, `rc_v2`: name and example values of the right neighboring column.

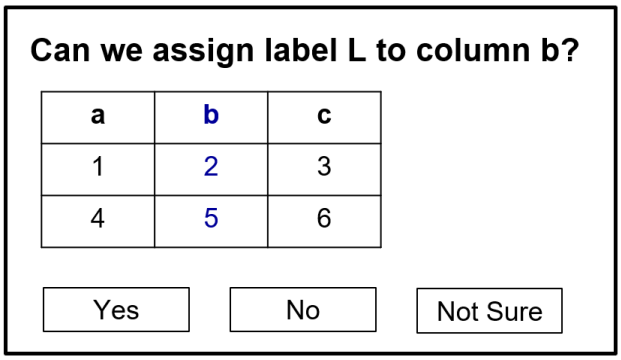

Figure 4.6: Question format presented to workers for column classification.

tc_name	tc_v1	tc_v2	lc_name	lc_v1	lc_v2	rc_name	rc_v1	rc_v2
b	2	5	a	1	4	c	3	6
…								
…								

Figure 4.7: Input data representation used by Cymphony for column classification.

Figure 4.8 shows an example of the concrete task interface presented to workers. In the actual experiment, we provide broader context by displaying up to two immediate neighboring columns on either side of the target column, whenever available.

This structured input is then processed using the same multi-stage workflow and CY program employed in the color extraction experiment (Figures 4.2, 4.3, and 4.4). Only the contents of the input table (i.e., `data.csv`) differ between the two experiments, while the workflow logic and control flow remain unchanged.

Dataset Preparation: We use a multi-column table dataset derived from Sato, available at `https://github.com/megagonlabs/sato/tree/master/table_data`. From this corpus, we select 606 tables, each containing between 2 and 5 columns. Two representative examples are shown in Tables 4.1 and 4.2. Collectively, these tables contain 1,640 columns, among which 216 correspond to our target semantic label *person_age*.

Here is how we prepared the dataset. The original Sato corpus consists of two subsets: approximately 80,000 predominantly single-column tables and 33,000 multi-column tables. We focus on

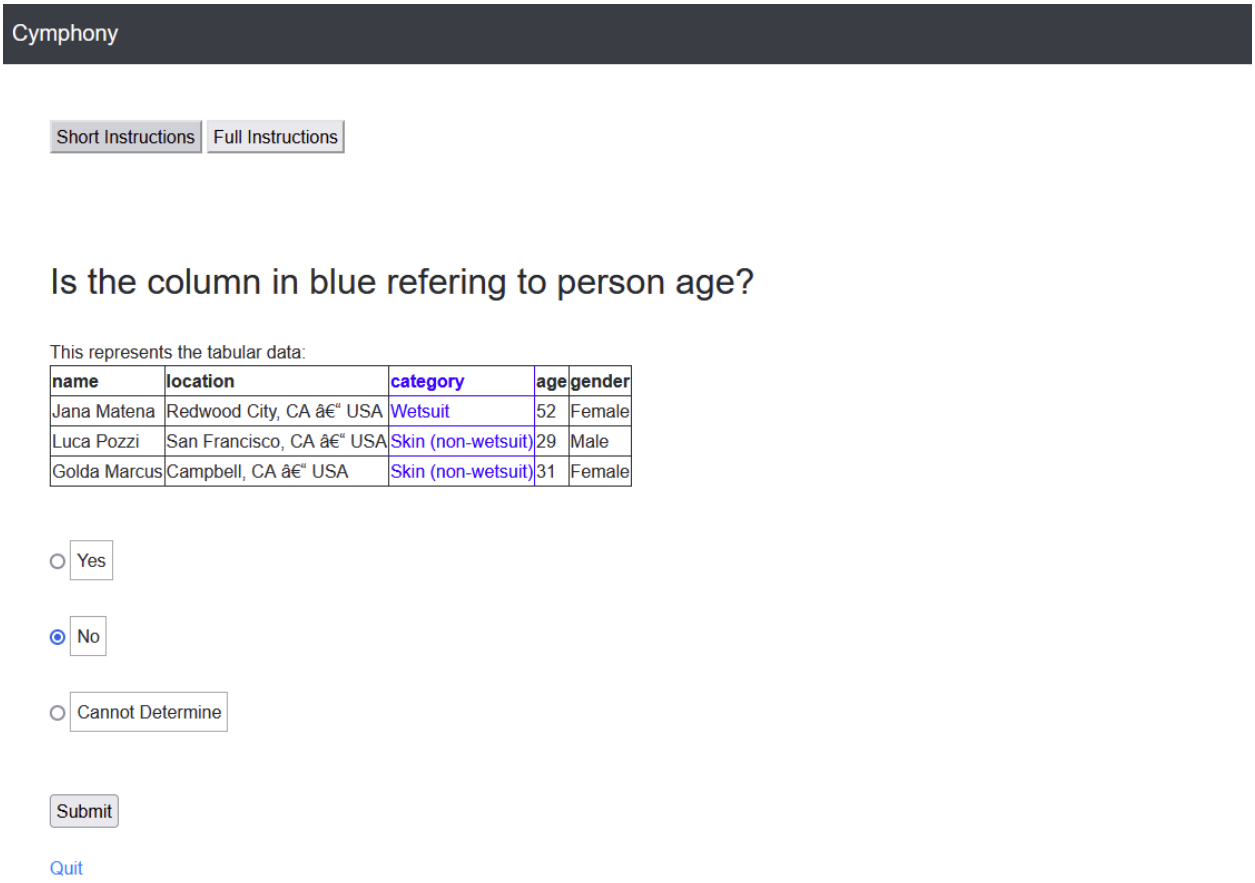

Figure 4.8: Example question presented to workers for column classification.

name	age	club
Dale Strickland	40	EMP
Sue R Herrington	42	DCM
Mary M Pohlmann	43	LINC
Donna Burkhart	42	SMMM

Table 4.1: Person table.

the multi-column subset, since these tables provide contextual information via multiple columns, named attributes, and sample values. From this subset, we randomly selected 606 tables, each containing between 2 and 5 columns, yielding a total of 1,640 columns for evaluation.

Cost: We compute AMT costs as follows. We initially collect two annotations for each of the 1,640 columns. Among these, AMT workers disagreed on 86 columns, requiring one additional

category	name	type	year
Greenhouse Gases	Methane (CH4)	Flask	Multiple
Greenhouse Gases	Carbon Monoxide (CO)	Flask	Multiple
Greenhouse Gases	Carbon Dioxide (CO2)	Flask	Multiple
Greenhouse Gases	C13/C12 in Carbon Dioxide	Flask	Multiple

Table 4.2: Greenhouse gas table.

annotation for each. In total, this yields $2 \times 1{,}640 + 86 = 3{,}366$ AMT annotations. At \$0.03 per annotation, the total AMT cost is \$100.98.

As far as in-house worker usage is concerned, we collect one annotation per column as follows. After collecting AMT labels for all 1,640 columns, 1,636 columns received a definitive label from AMT, while only four columns ($< 0.3\%$ of all columns) remained undecided due to disagreement among AMT workers; these four columns are therefore labeled by an in-house steward. In addition, we use the in-house steward to label a random sample of 100 columns for quality estimation. In total, this results in 104 in-house annotations.

Accuracy: On the 100-column validation sample, AMT annotations agreed with in-house steward labels on all 100 columns. Therefore, for the 1,636 columns that received definitive AMT labels, the estimated precision is 100% and the estimated recall is $1{,}636/1{,}640 \approx 99.75\%$. Under this estimate, the overall precision and recall of the Cymphony workflow are both $1{,}640/1{,}640 = 100\%$.

Because gold-standard labels are available for this dataset, we also compute actual accuracy. Among the 1,636 columns labeled by AMT, only one column is mislabeled, yielding an actual AMT precision of $1{,}635/1{,}636 \approx 99.93\%$ and recall of $1{,}635/1{,}640 \approx 99.69\%$. When combined with the 104 columns labeled by in-house stewards, the overall Cymphony precision and recall both become $(4 + 100 + 1535)/1640 \approx 99.93\%$.

Therefore, we conclude that Cymphony provides high accuracy at a reasonable cost, proving its utility for column classification tasks. This experiment also demonstrates that Cymphony's workflow abstraction is reusable, i.e., the same control structure (agreement, sampling, escalation) can be applied to a classification-style task with different UI layouts and worker instructions.

4.1.3 Entity Matching via Active Learning

We next evaluate Cymphony on an entity matching (EM) task driven by active learning (AL). We first introduce the entity matching via active learning (EM–AL) problem. Second, we describe the active learning workflow executed using Cymphony. Finally, we present the dataset used and analyze the experimental results.

Entity Matching: The *entity matching* problem is to find records which refer to the same real world entity. Formally, given two table A and B with the same schema, find all pairs $(a, b) \in A \times B$ that "match", where two records $a \in A, b \in B$ "match" if they refer to the same real world entity.

Example 4.1.1. *Figure 4.9 shows a tiny example of entity matching. Table A and table B contain records referring to people. Examining these records we can see that ('Dave Smith', 'Madison', 'WI') and ('Dave D. Smith', 'Madison', 'WI'), as well as ('Dan Smith', 'Middleton', 'WI') and ('Daniel W. Smith', 'Middleton', 'WI') refer to the same person. The goal of entity matching is to find all such pairs between tables A and B.*

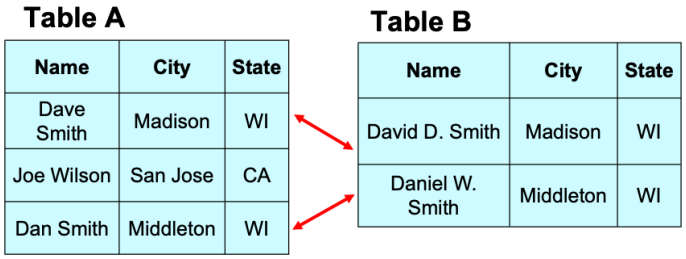

Figure 4.9: An entity matching example.

Entity matching systems are commonly structured as a two-stage pipeline. First, a *blocking* stage produces a candidate set of pairs that are likely to match. Second, a *matching* stage classifies candidate pairs as *match* or *non-match*. In this experiment, we focus on the human-in-the-loop labeling and learning aspects of the matching stage, assuming that blocking has already produced a candidate set.

Example 4.1.2. *Figure 4.10 shows a simple example of blocking and matching. In this example, the blocker outputs all pairs which have the same value in the 'State' column, creating a candidate set*

with four pairs $C = \{(a_1, b_1), (a_1, b_2), (a_3, b_1), (a_3, b_2)\}$. *The matching algorithm is then applied to the candidate set to get two predicted matches* $M = \{(a_1, b_1), (a_3, b_2)\}$.

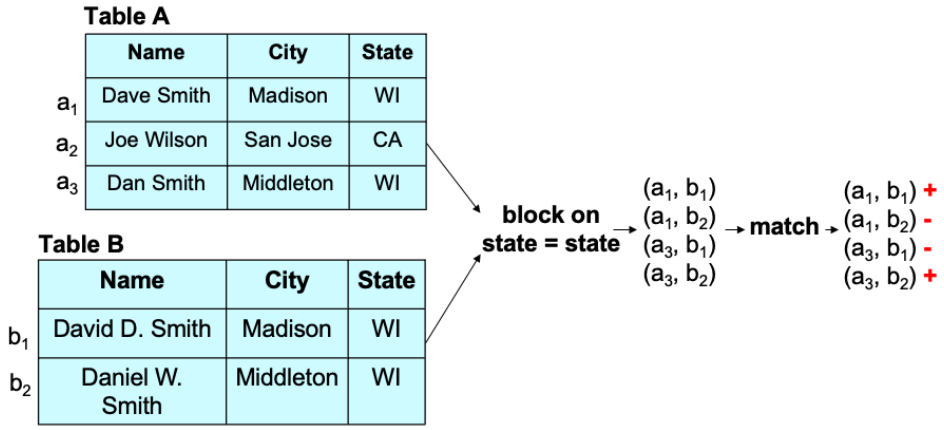

Figure 4.10: An entity matching workflow.

Active Learning: Active learning is a learning paradigm in which a model is trained iteratively by requesting labels for selected examples. Instead of labeling a large dataset uniformly at random, the learner adaptively chooses which examples to label in order to improve model quality using fewer labeled examples. This is particularly useful in settings where labels are expensive, such as human annotation tasks.

At a high level, an active learning loop repeatedly executes the following steps:

1. Train a model on the currently labeled set.
2. Score unlabeled examples and select a batch to label.
3. Acquire labels for the selected batch (e.g., via crowd workers).
4. Add the newly labeled examples to the training set and repeat.

Entity Matching via Active Learning: These two ideas are combined as follows. Rather than labeling a large number of candidate pairs up front, the system iteratively selects informative record pairs for labeling and retrains the matcher as new labels arrive. This approach is well suited to

entity matching because the candidate space can be large, and the most informative examples are typically those near the current decision boundary of the model.

In the context of Cymphony, entity matching via active learning naturally induces a workflow that alternates between (i) model-driven pair selection and (ii) redundant human labeling with aggregation. Specifically, given record pairs from two product catalogs, determine whether each pair refers to the same real-world entity. As already discussed, this task is typically addressed via machine learning with *human-in-the-loop* labeling. The evaluation question is whether Cymphony can serve as the execution backbone for an active learning loop that repeatedly requests labels, retrains a model, and selects new examples.

Using Cymphony to Solve EM–AL: To solve the above problem, we require an entity matching workflow that integrates crowdsourced labeling within an active learning loop. We implement this workflow using the Magellan platform [35] for matching and Cymphony as the human-in-the-loop labeling backend.

Given input tables A and B, Magellan first applies a blocking procedure to obtain a set of candidate record pairs C (when not already provided by the dataset). A classifier M is trained and applied to each candidate pair to predict whether it is a match. During training, the classifier is refined iteratively using active learning as follows:

1. Train an initial classifier using a small labeled seed set.
2. Apply the classifier to the remaining unlabeled candidate pairs.
3. Select a batch of $K = 30$ most controversial (i.e., uncertain) pairs.
4. Request aggregated human labels for these pairs using Cymphony.
5. Retrain the classifier using the expanded labeled set.

This procedure is repeated for 20 iterations, resulting in 600 crowd-labeled record pairs in total. Rather than directly soliciting human labels, the selected pairs are submitted to Cymphony, which coordinates redundant crowd labeling and aggregation before returning consolidated labels

to Magellan. Figure 4.11 illustrates this integrated EM–AL workflow, in which the Magellan code interacts with Cymphony via APIs. We formalize this procedure in Algorithm 4.1 and present a code snippet of the actual Python implementation in Figure 4.12.

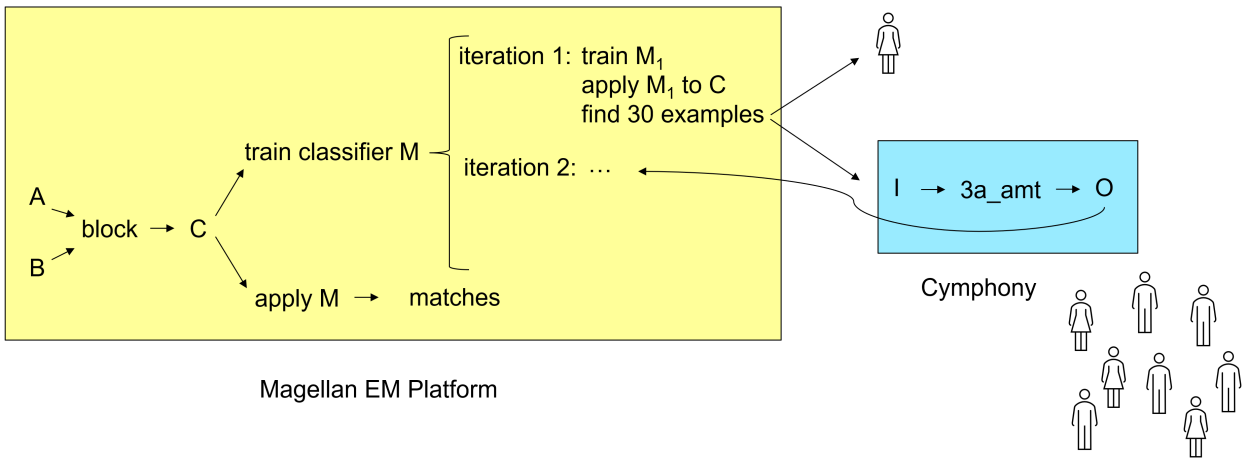

Figure 4.11: Desired EM workflow using Cymphony.

We now describe the dataset, the worker task interface, and the cost–quality performance of the EM–AL workflow executed using Cymphony.

Dataset: We use a Walmart–Amazon product matching benchmark, available at `https://github.com/anhaidgroup/deepmatcher/blob/master/Datasets.md`. This dataset consists of 2,554 Walmart tuples and 22,074 Amazon tuples, where each tuple represents a product description. After blocking, the resulting candidate set contains 10,242 tuple pairs, of which 962 are true matches.

Tasks: During the experiment, this dataset is processed by the active learning procedure described earlier. Whenever the learning algorithm requests labels, the selected record pairs are submitted to Cymphony for crowd annotation. Workers are presented with tasks of the form shown in Figure 4.13, where each task displays a tuple pair in tabular format and asks the worker to determine whether the two records refer to the same real-world entity.

Algorithm 4.1 Pseudocode for Entity Matching via Active Learning (EM–AL) using Cymphony

Input : Tables A and B, seed set T of labeled pairs, max iterations $I_{\max}$, batch size K

Output : Final predictions P over all candidate pairs C

procedure EM_AL_CYMPHONY($A, B, T, I_{\max}, K$)

Goal: Match tables A and B

$C \leftarrow$ BLOCK(A, B) ▷ C is the set of candidate pairs

$U \leftarrow C \setminus T$ ▷ unlabeled pool (exclude seeds already labeled in T)

$M \leftarrow$ TRAIN(T)

$iter \leftarrow 1$

while $iter \leq I_{\max}$ **and** $|U| > 0$ **do**

$\hat{y} \leftarrow M(U)$ ▷ Predict only on remaining unlabeled candidates

$Q \leftarrow$ SELECTCONTROVERSIAL$(U, \hat{y}, K)$ ▷ Select K controversial pairs from U

$L \leftarrow$ CYMPHONYLABEL(Q) ▷ Obtain aggregated labels for Q

$T \leftarrow T \cup L$ ▷ Expand labeled set

$U \leftarrow U \setminus Q$ ▷ Remove newly-labeled pairs from the unlabeled pool

$M \leftarrow$ TRAIN(T) ▷ Retrain on expanded labeled set

$iter \leftarrow iter + 1$

end while

$P_T \leftarrow T$ ▷ All pairs labeled during active learning (seed + queried)

$P_U \leftarrow M(U)$ ▷ Predict remaining unlabeled candidates

$P \leftarrow P_T \cup P_U$ ▷ Combine observed labels with model predictions

return P

end procedure

Note that the figure has a darker appearance because these tasks are rendered for Amazon Mechanical Turk workers. We previewed these tasks on the AMT platform but did not complete them ourselves, as task participation is restricted to registered workers.

Cost and Accuracy: We report the total crowdsourcing cost and the overall precision, recall, and F1 score achieved by the active-learning-driven matching pipeline.

AMT workers labeled a total of 904 record pairs, including 4 seed pairs and 900 pairs selected across 30 active learning iterations (30 pairs per iteration). Since each pair was annotated by three

```
def get_labels(iteration, data, s, project_id):
    # data to take labels
    # write to file
    file_name = 'data_' + str(iteration) + '.csv'
    data.to_csv(file_name, encoding='utf-8')    # for bookkeeping
    data.to_csv('data.csv', encoding='utf-8', index=False)  # for further processing

    # assuming, workflow.cy, layout.html, instructions.html are already prepared, publish data to cymphony
    create_workflow_url = target_url + '/controller/?category=workflow&action=create&pid=' + str(project_id)
    workflow_name = 'workflow_' + str(iteration) + 'for_em_al_experiment'
    create_workflow_data = {'wname': workflow_name, 'wdesc': 'experimental workflow'}
    create_workflow_response = s.post(create_workflow_url, data=create_workflow_data)
    print(create_workflow_response.content)
    workflow_id = create_workflow_response.json().get('workflow_id')
    upload_workflow_files_url = target_url + '/controller/?category=workflow&action=edit_workflow_upload_files&pid=' + str(project_id) + '&wi
    with open('./data.csv', 'rb') as f:
        upload_workflow_files_response = s.post(upload_workflow_files_url, files={'fname': f})
    with open('H:/cymphony_qual/input_material/cymphony4cs_paper_experiment_samples/em_al_experiment_full_2/instructions.html', 'rb') as f:
        upload_workflow_files_response = s.post(upload_workflow_files_url, files={'fname': f})
    with open('H:/cymphony_qual/input_material/cymphony4cs_paper_experiment_samples/em_al_experiment_full_2/layout.html', 'rb') as f:
        upload_workflow_files_response = s.post(upload_workflow_files_url, files={'fname': f})
    with open('H:/cymphony_qual/input_material/cymphony4cs_paper_experiment_samples/em_al_experiment_full_2/workflow.cy', 'rb') as f:
        upload_workflow_files_response = s.post(upload_workflow_files_url, files={'fname': f})

    create_run_url = target_url + '/controller/?category=run&action=create&pid=' + str(project_id) + '&wid=' + str(workflow_id)
    create_run_data = {'rname': 'run_1', 'rdesc': 'running the workflow'}
    create_run_response = s.post(create_run_url, data=create_run_data)
    print(create_run_response.content)
    run_id = create_run_response.json().get('run_id')
```

Figure 4.12: Python code interacting with Cymphony.

workers, this resulted in 2,712 individual labels. At a payment rate of \$0.03 per label, the total crowdsourcing cost was \$81.36.

Executing Algorithm 4.1 over our dataset yields final predictions P over all 10,242 candidate pairs. Comparing these predictions against the gold standard, we obtain an actual precision of 82.31%, recall of 62.88%, and an F1 score of 71.30. As shown in Table 4.3, these results are comparable to those reported for DeepMatcher [45], though somewhat lower than those reported by Corleone [29].

The slight performance gap relative to Corleone can be attributed to two primary factors. First, Corleone used a larger labeled training set (1,060 examples compared to the 900 used in our experiments). Second, Corleone applies specialized data preprocessing techniques that are not employed in our current pipeline.

Method	Precision	Recall	F1
Cymphony (This Work)	82.31	62.88	71.30
DeepMatcher [45]	72.30	71.50	71.90
Corleone [29]	89.70	82.80	86.00

Table 4.3: Comparison of entity matching performance.

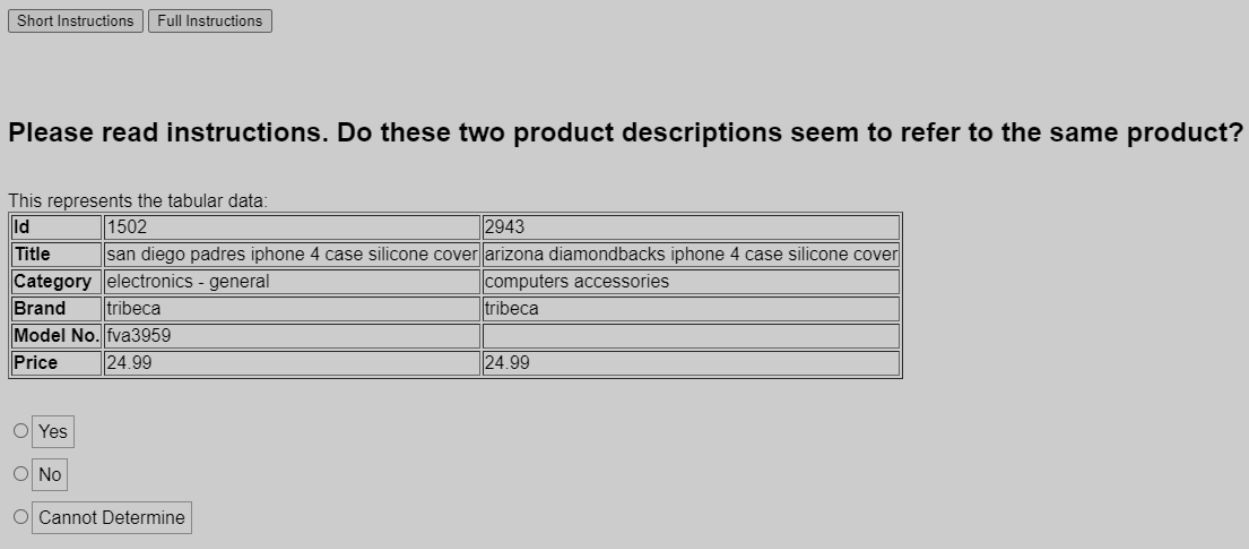

Figure 4.13: Task shown to worker.

These results demonstrate that Cymphony can solve EM–AL at a reasonable labeling cost while achieving accuracy comparable to state-of-the-art systems. More broadly, this experiment also shows that Cymphony can serve as an effective crowdsourcing execution backbone within larger data integration pipelines, enabling human-in-the-loop labeling to be seamlessly embedded as a component of complex, model-driven workflows.

4.2 Scaling Experiments

The previous experiments demonstrate that Cymphony can express and execute diverse DI workflows while achieving good accuracy at reasonable cost. However, these experiments do not fully stress the runtime, since worker arrival patterns and task completion times are not controlled.

To isolate scaling effects and systematically study runtime behavior, we conduct a set of simulated experiments using synthetic workers. We first define our target scalability requirements. Second, we briefly describe the simulator used to test whether Cymphony can scale to our target. Third, we evaluate Cymphony under three scaling dimensions: increasing dataset sizes, increasing worker concurrency, and increasing machine resources. Finally, we assess whether a distributed deployment is necessary to further reduce execution time.

4.2.1 Target Scale

For Cymphony to be useful in practical crowdsourcing deployments, we target workloads consisting of up to hundreds of thousands of tuples (e.g., 500K) and worker pools of up to several hundred concurrent workers (e.g., 500). These targets are chosen to reflect the scale of the vast majority of real-world crowdsourcing and data integration workloads.

4.2.2 The Simulator

To conduct controlled scalability experiments, we implement a **simulator** that enables the execution of synthetic workflow runs under precisely controlled conditions. The simulator allows us to regulate worker arrival processes, annotation latencies, labeling accuracy, and levels of concurrent participation.

In addition to the standard requester APIs used to create and manage workflow runs, we introduce APIs for creating simulated runs. We also expose a complementary set of worker APIs that programmatically emulate worker behavior. These APIs point to the standard worker interface and support operations such as retrieving available jobs, requesting task assignments, and submitting annotations. The simulator interacts directly with the Cymphony runtime through these interfaces, allowing synthetic workers to exercise the full execution stack in exactly the same manner as real human workers.

Example 4.2.1. *To illustrate the use of the simulator, consider the following execution flow. A user logs in and initiates a simulated run for a given workflow. Cymphony parses the workflow and, for each human operator in the DAG, requests a set of simulation parameters that govern the behavior of synthetic workers assigned to that job. After these parameters are supplied, the workflow is executed in the same manner as a normal run, except that worker arrivals and annotations are generated by the simulator according to the specified configuration.*

For each human job, the requester specifies three parameters: the number of synthetic workers that will participate in the job, the worker reliability that determines the probability with which a

synthetic worker produces a correct annotation, and the annotation latency that specifies the time spent by a worker on each task.

Synthetic workers interact with Cymphony exclusively through the standard worker APIs. They discover available jobs, repeatedly request task assignments, wait for a fixed amount of simulated annotation time, and submit annotations. This design enables controlled evaluation of runtime behavior under increasing levels of parallelism while exercising the complete worker scheduling, session management, and aggregation pipeline exactly as in real deployments.

4.2.3 Experiments Across Size, Concurrency Levels, and Machines

All simulated experiments are conducted under a common set of baseline assumptions. We focus on a simple workflow consisting of a single `3a_kn` human operator. The operator uses a 2-out-of-3 aggregation policy, meaning that a task is assigned a final label once any two of three collected annotations agree. Each synthetic worker requires four seconds to complete a task, modeling stable per-task annotation effort, and workers are restricted to annotating each task at most once.

Varying Dataset Size: We vary the number of tuples from 10K to 500K and measure end-to-end runtime under fixed worker pools of 160 and 320 workers. Workers arrive uniformly, and each synthetic worker requires four seconds per annotation.

All experiments in this section are executed on an AWS EC2 `m6a.4xlarge` instance with 16 vCPUs, 64 GB of memory, and 64 GB of EBS storage.

Table 4.4 summarizes the experimental results. The first three columns specify the simulation configuration: dataset size, number of workers, and worker reliability distribution. The remaining columns report runtime and workload statistics. Accuracy is defined as the fraction of tuples whose final aggregated label matches the gold label. Runtime measures the elapsed time between the first task assignment and the completion of the final aggregation. The *Annotations per worker* columns report the minimum, maximum, and average number of tasks completed by workers who participated in the run. The final column reports the number of workers who completed at least one task.

Example 4.2.2. *Consider experiment 1 in Table 4.4. In this run, 160 workers arrive on the human job containing 10K total tuples, with each worker's reliability drawn uniformly from* $[0.8, 1.0]$*. The resulting accuracy of 0.9713 indicates that the aggregated labels match the gold labels for 97.13% of tuples. The runtime of 00:12:04 reflects the elapsed time between the first task assignment and the final task aggregation. Among participating workers, the minimum, maximum, and average numbers of completed tasks are 98, 176, and 136.72, respectively. All 160 workers who arrived at the job, got to complete at least one task during its execution.*

Id*	Tuples	Workers	Worker reliability	Accuracy	Runtime (hh:mm:ss)	Annotations per worker			Number of workers who annotated
						min	max	avg	
1	10k	160	uniform(0.8,1)	0.9713	00:12:04	98	176	136.72	160
2	10k	320	uniform(0.8,1)	0.9710	00:07:43	30	110	69.86	312
3	20k	160	uniform(0.8,1)	0.9727	00:22:09	233	314	272.94	159
4	20k	320	uniform(0.8,1)	0.9737	00:12:53	97	178	138.22	314
5	50k	160	uniform(0.8,1)	0.9724	00:50:54	637	722	680.28	160
6	50k	320	uniform(0.8,1)	0.9756	00:28:35	303	387	344.67	315
7	100k	160	uniform(0.8,1)	0.9719	01:36:43	1323	1405	1364.84	160
8	100k	320	uniform(0.8,1)	0.9718	00:52:52	654	738	696.34	313
9	150k	160	uniform(0.8,1)	0.9691	02:27:00	2051	2130	2091.47	157
10	150k	320	uniform(0.8,1)	0.9729	01:16:40	1000	1087	1044.01	313
11	300k	160	uniform(0.8,1)	0.9727	04:42:41	4042	4123	4083.33	160
12	300k	320	uniform(0.8,1)	0.9722	02:27:01	2021	2103	2061.46	317
13	500k	160	uniform(0.8,1)	0.9704	07:53:24	6785	6865	6824.83	160
14	500k	320	uniform(0.8,1)	0.9702	04:15:25	3424	3516	3469.09	315

Table 4.4: Scaling experiments with varying dataset sizes and worker concurrency.

Figure 4.14 visualizes the results from Table 4.4. Across all experiments, runtime increases approximately linearly with dataset size for a fixed number of workers, indicating that Cymphony scales linearly in the number of tuples under constant concurrency.

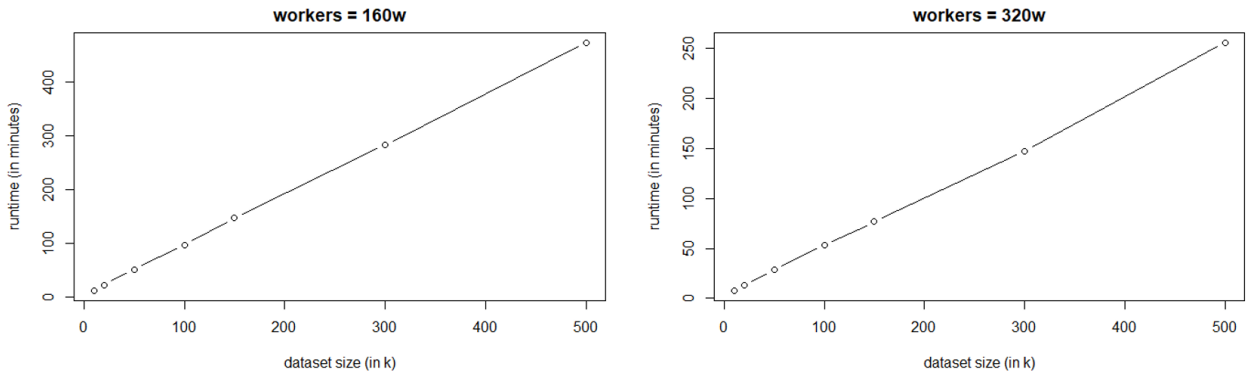

Figure 4.14: Varying Dataset Size.

Varying Number of Workers: Keeping the same deployment as before, we next study the effect of worker concurrency while holding the dataset size fixed. As a reminder, concurrency here refers to the number of workers simultaneously active on the same `3a_kn` human job.

Table 4.5 summarizes the results. The structure and interpretation of columns are identical to Table 4.4. Figure 4.15 visualizes these results.

Across all dataset sizes, increasing the number of concurrent workers yields substantial reductions in end-to-end runtime initially. However, beyond a moderate level of concurrency, additional workers provide diminishing returns as machine resources become saturated. This behavior is particularly evident for the 100K and 500K workloads, where increasing concurrency from 320 to 640 workers results in only modest runtime improvements relative to the earlier gains.

Vertical Scaling: We next evaluate how Cymphony benefits from increased compute capacity. For these experiments, we deploy Cymphony on a larger AWS EC2 instance (c6a.8xlarge, 32 vCPUs, 64 GB RAM, 64 GB EBS storage) and compare performance against the baseline deployment (m6a.4xlarge, 16 vCPUs, 64 GB RAM, 64 GB EBS storage). Table 4.6 reports the resulting runtimes under varying dataset sizes and worker concurrency levels.

At first glance, the larger machine appears to provide only modest runtime improvements. However, a closer examination reveals that the larger deployment consistently brings runtime close to the *ideal runtime* under the given workload.

Tuples	Workers	Worker reliability	Accuracy	Runtime (hh:mm:ss)	Annotations per worker			Number of workers who annotated
					min	max	avg	
10k	160	uniform(0.8, 1)	0.9713	00:12:04	98	176	136.72	160
10k	320	uniform(0.8, 1)	0.9710	00:07:43	30	110	69.86	312
20k	160	uniform(0.8, 1)	0.9727	00:22:09	233	314	272.94	159
20k	320	uniform(0.8, 1)	0.9737	00:12:53	97	178	138.22	314
50k	160	uniform(0.8, 1)	0.9724	00:50:54	637	722	680.28	160
50k	320	uniform(0.8, 1)	0.9756	00:28:35	303	387	344.67	315
100k	160	uniform(0.8, 1)	0.9719	01:36:43	1323	1405	1364.84	160
100k	320	uniform(0.8, 1)	0.9718	00:52:52	654	738	696.34	313
100k	640	uniform(0.8, 1)	0.9735	00:31:17	308	396	352.85	617
150k	160	uniform(0.8, 1)	0.9691	02:27:00	2051	2130	2091.47	157
150k	320	uniform(0.8, 1)	0.9729	01:16:40	1000	1087	1044.01	313
300k	160	uniform(0.8, 1)	0.9727	04:42:41	4042	4123	4083.33	160
300k	320	uniform(0.8, 1)	0.9722	02:27:01	2021	2103	2061.46	317
500k	160	uniform(0.8, 1)	0.9704	07:53:24	6785	6865	6824.83	160
500k	320	uniform(0.8, 1)	0.9702	04:15:25	3424	3516	3469.09	315
500k	640	uniform(0.8, 1)	0.9727	03:16:28	1675	1840	1768.22	616

Table 4.5: Scaling experiments with varying dataset sizes and worker concurrency, with red rows indicating additional high-concurrency runs.

Id	Tuples	Workers	Worker reliability	Runtime on small machine (hh:mm:ss)	Runtime on big machine (hh:mm:ss)
1	20k	160	uniform(0.8, 1)	00:22:09	00:21:03
2	50k	320	uniform(0.8, 1)	00:28:35	00:25:53
3	100k	320	uniform(0.8, 1)	00:52:52	00:49:19
4	100k	640	uniform(0.8, 1)	00:31:17	00:27:26

Table 4.6: Runtime comparison on small vs. big machine under varying dataset sizes and worker concurrency.

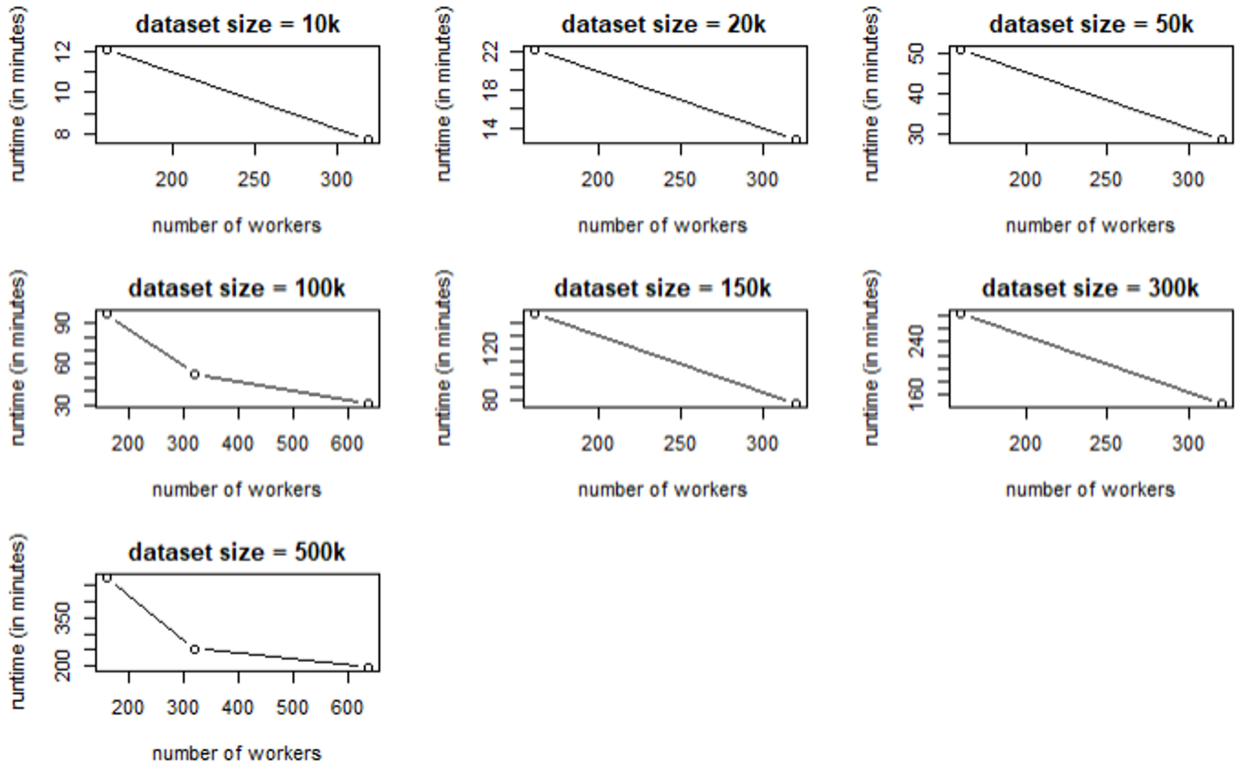

Figure 4.15: Varying the number of workers.

Under our simulation assumptions, each tuple requires on average 2.5 annotations (due to 2-out-of-3 aggregation with imperfect workers), and each worker requires four seconds per annotation. Thus, the ideal runtime for a workload with N tuples and W workers is: $T_{\text{ideal}} = \frac{N \times 2.5 \times 4}{W}$

Example 4.2.3. *Consider Simulation 1 in Table 4.6. With* $20K$ *total tuples and* 160 *workers, the ideal completion time is* $T_{ideal} = \frac{20{,}000 \times 2.5 \times 4}{160} = 1250$ *seconds* $\approx 20{:}50$. *The small machine completes this workload in 22:09, while the larger machine completes it in 21:03—within 13 seconds of the theoretical lower bound.*

Example 4.2.4. *Similarly, for Simulation 4 (100K tuples, 640 workers), the ideal completion time is approximately 26:02. The small machine requires 31:17, while the larger machine completes in 27:26—again closely approaching the ideal bound.*

These results indicate that the baseline deployment becomes compute-bound under high concurrency, whereas the larger machine shifts execution into a work-bound regime dominated by annotation throughput. Increasing machine capacity therefore allows Cymphony to approach the theoretical optimal runtime dictated by worker speed and aggregation requirements.

4.2.4 Do We Need Horizontal Scaling?

Finally, we consider whether Cymphony requires horizontal (distributed) scaling to support practical data integration workloads.

Recall that our target deployment regime consists of datasets with up to hundreds of thousands of tuples (e.g., 500K) and worker pools of up to several hundred concurrent workers (e.g., 500). Table 4.4 shows that within this regime, Cymphony already achieves highly practical end-to-end runtimes on a single moderately provisioned machine.

For example, with 640 concurrent workers and 100K tuples, Cymphony completes the workflow in approximately 30 minutes. Scaling the dataset by a factor of five to 500K tuples increases the total runtime to only about three hours. These runtimes are well within the operational expectations of real-world crowdsourcing pipelines, which are typically designed around hour-scale batch processing rather than sub-minute interactive response.

Moreover, our vertical scaling experiments demonstrate that modest vertical scaling is sufficient to push execution into a work-bound regime, where runtime is dominated by worker annotation throughput rather than system overhead.

Taken together, these results indicate that for the vast majority of realistic crowdsourcing-based data integration workloads, a single-node Cymphony deployment already provides sufficient throughput and scalability. Consequently, we conclude that a distributed version of Cymphony is not necessary to meet practical performance requirements within the targeted operating regime.

Chapter 5

Customizing Cymphony for Data Catalog Systems

In the previous chapters, we presented Cymphony as a general-purpose crowdsourcing execution engine and demonstrated its effectiveness on a range of data integration tasks. We now turn to a more systems-oriented question: how Cymphony can be customized and integrated into a real-world data catalog system that already manages large-scale data integration and curation.

In this chapter, we study this question in the context of Smartcat (SC), a data catalog platform that orchestrates automated data ingestion, enrichment, and human-in-the-loop curation. Our goal is to understand how Cymphony should be embedded within such a system, what challenges this entails, and what extensions to Cymphony are required to support diverse worker types, interaction modes, and curation workflows.

We begin by introducing Smartcat's terminology and architecture. Second, we discuss the overall solution for customizing Cymphony and integrating into Smartcat. Third, we present our concrete design and implementation. Finally, we evaluate our solution through a set of controlled experiments in varied deployment and workload settings.

5.1 The Smartcat Data Catalog System

In this section, we introduce Smartcat (SC). Smartcat represents a data catalog for an *organization*, such as a company, a scientific domain, a government agency, or another institutional entity.

SC distinguishes several human roles. **Developers** develop the SC software platform. **Admins** deploy SC for a particular organization and are responsible for installation and system maintenance. **Stewards** understand the organization's data and oversee data quality and data management within the catalog. **Users** (business users) consume the catalog to discover and understand relevant assets for their work. **Crowd workers** perform curation tasks that enrich or verify catalog content (e.g., AMT turkers).

Business users typically expect the catalog to track a set of *assets* and *relationships*. Assets include any entities judged valuable to the organization and worth tracking, such as tables, files, directories, databases, images, or even human owners of datasets.

Users may also be interested in relationships among assets, such as whether a table is unionable with another table or whether one table is derived from another.

The overall goal of SC is to build and maintain a catalog graph G that captures these assets (as nodes in G) and relationships (as edges in G) within an organization, and to make G available to business users for discovery and downstream consumption.

To do so, an admin (or, in some cases, a steward) first defines a *catalog schema* S specifying the types of assets and relationships that should be tracked. Then, SC operates in *epochs*. In each epoch, it runs *feeders*, *enrichers*, *curators*, and *consumers*:

1. **Feeders:** SC crawls organizational data sources, parses raw data to discover assets and relationships, and loads the results into graph G via one or more ETL pipelines.

2. **Enrichers:** SC executes automated enrichments that add derived information to G (e.g., value normalization, column name expansion, or model-driven annotations).

3. **Curators:** SC manages human curation by running curation workflows whose outputs are incorporated into G. A *curation* indicates that a human (e.g., a business user, steward, or crowd worker) has verified that a specific piece of catalog information is correct or incorrect.

4. **Consumers:** SC updates downstream catalog consumers, such as keyword search, browsing interfaces, and natural-language query components.

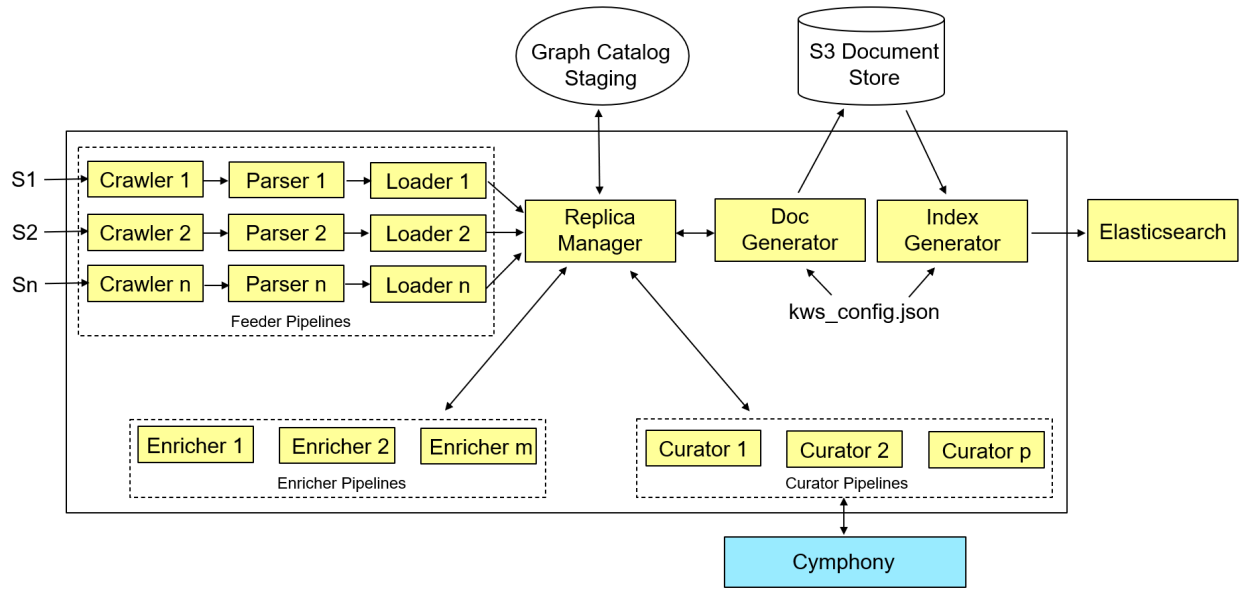

Figure 5.1: Architecture of the Smartcat (SC) backend.

Figure 5.1 illustrates the architecture of SC. Organizational data sources (S1. . . Sn) are ingested through feeder pipelines consisting of crawlers, parsers, and loaders, which persist catalog state via a Neo4j-backed Replica Manager. Subsequent enricher and curator pipelines operate over graph fragments derived from the catalog graph. Document and index generators enable search via artifacts stored in AWS S3 and indexed in Elasticsearch. Cymphony integrates with curator pipelines to provide human-in-the-loop curation, while preserving SC's core graph-based execution model.

From the perspective of **curation**, SC has the following requirements:

1. Curation should be *easy to set up and maintain*.

2. The system should *support a variety of curation workflows*.

3. The system should *support multiple worker types*, including stewards, regular organization users, and turkers.

4. The system should *support multiple interaction modes*, including drive-by curation and bulk curation. Here, *drive-by curation* denotes lightweight, inline validation actions performed

during normal catalog browsing, whereas *bulk curation* denotes structured annotation sessions in which a user processes a sequence of system-assigned tasks.

5.2 Overall Solution Idea

To fulfill SC's curation requirements, it is natural to leverage Cymphony's specialized workflow and execution capabilities that we demonstrated in the previous chapters. The key question therefore shifts from *whether* to use Cymphony to *how* to integrate it effectively. We consider two integration options:

1. **Embedding Cymphony within SC.** In this approach, Cymphony's core logic is integrated directly into the SC codebase, and SC invokes Cymphony functionality as an internal module. The advantages of this approach are potentially lower latency (due to direct in-process calls) and a unified deployment artifact. The disadvantages include increased codebase complexity, tighter interdependencies and version coupling, and potential resource contention due to shared databases and compute resources.

2. **API-driven integration.** In this approach, SC and Cymphony run as separate systems, and SC invokes Cymphony through a stable set of API endpoints. Advantages include loose coupling, independent evolution and deployment, and a clear contract boundary. Under this model, SC can use any internal technology stack as long as it conforms to the API. Disadvantages include minor network latency and serialization/deserialization overhead per API call, which we do not expect to be a bottleneck for our workloads.

Given these trade-offs, we adopt **API-driven integration** as the preferred design. This choice avoids unnecessary coupling and makes the combined system easier to deploy and maintain—which aligns directly with SC's curation goals.

Furthermore, while Cymphony already supports a broad range of workflows, SC's requirements around diverse worker types and interaction modes motivate a set of **targeted Cymphony customizations**, which we describe in the following sections.

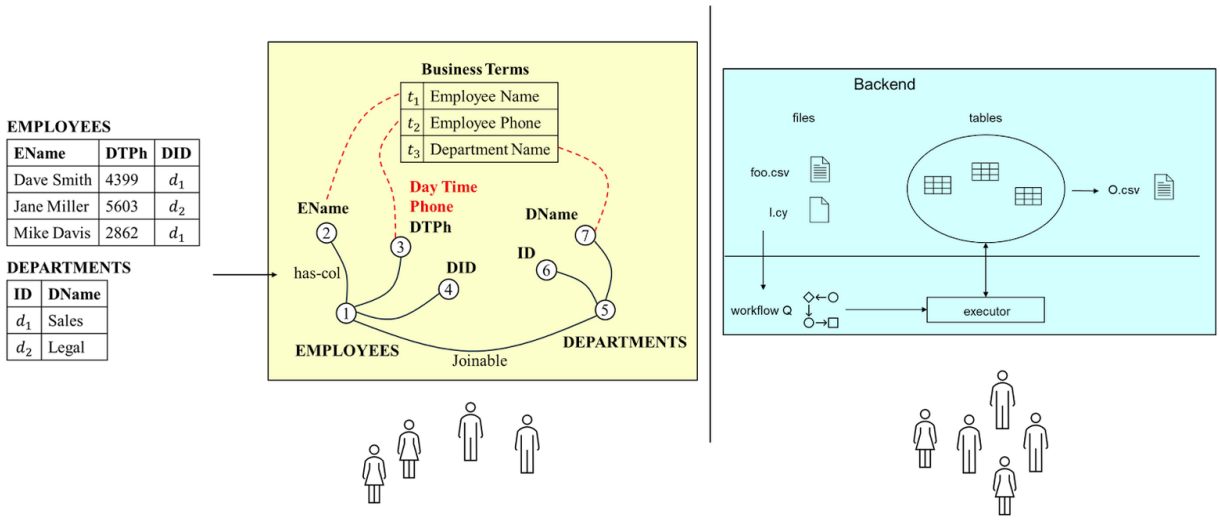

Figure 5.2: API-driven integration of Smartcat and Cymphony.

Figure 5.2 illustrates a typical collaborative scenario between SC and Cymphony under this API-driven design. A representative end-to-end flow is as follows:

1. SC runs feeders to crawl organizational data sources and populate the catalog graph.

2. SC runs enrichers to add derived or normalized information to the catalog (e.g., mapping the column name `DTPh` to "Day Time Phone" using automated algorithms or LLM-based enrichments).

3. SC consolidates these enrichments and prepares a set of candidate curation tasks (e.g., column-name expansions requiring validation). SC then submits these tasks, together with a workflow specification, to Cymphony, resulting in the creation of a Cymphony `Run`.

4. During execution of this run, organization users and/or stewards may perform bulk curation directly within Cymphony. In parallel, SC users may submit drive-by annotations through the Smartcat interface, which are incrementally forwarded to the same Cymphony `Run`.

5. Cymphony aggregates all annotations according to the configured workflow semantics and produces final labels for each task.

6. SC retrieves these results and updates the catalog graph accordingly.

We now turn to the concrete realization of this solution, describing both its design and implementation.

5.3 Realizing the Solution: Design and Implementation

In this section, we describe how we realize the customization and API-driven integration of Cymphony with Smartcat. We proceed in four steps. First, we discuss how we customize Cymphony's user registration and management model to support multiple worker roles. Second, we describe modifications to Cymphony's operators and program model to support Smartcat-style curation workflows. Third, we describe the API-driven integration of Smartcat with Cymphony while supporting the different forms of interaction required by Smartcat. Finally, we summarize our solution and reflect on the lessons learned during the process.

5.3.1 Customizing User Registration

In what follows we describe how Cymphony's user management model is customized to support a Smartcat-like curation platform.

Motivation: A central goal of Smartcat is to support flexible, high-quality human curation over a large-scale data catalog. As discussed in Section 5.1, Smartcat's curation goals include making curation easy to set up and maintain, supporting a variety of workflows, supporting different types of workers, and supporting multiple interaction modes such as drive-by and bulk curation.

The original Cymphony platform was primarily designed around a largely uniform user population: any authenticated user could act as a requester or a worker, and during curation, the system did not meaningfully distinguish between users beyond basic authentication.

This design is insufficient for Smartcat, which explicitly requires multiple human roles with different expertise and trust assumptions.

Therefore, although Cymphony's default user registration is robust, when Cymphony is customized to serve as the curation backend for Smartcat, the concept of a *user* must go beyond a generic account and explicitly incorporate distinct roles.

We identify the need for three primary user roles in Cymphony:

1. **Regular users**, who are the primary contributors to the curation process. Their main responsibility is to annotate individual data items (tasks). Their accuracy may vary, so it should be assumed that any single annotation may be noisy and should be aggregated using redundancy and subsequent majority voting. These users correspond to the business users on the SC side.

2. **Steward users**, who represent domain experts with deeper knowledge of the relevant data. Their annotations are assumed to be more accurate and trustworthy than those of regular workers. Consequently, when the Cymphony program specifies aggregation policies, steward annotations can carry higher weight based on predefined criteria, leading to higher overall data quality.

3. **Admins**, who are responsible for the foundational setup and ongoing management of Cymphony and/or Smartcat. Their duties include creating and managing user accounts, assigning roles, and overseeing system configuration and health. Admins typically interact with the system via the Django admin interface and/or programmatic tools such as Django management commands to support headless deployments. We don't expect admins to participate in the curation process in any way.

Design of User Management Customizations: The conceptual workflow for user onboarding in a Smartcat-style deployment is as follows.

- The admin first deploys Cymphony and completes initial configuration.

- The admin then accesses a special admin-only onboarding interface or a fully programmatic provisioning path to create accounts and assign roles such as `admin`, `steward`, or (regular) `user`.

- The admin then shares initial account details with newly created users, who can then update their password upon login.

Any user who registers through the standard public signup flow is automatically assigned the `user` role by default.

To support the above workflow while preserving the robustness of Django's authentication system, we make the following four design decisions:

1. **Use of built-in Django role groups.** We represent roles using Django's built-in *Groups* abstraction. Although Django groups are typically used to bundle fine-grained Object-Relational Mapper (ORM) permissions over application models, we intentionally avoid relying on ORM-level permissions for most application data in order to preserve flexibility for research and system evolution. Instead, we use groups primarily to distinguish between different types of users (`admin`, `steward`, `user`), and grant admins the privilege to create and manage these users.

2. **Two ways for admin to manage users.** We design two complementary administration paths. First, a GUI-based approach via the Django admin interface for interactive user and role management, and second, a programmatic approach via a custom Django management command for environments where GUI access is unavailable or undesirable. For example, in CS lab machines that disallow remote graphical access.

3. **Automated default role assignment.** To minimize administrative overhead in potentially large-scale deployments, we automatically assign users who sign up via the public interface to the `user` group. This ensures immediate readiness for participation in curation tasks.

4. **Role awareness in the curation pipeline.** Beyond registration, Cymphony must be able to recognize user roles when they participate in labeling tasks, so that Cymphony can differentiate between regular workers and stewards while they are actually participating on the human jobs in a run.

Implementation of User Management Customizations: To realize the design described above, we first create three Django role groups: `admin`, `steward`, and `user`. Second, we enable and configure the Django admin interface to allow interactive user and role management. Admins can create users, assign users to groups, and manage permissions. Third, to support programmatic setup, we implement a Django management command, `manage_roles_and_users`, that enables admins to create role groups, create user accounts, and assign users to groups. It also automatically configures Django's built-in flags (`is_staff` and `is_superuser`) based on group assignments. Fourth, we implement a `post_save` signal handler that automatically assigns newly registered users to the `user` group. This ensures that self-service signups are categorized without manual intervention.

Using this customized implementation, we successfully implement the workflow for user onboarding in a Smartcat-style deployment, as follows:

1. **GUI-based setup:**
   (a) Admin creates a superuser and logs into the Django admin panel.
   (b) Admin creates role groups: `admin`, `steward`, `user`.
   (c) Admin creates privileged accounts (admins and stewards) and assigns them to appropriate groups.
   (d) Public users sign up via the standard Cymphony registration flow and are automatically assigned to the `user` group.
2. **Programmatic setup:**
   (a) Admin runs `manage_roles_and_users` to create role groups.
   (b) Admin creates users and assigns them to roles via command-line flags.
   (c) The system automatically updates `is_staff` and `is_superuser` based on role.
   (d) New users signing up through the public interface are still auto-assigned to the `user` group via the signal handler.

All of the above user-management customizations are implemented entirely on the Cymphony side. Throughout this process, Cymphony remains fully integrated with Django's built-in authentication system and the `django-registration` package, preserving standard security mechanisms such as password hashing and session management.

Underlying Data Structures (Tables): User and role management rely primarily on Django's built-in authentication tables. Core user identity and credential information used for authentication is stored in `AUTH_USER(id, username, password, first_name, last_name, email)`. The set of role groups (`admin`, `steward`, `user`) are stored in `AUTH_GROUP(id, name)`. The assignment of users to groups is recorded in a join table `AUTH_USER_GROUPS(id, user_id, group_id)`. Additional tables such as `AUTH_GROUP_PERMISSIONS`, `AUTH_PERMISSIONS`, and `AUTH_USER_USER_PERMISSIONS` exist in Django's schema but are not actively used in our current implementation, since Cymphony enforces most application-level access via explicit role checks rather than fine-grained ORM permissions.

5.3.2 Adding the 3a_knlm Human Operator

We now describe how Cymphony's operator model is customized to support a Smartcat-like curation platform.

Motivation: A wide variety of data integration (DI) problems—such as extraction, cleaning, classification, and entity matching—require human input, but differ significantly in structure and/or semantics. In Section 2.1, we introduced a general *input/output template* (see Figure 5.3) that abstracts away domain-specific details while capturing the common structure of crowdsourcing (CS) problems.

To realize this template in a flexible and extensible way, Cymphony is built around the notion of an *operator*. An operator is a self-contained, modular processing unit that performs a specific function within a larger curation workflow. Workflows are defined as a sequence of interconnected operators that collectively implement a curation process.

Product Title	Price
”ASUS X205TA 11.6 Inch Laptop (Intel Atom, 2 GB, 32GB SSD, Gold) - Free Upgrade to Windows 10”	$199.00
”Lenovo G50 Entertainment Laptop - Black: DOORBUSTER - Intel Core i7-5500U (2.4GHz / 3.0 GHz Turbo), 8GB RAM, 1TB HDD, 15.6 FHD 1080P Display, DVD Burner, AC-WiFi, USB3.0, HDMI,Bluetooth, Windows 8.1”	$799.77
...	...

(a) Input

Product Title	Price	Final Label
”ASUS X205TA 11.6 Inch Laptop (Intel Atom, 2 GB, 32GB SSD, Gold) - Free Upgrade to Windows 10”	$199.00	32GB
”Lenovo G50 Entertainment Laptop - Black: DOORBUSTER - Intel Core i7-5500U (2.4GHz / 3.0 GHz Turbo), 8GB RAM, 1TB HDD, 15.6 FHD 1080P Display, DVD Burner, AC-WiFi, USB3.0, HDMI,Bluetooth, Windows 8.1”	$799.77	1TB
...	...	...

(b) Output

Figure 5.3: The input and output for extracting hard disk space from product titles.

As a reminder, one of Cymphony's core operators is the `3a_kn` operator (see Section 2.3.2), which implements a common assign–annotate–aggregate pattern. For each task t, it solicits up to n votes and aggregates as follows. If at least k of the n votes agree, the agreed label is returned as the aggregated annotation. Otherwise, the task remains undecided, and given the label *null*.

While `3a_kn` is sufficient when all workers are treated uniformly, Smartcat requires differentiation between *regular workers* and *stewards*. In many real deployments, a desirable per-tuple policy is to collect votes from multiple regular workers, *or*, accept a smaller number of votes from more trustworthy stewards.

For example, a system may require 2 out of 3 regular workers to agree on a tuple ($k = 2, n = 3$), or alternatively allow a single steward vote to finalize the task ($l = 1, m = 1$). This configuration captures the intuition that expert input may substitute for multiple regular-worker votes.

To express such mixed voting policies, we introduce a new operator: **3a_knlm**.

Design of the 3a_knlm Operator: For each task t in the input table, the `3a_knlm` operator solicits up to n regular worker votes *or* up to m steward votes. If at least k regular worker votes agree or at least l steward votes agree, then it returns that as the *aggregated annotation*. Otherwise, it returns *null (undecided)*. As already discussed, this strategy allows us to capture many common

crowdsourcing scenarios, such as getting up to 3 regular votes, or 1 steward vote then taking the majority: $k = 2, n = 3; l = 1, m = 1$.

This operator has the form $(B, C) = 3a\_knlm(I, Instr, k, n, l, m)$, where

- Input table I is a table where each tuple $x \in I$ represents a single microtask (or just task for short).
- Instruction file $Instr$ is an HTML file that defines the task completion instructions that are presented to the worker.
- Threshold k is the minimum number of regular workers who must agree on the same label for a tuple to be considered *completed*.
- Limit n is the maximum number of regular worker annotations the system will collect for a single tuple before giving up on reaching a consensus.
- Steward-Threshold l is the maximum number of stewards who must agree on the same label for a tuple to be considered *completed*.
- Steward-Limit m is the maximum number of steward annotations the system will collect for a single tuple before giving up on reaching a consensus.
- Annotation table B is an output table storing every individual raw annotation, with the schema *(tuple_id, worker_id, annotation)* where *tuple_id* represents a tuple $x \in I$.
- Aggregated table C is a table containing the final aggregated results. For each tuple $x \in I$, the system provides a final label a^* if at least k regular workers agreed or at least l stewards agreed; otherwise, the label is `NULL`.

Example 5.3.1. *To illustrate the `3a_knlm` operator, consider an entity matching task where we set $k = 2$, $n = 3$, $l = 1$, $m = 1$. When Cymphony executes this operator (called a job), let us say the regular workers $w1$, $w2$, $w3$, and steward worker $s1$ start working on the job. Then, for a tuple-pair $x \in I$:*

- *If $w1$ says Yes and $w2$ says Yes, the task finishes early with a Yes label for x (reaching k = 2).*
- *If $w1$ says Yes, $w2$ says No, and $w3$ says No, the task finishes with a No label for x (reaching k = 2, n = 3).*
- *If $w1$ says Yes, $w2$ says No, and $w3$ says Cannot Determine, the task finishes with a `NULL` for x because no label achieved k = 2 agreements within n = 3 attempts.*
- *If no regular worker has voted, and $s1$ says Yes, the task finishes with a Yes label for x (reaching l = 1, m = 1).*
- *If $w1$ says No, and $s1$ says Yes, the task finishes with a Yes label for x (reaching k = 1; l = 1, m = 1).*
- *If $w1$ says Yes, $w2$ says No, and $s1$ says No, the task finishes with a No label for x (reaching k = 1, n = 2; l = 1, m = 1).*

Implementation of the 3a_knlm Operator: Algorithm 5.1 presents the implementation of the `3a_knlm` operator, which extends the original `3a_kn` design with role-aware aggregation semantics. As in the original algorithm, execution proceeds through the same loop of assignment, annotation, and aggregation, and the stopping condition remains that every tuple in T has received a final label in C.

The overall interaction pattern among the operators is unchanged. Tuple assignment, handling of abandoned tasks, and reinsertion of tuples into T when aggregation is inconclusive follow exactly the same control flow as in Algorithm 2.1.

The extension occurs inside the aggregation step. Aggregation in the `3a_knlm` operator proceeds in a role-triggered manner. After recording a new annotation (x, w, a), the system first determines the role of the annotating worker w by querying authentication metadata. As shown in Algorithm 5.1, this is implemented as an existence check over a join between the user–group membership table and the group table.

Algorithm 5.1 3a_knlm: Role-aware extension of the 3a_kn operator

Input : Tuples table T, worker pool W

Data : Assignments table A, Annotations table B, Final labels table C

Auxiliary Data : Auth tables U_G (`AUTH_USER_GROUPS`), G (`AUTH_GROUP`)

Initialize $A, B, C \leftarrow \emptyset$
while there exists $x \in T$ such that $x \notin C$ **do**
  if a new worker $w \in W$ arrives **then**
    $x \leftarrow \text{assign}(T, w, A)$
    $T \leftarrow T \setminus \{x\}$
    $A \leftarrow A \cup \{(x, w)\}$
    $a \leftarrow \text{annotate}(x, w)$
    if a is **null then**
      $T \leftarrow T \cup \{x\}$ ▷ Reclaim tuple on abandonment
    else
      $B \leftarrow B \cup \{(x, w, a)\}$
      ▷ — Role-aware extension —
      $is\_steward \leftarrow \exists\Big(\sigma_{\text{U\_G.user\_id}=w \,\wedge\, \text{G.name}=\texttt{steward}}(U\_G \bowtie_{U\_G.group\_id=G.id} G)\Big)$
      $type \leftarrow$ STEWARD if $is\_steward$ else REGULAR
      $a^* \leftarrow \text{aggregate}(x, B, type)$
      if a^* is **not null then**
        $C \leftarrow C \cup \{(x, a^*)\}$ ▷ Add to final table
      else
        $T \leftarrow T \cup \{x\}$ ▷ Insufficient agreement
      end if
    end if
  end if
end while
return C

Once the worker role is identified, aggregation logic is executed only for the corresponding role. That is, if the annotation originates from a regular worker, the system checks only the regular-worker consensus condition (k out of n). If the annotation originates from a steward, the system

checks only the steward consensus condition (l out of m). The two checks are never evaluated simultaneously.

Formally, let B denote the annotations table (`Ui_Pj_Wk_Rm_Jn_OUTPUTS`), U_G denote `AUTH_USER_GROUPS`, and G denote `AUTH_GROUP`. For a given task x and role $r \in \{\text{REGULAR}, \text{STEWARD}\}$, the operator computes the following grouped vote counts:

$$\gamma_{\text{annotation, count}(*)\to c}\left(\sigma_{B.task\_id=x \,\wedge\, G.\text{name}=r}\left(B \bowtie_{B.\text{worker\_id}=U\_G.\text{user\_id}} U\_G \bowtie_{U\_G.\text{group\_id}=G.\text{id}} G\right)\right).$$

This expression has the following semantics. The join $B \bowtie U\_G \bowtie G$ associates each annotation with the role of the worker who produced it. The selection $\sigma_{B.task\_id=x \,\wedge\, G.\text{name}=r}$ filters annotations to those corresponding to task x and role r. The grouping operator γ then groups the remaining tuples by annotation value and computes the number of votes c for each distinct annotation. If any annotation value has $c \geq k$ (for regular workers) or $c \geq l$ (for stewards), aggregation succeeds and that value becomes the final label a^*.

Separately, the operator must determine whether the maximum number of allowed annotations for the role has been reached. This is computed using a role-filtered count:

$$\gamma_{\text{count}(*)\to t}\left(\sigma_{B.task\_id=x \,\wedge\, G.\text{name}=r}\left(B \bowtie_{B.\text{worker\_id}=U\_G.\text{user\_id}} U\_G \bowtie_{U\_G.\text{group\_id}=G.\text{id}} G\right)\right).$$

Here, the grouping operator computes the total number of annotations t submitted for task x by workers of role r. This value is compared against the corresponding upper bound (n for regular workers or m for stewards). If the maximum has been reached and no consensus has emerged, the task is marked as undecided; otherwise, it is returned to T to collect additional annotations of the same role.

Thus, the aggregation logic mirrors the original `3a_kn` semantics while refining them along the dimension of worker roles. The control flow, failure handling, and termination conditions remain unchanged, but consensus is evaluated within role-specific vote pools, enabling expert annotations to resolve tasks with fewer total votes while preserving the original execution structure.

This enables Smartcat-style curation policies in which expert annotations can accelerate decision making without introducing new execution paths, state machines, or coordination mechanisms beyond those already present in the original design.

Underlying Data Structures (Tables): The execution of the `3a_knlm` operator relies on a set of *job-scoped relational tables* that isolate per-job state and coordinate concurrent worker interactions. These tables extend the job-scoped execution model introduced in Chapter 3, and reuse the same structural abstractions with minimal modification.

1. **Tuples table** (T) `ui_pj_wk_rm_jn_tuples(task_id, ...)` stores the input tuples to be curated as part of the job. Each tuple is assigned a system-generated `task_id`, which serves as the key to identify a task throughout assignment, annotation, and aggregation. This table also provides a mapping between user-provided identifiers (if any) and Cymphony's internal task identifiers.

2. **Tasks table** `ui_pj_wk_rm_jn_tasks(task_id, total_assigned, abandoned, pending_annotations, done)` maintains task-level execution metadata required to coordinate concurrent worker arrivals. The `done` flag is used to prevent conflicting assignments by temporarily removing a task from the assignment pool once sufficient active assignments exist. This flag may be reset during aggregation if consensus is not yet reached, allowing the task to be reassigned.

3. **Assignments table** (A) `ui_pj_wk_rm_jn_assignments(task_id, worker_id, status, timeout_threshold_at, abandoned_at, completed_at)` records task assignments to individual workers and tracks their lifecycle. The `status` attribute takes values in $\{$`PENDING_ANNOTATION`, `ABANDONED`, `COMPLETED`$\}$. This table supports both voluntary abandonment (e.g., skip or quit) and involuntary abandonment triggered by a timeout daemon.

4. **Outputs table** (B) `ui_pj_wk_rm_jn_outputs(task_id, worker_id, annotation)` persists raw annotations submitted by workers. This table serves as the input relation for role-aware aggregation in the `3a_knlm` operator, and is joined with authentication metadata to filter annotations by worker role during aggregation.

5. **Final labels table** (C) `ui_pj_wk_rm_jn_final_labels(task_id, label)` stores the aggregated label produced for each task once a consensus condition is met. The presence of a

tuple (x, a^*) in this table signals task completion and contributes to the termination condition of the operator.

5.3.3 Do We Need to Customize the CY Program?

We now describe how the Smartcat-specific customizations are reflected at the level of the Cymphony *program model*. Importantly, while we introduce new functionality in the form of a role-aware operator, the overall programming abstraction in Cymphony remains unchanged.

Recall that a Cymphony workflow is specified as a directed acyclic graph (DAG), where nodes correspond to operators and edges represent the flow of data between operators.

Conceptually, each operator H is modeled as a transformation $X = H(Y)$ where Y and X denote tables, files, or atomic values.

A complete workflow is written as a Cymphony program that composes such operators into a larger execution graph. For example, the following program specifies a simple crowdsourcing workflow using the original `3a_kn` operator:

```
T1 = read_table("data1.csv");
(B1, C1) = 3a_kn(T1, "instruction1.html", k = 2, n = 3);
write_table(C1, "final_annotations1.csv");
```

This program should be read as follows:

- The input dataset `data1.csv` is ingested into an internal table `T1`.
- The human operator `3a_kn` assigns tuples from `T1` to workers for annotation, using the task completion instructions specified in `instruction1.html`, and aggregates the resulting annotations according to a k-out-of-n majority policy. The operator completes when all tuples converge to final labels.
- The resulting aggregated labels are materialized as `final_annotations1.csv`.

As discussed earlier, to support Smartcat-style curation, where both regular workers and domain stewards participate in annotation, we introduce a new human operator, `3a_knlm`.

This extension does *not* require any changes to the Cymphony programming language, execution model, or workflow semantics. From the perspective of the program author, `3a_knlm` is simply another operator that can be used in place of `3a_kn` when role-aware aggregation is desired. For example:

```
T1 = read_table("data1.csv");
(B1, C1) = 3a_knlm(T1, "instruction1.html", k = 2, n = 3, l = 1, m = 1);
write_table(C1, "final_annotations1.csv");
```

Here, `3a_knlm` allows both regular workers and stewards to annotate tasks in `T1`. A task is finalized either when k out of n regular-worker votes agree, or when l out of m steward votes agree. As with `3a_kn`, the operator completes when all tasks converge to final labels.

Discussion: The key takeaway is that Smartcat-specific behavior is introduced entirely through *operator extension*, not through changes to the program model itself.

The `3a_knlm` operator is implemented as a role-aware generalization of `3a_kn`, with a customized aggregation subroutine, but it obeys the same assign–annotate–aggregate structure and integrates seamlessly into existing workflows.

This design directly validates the extensibility goals discussed earlier in Section 2.2, particularly with respect to enabling developer customization without modifying the core system architecture. By adding a single new operator, Cymphony can be adapted to support Smartcat's heterogeneous worker populations and curation requirements, while preserving the simplicity and composability of its original programming model.

5.3.4 The API-driven Cymphony–Smartcat Integration

We now describe the API-driven integration of Cymphony with Smartcat, designed to support the diverse interaction modes required by a Smartcat-like data curation platform.

5.3.4.1 Motivation from Practical Catalog Workflows

We design the Cymphony–Smartcat (CN–SC) API-driven integration by working backwards from a realistic enterprise data catalog scenario. In practice, organizations often manage hundreds of thousands of tables distributed across many data sources. At this scale, it becomes difficult for data scientists and analysts to manually locate relevant datasets. Data catalogs such as Smartcat address this challenge by crawling organizational data assets, extracting metadata, and organizing them into a catalog graph. Nodes in this graph represent assets such as tables or columns (e.g., `Employees`, `Departments`, or `DTPh`), while edges capture semantic or lineage relationships among them.

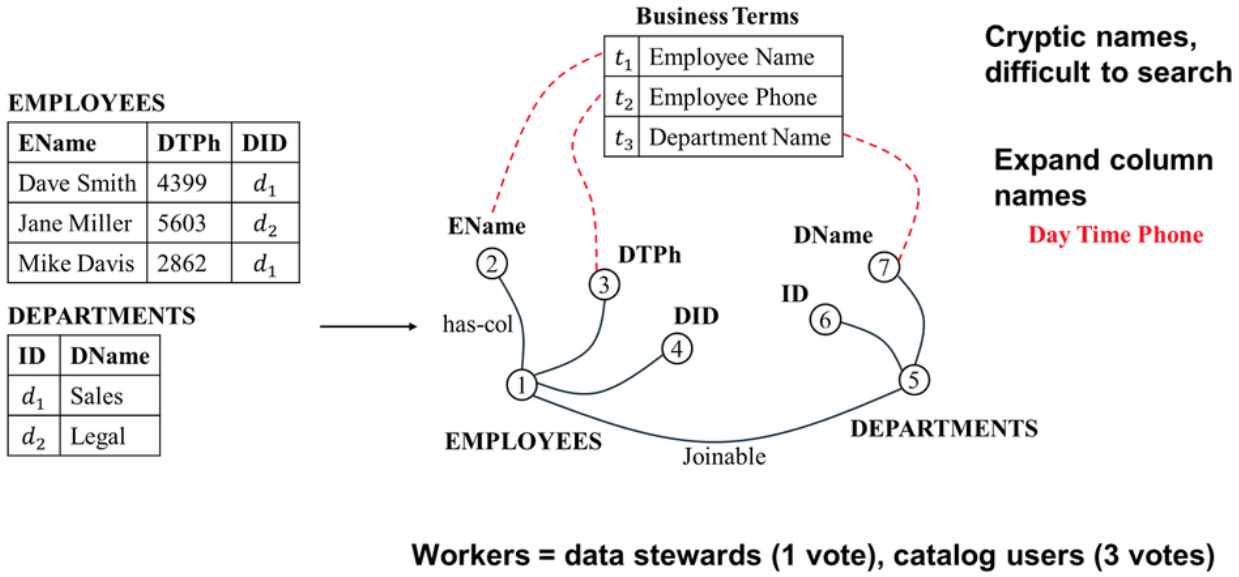

Figure 5.4: A catalog curation scenario.

Once such a catalog graph is constructed, users interact with it through search, browsing, or downstream analytics. A persistent challenge, however, is that many table and column names are cryptic or abbreviated (e.g., `DTPh`), making discovery difficult. To mitigate this, catalog systems commonly apply automated techniques—such as machine learning or large language models—to generate expanded or human-readable descriptions (e.g., mapping `DTPh` to "Day Time Phone").

These automatically generated expansions are inherently uncertain. Determining whether a proposed expansion is correct requires human validation, motivating the use of crowdsourcing and expert curation.

In Smartcat, different human roles participate in validation under different trust assumptions. A single vote from a data steward may be sufficient to validate an expansion, whereas annotations from regular catalog users may require redundancy and majority voting. As discussed earlier, Cymphony's customized role-aware operators naturally support this distinction.

Equally important, catalog users interact with the catalog in different modes depending on their availability and intent.

- In a *drive-by curation* mode, a catalog user browsing Smartcat may immediately flag or confirm a suggested expansion inline, with minimal context switching.
- In contrast, a steward—or even a catalog user—may engage in *bulk curation*, where they are explicitly assigned a sequence of validation tasks and complete them in a focused session.

Together, these interaction patterns impose concrete requirements on the CN–SC integration: (i) Smartcat must be able to submit individual curation actions to Cymphony as they occur, (ii) Cymphony must support longer-lived bulk annotation sessions, and (iii) both systems must exchange task identifiers, annotations, and final aggregated results in a structured manner.

While Cymphony's existing APIs support workflow specification and run execution, they do not directly accommodate these fine-grained, interactive curation patterns. To bridge this gap, we customize the CN–SC interface around two complementary curation modes: **bulk curation** and **drive-by curation**. Both modes operate over the same underlying Cymphony execution model, differing only in how annotations are supplied and interleaved.

Bulk curation corresponds to structured, dedicated curation sessions, typically performed by stewards or workers who explicitly engage with Cymphony:

1. Smartcat submits a table E (e.g., candidate column-name expansions) together with a workflow specification F (which crowdsources the above table) to Cymphony. Cymphony instantiates a new execution run and returns a run identifier.

2. Smartcat periodically queries Cymphony for execution status and intermediate results, eventually retrieving the final aggregated outputs.

3. To reduce latency, Cymphony additionally supports asynchronous completion notifications via a webhook mechanism. When a run reaches a terminal state, Cymphony invokes a Smartcat-provided callback URL, allowing Smartcat to retrieve results immediately without relying solely on polling.

Drive-by curation corresponds to lightweight, opportunistic annotations performed by users while browsing the Smartcat catalog:

1. As in bulk curation, Smartcat submits table E and workflow F to Cymphony, which initializes a run and returns a run identifier.

2. Smartcat continues to poll for status and partial results.

3. In parallel, if a Smartcat user X performs a drive-by curation action by selecting item Y and submitting annotation Z, then Smartcat forwards the tuple (X, Y, Z) to the corresponding Cymphony run, which records the annotation and immediately attempts aggregation for the affected task.

5.3.4.2 Integration Design

Supporting both bulk and drive-by curation within a single execution model raises several nontrivial API and semantic design challenges. Below we describe the key issues and the corresponding design decisions we adopted.

1. **Drive-by curations before a Cymphony run exists.** A potential complication arises if SC receives drive-by curations before any CN run has been created. For example, SC may accumulate expansions in a table E over time, during which users may already start curating items through the SC interface. Since this issue primarily concerns Smartcat's internal catalog management rather than Cymphony's execution model, we deliberately scope out this

complexity. We require that **SC enables drive-by curation in its GUI only after the corresponding workflow F has been published to CN**, and only for tasks that are part of the submitted table E. This restriction simplifies the integration while preserving the expressiveness needed to demonstrate Cymphony's extensibility.

2. **Excess drive-by annotations beyond CN's required vote budget.** In workflows such as `3a_kn`, CN would require at most n annotations per task. However, SC may receive more drive-by annotations than CN ultimately needs. We choose to have **SC forward all drive-by annotations to CN without pre-filtering**. CN then applies a first-come-first-served policy internally, considering only the earliest annotations up to the configured thresholds. This approach keeps the SC–CN interface simple and leaves room for future, more sophisticated pre-processing strategies on the SC side.

3. **Concurrent drive-by and bulk annotations.** Drive-by annotations from SC and bulk annotations from CN workers may arrive concurrently for the same task. This raises the question of which annotations should be considered during aggregation. We adopt a **first-come-first-served semantics** across both interaction modes. If early annotations already satisfy the aggregation criteria, later annotations are ignored. This policy is simple, consistent with the decision above, and avoids introducing complex prioritization logic across interaction modes.

4. **Identity of drive-by curation users.** This design question concerns the identity under which drive-by annotations are submitted to CN. One option is to tightly integrate user management, requiring SC and CN to share a unified authentication and identity system. Alternatively, SC could treat drive-by annotations as originating from a single logical entity. We adopt the latter approach and **treat all drive-by curations as originating from a dedicated SC-controlled account in CN**. This avoids tight coupling between SC and CN user registration systems, simplifies deployment, and aligns with our goal of demonstrating how CN can be customized and embedded without imposing global identity constraints.

5. **Number of human operators per curation run.** When SC creates a curation run, CN returns a single run identifier that is subsequently used by SC to submit drive-by annotations and retrieve results. If a run were to contain multiple human jobs (that is, human operators), it would be ambiguous from SC's perspective which job a given drive-by annotation should be routed to.

   Resolving this ambiguity would require CN to expose internal job identifiers and for SC to maintain workflow-specific mappings, thereby increasing coupling across the SC–CN service boundary. This complexity stems from API semantics rather than execution support. Job identifiers are internal execution artifacts whose assignment depends on DAG linearization and job instantiation order and does not necessarily correspond to the syntactic order of operators in the workflow specification.

   By restricting each Smartcat-issued **curation run to contain a single human operator**, all drive-by annotations and result queries remain unambiguous and can be routed solely based on the run identifier. This is not a limitation of Cymphony's execution model, which fully supports workflows with multiple human operators, but a deliberate API design choice.

We now concretize these design decisions by briefly describing the external API surface exposed by Cymphony to Smartcat. To support both bulk and drive-by curation, we define a small, focused set of APIs that form the integration boundary between SC and CN. Together, these APIs allow Smartcat to treat Cymphony as a specialized curation service while preserving loose coupling between the two systems.

Creating a Curation Run: The primary entry point from SC into CN is an API for creating a new curation run. SC invokes this API when it is ready to publish a batch of candidate curation tasks together with a workflow specification. The request includes (i) an input table T, provided as a CSV file, (ii) the name of the identifier field used to uniquely identify tuples, (iii) a Cymphony workflow specification F expressed as a `.cy` program, and (iv) auxiliary presentation artifacts such as an instruction HTML file and a layout HTML file. In addition, SC supplies a callback URL that CN can later invoke to notify SC upon run completion. If successful, CN initializes a

new execution instance, persists the input data and workflow, and returns a unique run identifier along with a status code and diagnostic message.

Submitting Drive-by Curations: To support opportunistic, inline validation from the Smartcat user interface, CN exposes a drive-by curation API that accepts incremental annotations for an existing run. Each request specifies the target run identifier together with a list of curation triples of the form $\langle x, u, v \rangle$, where x denotes the tuple identifier, u denotes the user identifier, and v denotes the annotation value. Upon receiving such a request, CN records the annotations, updates its internal execution state, and immediately attempts aggregation for the affected tasks according to the configured workflow. The API returns a status code and message indicating whether the request was successfully processed.

Monitoring Run Execution: Since curation runs may execute for extended periods of time, SC requires a lightweight mechanism to track progress. CN therefore provides a status API that allows SC to query the current execution state of a run using its run identifier. The response reports whether the run is `RUNNING`, `IDLE`, `COMPLETED`, or `ABORTED`, enabling SC to drive polling-based progress updates when desired.

Retrieving Execution Artifacts: Once a run has produced intermediate or final results, SC may retrieve these artifacts through a download API. This API allows SC to request one or more job-scoped tables associated with a run, including assignments, raw annotations, and aggregated labels. CN returns the requested tables as CSV files bundled into a single ZIP archive, allowing SC to ingest results efficiently.

Run Completion Notification: To avoid unnecessary polling delays, CN also supports asynchronous completion notification. When SC creates a run, it supplies a webhook endpoint that CN can invoke once the run reaches a terminal state. This callback includes the run identifier, the final run status (e.g., `COMPLETED` or `ABORTED`), and a completion timestamp in ISO format. Upon receiving this notification, SC can immediately fetch final results and update its catalog state without waiting for the next polling interval.

Together, these APIs provide a clean and expressive contract for integrating Smartcat with Cymphony. They support both structured bulk curation and low-latency drive-by curation while preserving CN's internal execution semantics and SC's control over user interaction and catalog state.

5.3.4.3 Integration Implementation

While Cymphony's core relational schema is designed to be generic and workflow-agnostic, integrating Cymphony as a curation backend for Smartcat requires a small number of targeted schema extensions. These extensions are intentionally minimal and preserve the overall structure and execution semantics of the original system.

First, to support Smartcat's catalog-centric workflows, we extend the `workflow_input_files` table with an additional `id_field_name` attribute. This field records the logical identifier used by Smartcat to refer to catalog entities (e.g., column or table identifiers), allowing Cymphony to maintain a stable mapping between user-facing catalog items and internally generated task identifiers throughout the execution of a curation run.

Second, to support asynchronous completion notifications, we extend the `all_runs` table with a `notification_url` attribute. This attribute stores a Smartcat-provided webhook endpoint that Cymphony invokes when a run transitions to a terminal state. By persisting the callback URL at the run level, Cymphony can issue completion notifications without introducing external state or tight coupling between the two systems.

These additions can be viewed as a form of controlled schema evolution that customizes Cymphony for Smartcat-style deployments. Importantly, the extensions do not alter the semantics of projects, workflows, runs, or jobs, nor do they affect existing operator implementations. All core coordination logic remains unchanged, ensuring that the system continues to support reproducible, concurrent, and long-running human-in-the-loop workflows while accommodating Smartcat's integration requirements.

The API implementations described below rely on these schema extensions, but otherwise reuse Cymphony's existing execution and coordination infrastructure unchanged.

Create Curation Run API: We now describe the implementation of the *Create Curation Run* API, which initializes a new CN execution instance corresponding to a Smartcat curation request. This API is responsible for persisting input artifacts, instantiating execution metadata, and preparing the runtime state needed to execute the submitted workflow.

The request includes (i) an input table T provided as a CSV file, (ii) the name of the identifier field (e.g., `tid`) used to uniquely identify tuples, (iii) a workflow specification F expressed as a `.cy` program, (iv) an instruction HTML file, and optionally (v) a layout HTML file. In addition, SC may provide an optional callback URL that CN will invoke upon run completion.

Upon receiving the request, CN performs the following steps. First, CN creates a new project entry and persists it in the `all_projects(p_id, u_id, p_name, p_desc)` table. Next, it creates a workflow under this project and inserts a corresponding record into `all_workflows(w_id, p_id, u_id, w_name, w_desc)`.

All input artifacts associated with the workflow are then registered in their respective ledger tables. Specifically, the input CSV file is recorded in `workflow_input_files(input_file_path, f_id, w_id, p_id, u_id, id_field_name)`, while the instruction file, optional layout file, and workflow program are recorded in `workflow_inst_files`, `workflow_layout_files`, and `workflow_cy_files`, respectively. The actual files are stored on disk under a hierarchical directory structure of the form `$HOME/u<u_id>/p<p_id>/w<w_id>/`. Storing the identifier field name alongside the input table ensures that tuple identities can be consistently recovered during execution.

CN then creates a new run instance and inserts a record into `all_runs(r_id, w_id, p_id, u_id, r_name, r_desc, r_status, notification_url)`. A run-specific directory `r<r_id>` is created under the workflow directory, and all workflow artifacts are copied into this location to isolate execution state.

Next, CN parses the `.cy` program to construct an in-memory directed acyclic graph (DAG) representing the workflow. This DAG is persisted using the tables `ui_pj_wk_rm_nodes(n_id, name, type)` and `ui_pj_wk_rm_edges(src_id, dst_id)`, and a topological execution order is computed and stored in `ui_pj_wk_rm_nodes_execution_order(n_id, position)`.

CN then performs a single pass over the DAG to initialize job-level execution state. For each operator node, a corresponding job entry is created in `all_jobs(j_id, r_id, w_id, p_id, u_id, j_name, j_type, j_status)`. Automatic operators are initialized with job type `automatic`, while human operators are initialized with job type `human`; all jobs start in the `IDLE` state. A one-to-one mapping between DAG nodes and jobs is recorded in `ui_pj_wk_rm_mapping_operator_node_vs_job`.

Finally, CN begins executing the workflow by advancing through the DAG in topological order. Execution proceeds until either the run completes or a human operator is encountered, at which point execution pauses and the corresponding job is transitioned to the `RUNNING` state, awaiting worker interaction.

If initialization succeeds, the API returns a success status, a diagnostic message, and a globally unique run identifier of the form `user_id.project_id.workflow_id.run_id`.

Drive-by Curation API: The drive-by curation endpoint enables Smartcat to submit opportunistic, inline annotations to an already-running Cymphony run. Conceptually, the endpoint takes a run identifier and a batch of curations of the form $\langle x, u, v \rangle$, where x is the Smartcat-side tuple identifier (e.g., an expansion ID), u is the Smartcat user identifier, and v is the annotation value. The endpoint then (i) maps Smartcat tuple identifiers to Cymphony task identifiers, (ii) materializes these curations into the job-scoped output relation, and (iii) immediately attempts aggregation for the affected tasks.

The request includes as input: (i) a composite run identifier (u_id, p_id, w_id, r_id) and (ii) a list of curations $\{\langle x, u, v \rangle\}$.

Upon receiving the request, CN first parses the composite run identifier to recover (u_id, p_id, w_id, r_id) and parses the JSON payload into an in-memory list of curation triples. It then identifies the relevant human job for this run by querying the global job catalog `all_jobs(j_id, r_id, w_id, p_id, u_id, j_name, j_type, j_status, date_creation)` and selecting the unique human job corresponding to the `3a_kn` or `3a_knlm` operator for this run.

Next, CN retrieves the Smartcat tuple identifier field name used in the input table (e.g., `tid`) from the workflow ledger entry `workflow_input_files(input_file_path, f_id, w_id, p_id, u_id, id_field_name)`. This identifier is required to map Smartcat's tuple-level curations to Cymphony's internal task identifiers.

For auditability and debugging, CN first appends the raw drive-by curations to a dedicated job-scoped table `ui_pj_wk_rm_jn_drive_by_curation_votes(`*id_field_name*`, worker_id, annotation)`. Here, the received x values populate the column named by *id_field_name*, while u and v populate `worker_id` and `annotation`, respectively.

Then, CN rewrites the incoming Smartcat user identifier u to a single CN-side "Smartcat drive-by" account identifier (as described in our design decisions), ensuring that SC and CN user management remain loosely coupled. This rewrite also prevents identifier collisions between Smartcat and Cymphony user namespaces, which are managed independently and may otherwise overlap.

Since aggregation operates over internal task identifiers, CN next maps each received tuple identifier x to its corresponding internal task identifier `task_id` by joining against the job-scoped tuples table `ui_pj_wk_rm_jn_tuples(task_id, ..., `*id_field_name*`, ...)`, producing triples of the form $\langle$`task_id`$, u, v\rangle$.

The mapped annotations are then appended to the standard job-scoped outputs table `ui_pj_wk_rm_jn_outputs(task_id, worker_id, annotation)`, making drive-by votes indistinguishable from other annotations at the aggregation layer.

After inserting drive-by annotations, CN attempts aggregation for each affected `task_id`. It first loads operator parameters (e.g., k and n) from `ui_pj_wk_rm_jn_config_parameters(key, value, value_data_type)`. For each affected task, CN: (i) checks whether a final label already exists in `ui_pj_wk_rm_jn_final_labels(task_id, label)` and skips aggregation if so; (ii) otherwise acquires a row-level lock on the corresponding row in `ui_pj_wk_rm_jn_tasks(task_id, total_assigned, abandoned, pending_annotations, done)` to prevent race conditions with concurrent bulk annotations; (iii) evaluates whether the current set of votes in `ui_pj_wk_rm_jn_outputs(task_id, worker_id, annotation)` satisfies the configured convergence criterion; and (iv) if convergence is achieved, inserts the aggregated label into

ui_pj_wk_rm_jn_final_labels and marks the task as done in the tasks table. Finally, CN updates any modified task metadata and releases the lock.

The endpoint returns a status code (SUCCESS, FAILURE, or an error type) and a diagnostic message describing request handling outcomes.

Run Status API: The run status API provides Smartcat with a lightweight mechanism for monitoring the execution state of a Cymphony run. This endpoint is used both for periodic polling and for sanity checks when reconciling asynchronous notifications.

The API takes as input a composite run identifier of the form user_id.project_id.workflow_id.run_id. Upon receiving a request, CN parses the identifier to recover the corresponding user, project, workflow, and run identifiers. It then retrieves the run metadata from the global catalog table all_runs(r_id, w_id, p_id, u_id, r_name, r_desc, r_type, r_status, notification_url).

The API returns the current execution status of the run, which is one of IDLE, RUNNING, COMPLETED, or ABORTED. This information allows Smartcat to track progress, detect failures, and coordinate result retrieval without exposing any internal execution state beyond the run lifecycle.

Download Tables API: The download tables API enables Smartcat to retrieve intermediate or final artifacts produced by a Cymphony run. This endpoint is typically invoked after run completion, but it may also be used to inspect partial results during long-running executions.

The API accepts as input a composite run identifier user_id.project_id.workflow_id.run_id together with a list of logical table names requested by Smartcat, such as Assignments, Annotations, or Aggregations. Upon receiving the request, CN first parses the run identifier and retrieves the corresponding run metadata from all_runs(r_id, w_id, p_id, u_id, r_name, r_desc, r_type, r_status, notification_url). From this metadata, CN determines the filesystem path of the run-scoped directory.

Next, CN identifies the unique human job associated with the run by querying the global job catalog all_jobs(j_id, r_id, w_id, p_id, u_id, j_name, j_type, j_status)

and selecting the job corresponding to the 3a_kn or 3a_knlm operator. Using this job identifier, CN resolves each requested logical table name to its internal job-scoped relation by prefixing it with the appropriate run and job namespace.

For each resolved table, CN joins the job-scoped relation with the corresponding tuples table ui_pj_wk_rm_jn_tuples(task_id, ...) on the internal task identifier task_id. This join restores the original input attributes alongside assignment, annotation, or aggregation metadata, producing a user-consumable view of the results.

The resulting tables are exported as CSV files into the run directory $HOME/u<u_id>/p<p_id>/w<w_id>/r<r_id>/. All requested CSV files are then bundled into a single ZIP archive, which is returned as the API response. In the event of an error, the API returns an appropriate failure status together with a diagnostic message describing the cause of the failure.

Run Completion Webhook: To support low-latency notification of run completion, Cymphony provides an asynchronous webhook mechanism that allows Smartcat to be notified immediately when a run reaches a terminal state. This mechanism complements the polling-based status API and avoids unnecessary delays in result ingestion.

On the Cymphony side, when all operator jobs in a run complete successfully, the run is transitioned to the COMPLETED state by updating the corresponding entry in the global runs table all_runs(r_id, w_id, p_id, u_id, r_name, r_desc, r_type, r_status, notification_url). At this point, Cymphony retrieves the notification_url associated with the run from the same table. This URL was provided by Smartcat at run creation time and uniquely identifies the Smartcat endpoint that should receive completion notifications. Cymphony then constructs a notification payload containing: (i) the composite run identifier user_id.project_id.workflow_id.run_id, (ii) the final run status (e.g., COMPLETED or ABORTED), and (iii) a completion timestamp encoded in ISO format. This payload is transmitted to the notification URL via an HTTP request. Failures to deliver the notification are logged but do not affect the final state of the run.

On the Smartcat side, the webhook endpoint accepts the notification payload and records the run's terminal status. The input to the Smartcat webhook API consists of the composite run identifier, the run status, and the completion timestamp. The endpoint returns a status code indicating whether the notification was successfully processed, along with a diagnostic message if necessary.

By externalizing completion notification through a webhook, the CN–SC integration avoids tight synchronization between the two systems while ensuring that Smartcat can react promptly to completed curation runs. This design preserves loose coupling, supports asynchronous execution, and integrates naturally with Smartcat's catalog update pipeline.

5.3.4.4 Underlying the Data Structures of Cymphony

The CN–SC integration relies on Cymphony's existing relational catalog, augmented with a small number of Smartcat-specific extensions. For clarity, we organize the tables below according to their scope: global catalog tables, workflow artifact tables, run-scoped execution tables, and job-scoped coordination tables.

Global Catalog Tables: These tables capture the core logical entities of projects, workflows, runs, and jobs, and store their persistent metadata. The optional `notification_url` attribute is a Smartcat-specific extension that enables asynchronous run-completion callbacks.

- `all_projects(p_id, u_id, p_name, p_desc)`
- `all_workflows(w_id, p_id, u_id, w_name, w_desc)`
- `all_runs(r_id, w_id, p_id, u_id, r_name, r_desc, r_status, r_type, notification_url)`
- `all_jobs(j_id, r_id, w_id, p_id, u_id, j_name, j_type, j_status)`

Workflow Artifact Tables: These tables record workflow-level input artifacts and presentation templates uploaded by the requester. The `id_field_name` attribute is a Smartcat-specific extension that preserves the mapping between catalog identifiers and internal task identifiers.

- workflow_input_files(input_file_path, f_id, w_id, p_id, u_id, id_field_name)
- workflow_inst_files(inst_file_path, f_id, w_id, p_id, u_id)
- workflow_layout_files(layout_file_path, f_id, w_id, p_id, u_id)
- workflow_cy_files(cy_file_path, f_id, w_id, p_id, u_id)

Run-Scoped Execution Tables: For each workflow execution, Cymphony creates a dedicated run-scoped namespace that stores the compiled workflow DAG and execution metadata.

- ui_pj_wk_rm_nodes(n_id, name, type)
- ui_pj_wk_rm_edges(src_id, dst_id)
- ui_pj_wk_rm_nodes_execution_order(n_id, position)
- ui_pj_wk_rm_mapping_operator_node_vs_job(n_id, j_id)

Job-Scoped Coordination Tables: Human operators require finer-grained coordination to manage concurrent worker assignments, annotations, and aggregation. Cymphony therefore isolates each human job using job-scoped tables.

- ui_pj_wk_rm_jn_config_parameters(key, value, value_data_type)
- ui_pj_wk_rm_jn_tuples(task_id, ⟨ fields_in_input_file ⟩)
- ui_pj_wk_rm_jn_tasks(task_id, total_assigned, abandoned, pending_annotations, done)
- ui_pj_wk_rm_jn_assignments(task_id, worker_id, status, timeout_threshold_at, abandoned_at, completed_at)
- ui_pj_wk_rm_jn_outputs(task_id, worker_id, annotation)

- `ui_pj_wk_rm_jn_final_labels(task_id, label)`

Smartcat-Specific Job Extensions: To support drive-by curation originating from the Smartcat interface, we introduce an additional job-scoped table that records raw drive-by annotations prior to task-ID resolution and aggregation.

- `ui_pj_wk_rm_jn_drive_by_curation_votes(⟨ id_field_name ⟩, worker_id, annotation)`

Together, these tables provide a complete and consistent relational substrate for CN–SC integration, supporting workflow instantiation, long-running execution, concurrent human interaction, role-aware aggregation, and asynchronous result delivery via run-scoped callbacks.

5.3.5 Solution Summary and Lessons Learned

The Smartcat integration provides a concrete, end-to-end case study of how Cymphony can be customized and embedded into a real-world data management system without compromising its core execution model. Viewed holistically, the integration exercise reinforces several key lessons about operator design, workflow modeling, and system extensibility.

Operator Extensibility Without Workflow Disruption: A central theme throughout this chapter is that defining the right abstraction boundary between operators and workflows is non-trivial. As discussed earlier in Section 2.2, crowdsourcing workflows often involve complex interactions among task assignment, annotation, and aggregation, which do not naturally fit into a simple DAG abstraction.

The introduction of the `3a_knlm` operator illustrates how Cymphony's operator model supports meaningful extensions without requiring changes to the surrounding workflow semantics. Rather than introducing new workflow constructs or execution paths, we extend an existing operator by parameterizing its aggregation logic with worker roles and thresholds. As a result, the Cymphony program remains unchanged except for a single operator substitution. This demonstrates that carefully chosen operator abstractions can absorb substantial complexity internally while preserving a stable and composable workflow interface.

Separation of Execution Semantics and Interaction Modes: Another key insight from the integration is the clear separation between Cymphony's execution semantics and Smartcat's interaction patterns. Bulk curation and drive-by curation differ substantially from a user experience perspective, yet both are mapped onto the same underlying execution model in Cymphony. By treating drive-by annotations as incremental inputs to an ongoing run—rather than as a separate execution mode—we avoid duplicating logic or introducing special cases into the runtime.

This design reinforces the idea that interaction modes belong primarily at the API boundary, while the execution engine should remain agnostic to how annotations arrive. As long as annotations are recorded in the appropriate job-scoped relations, the existing coordination and aggregation mechanisms apply unchanged.

Schema Evolution as a Controlled Customization Mechanism: Supporting Smartcat required only minimal schema extensions: recording a catalog-facing identifier field and persisting a run-level notification URL. These changes illustrate a pragmatic approach to schema evolution in research systems. Rather than over-generalizing the schema upfront, we introduce targeted extensions that are localized, backward-compatible, and easy to reason about.

Crucially, these extensions do not alter the semantics of projects, workflows, runs, or jobs. They demonstrate that Cymphony's relational catalog can evolve to support new integration scenarios while preserving isolation and reproducibility.

API Boundaries Enable Loose Coupling: The CN–SC APIs serve as a clean contract between a data catalog system and a crowdsourcing execution engine. By exposing run creation, incremental annotation submission, status monitoring, and artifact retrieval as explicit APIs, Smartcat can treat Cymphony as a specialized service rather than an embedded component. This loose coupling simplifies deployment, avoids shared assumptions about identity management, and enables each system to evolve independently.

From a systems perspective, this reinforces the value of designing human-in-the-loop platforms around explicit service boundaries, even when tight integration might appear attractive initially.

Implications for Research and Practice: Taken together, these lessons suggest that Cymphony's design strikes a useful balance between abstraction and flexibility. Operators encapsulate complex coordination logic, workflows remain simple and compositional, and targeted customization is achieved through operator extensions, schema evolution, and APIs rather than invasive architectural changes.

The Smartcat integration thus demonstrates that Cymphony is not only a research prototype for isolated crowdsourcing tasks, but a practical execution substrate that can be adapted to support large-scale, heterogeneous, and interactive data management systems.

5.4 Evaluating the Solution

This section evaluates our CN–SC customization and integration. We begin by describing the experimental setup and evaluation methodology. We then introduce the dataset used in our study. Next, we present the experimental results across a range of bulk, drive-by, and mixed curation scenarios. Finally, we synthesize findings across all experiments and summarize the overall implications for the CN–SC integration.

5.4.1 API-Driven Smartcat Client for Evaluation

To evaluate our solution in isolation, we implement an API-driven Smartcat client that emulates the behavior of a catalog frontend interacting with Cymphony. Rather than modifying Cymphony's internals or introducing integration-specific shortcuts, this client interacts exclusively through the Cymphony APIs. As a result, all integration logic exercised in these experiments is identical to what would be used by a production Smartcat deployment.

This client serves as an experimental harness that allows us to systematically explore bulk curation, drive-by curation, and mixed interaction scenarios, while maintaining full control over worker behavior and timing. Conceptually, it plays the role of Smartcat in the evaluation, while Cymphony remains an unmodified backend service.

At a high level, once an administrator has configured Cymphony and created a dedicated Smartcat account, the evaluation client executes the following workflow:

1. Exposes an HTTP endpoint to receive asynchronous run-completion notifications from Cymphony.
2. Prepares and pre-processes the experimental dataset.
3. Authenticates with Cymphony using its public login API.
4. Creates a curation run via the *Create Curation Run* API, uploading the input dataset, instruction template, and workflow specification.
5. Monitors run progress by combining periodic status polling with webhook-based completion notifications, prioritizing callbacks when available.
6. Downloads intermediate or final results via the results retrieval API, depending on the run state.
7. Computes evaluation statistics over the returned outputs.

On top of the core interaction loop, the evaluation client supports three complementary simulation modes, which can be enabled in isolation or combined within a single run depending on the experiment:

1. **Bulk curation by synthetic regular workers.**
2. **Bulk curation by synthetic regular workers and synthetic stewards.**
3. **Drive-by curation originating from the Smartcat interface.**

These three modes are chosen to isolate the effects of (i) redundancy among homogeneous workers, (ii) role-aware trust differentiation, and (iii) interaction modality (bulk versus drive-by). Together, they allow us to independently evaluate aggregation robustness, role-aware customization, and API-level interleaving of annotation streams.

Bulk curation by synthetic stewards and regular workers follows a common template. For a given target run and set of simulation parameters, synthetic workers repeatedly: (i) authenticate

with Cymphony, (ii) discover available jobs, (iii) select the job corresponding to the target run, (iv) perform annotations according to a configured accuracy model, (v) remain active for a configurable duration, and (vi) record per-worker statistics such as completed tasks and annotation outcomes. This setup allows us to control worker arrival rates, annotation accuracy, and participation duration in a reproducible manner.

Drive-by curation is simulated from the Smartcat side of the interaction. After creating a curation run, the client: (i) samples tuples from the same input table supplied to Cymphony, (ii) generates synthetic drive-by annotations, (iii) batches these annotations, and (iv) submits them to Cymphony via the *Drive-by Curation* API. These annotations are interleaved with ongoing bulk curation, exercising Cymphony's ability to aggregate votes arriving through both interaction modes.

Throughout all experiments, the client logs detailed traces of API invocations, worker actions, timing events, and returned results. These logs are later used to compute evaluation metrics and to verify correctness and consistency across runs.

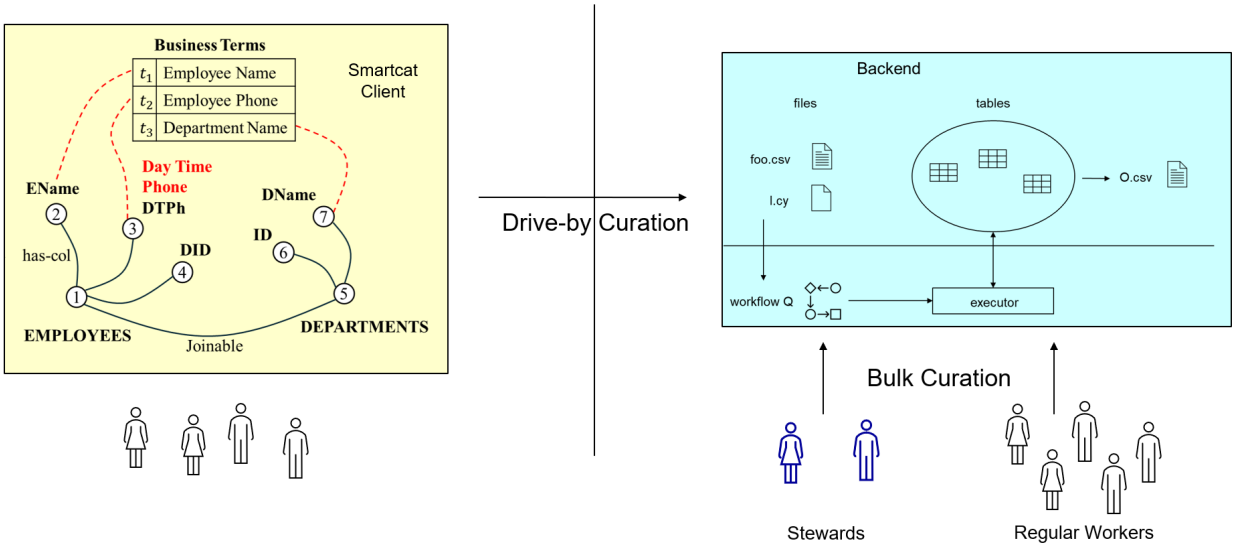

Figure 5.5: API-driven Smartcat client interacting with Cymphony through bulk and drive-by curation APIs.

5.4.2 Dataset

For all our subsequent experiments, we use a real-world dataset derived from the Environmental Data Initiative (EDI) corpus [25], enriched through Smartcat's enricher pipeline. Specifically, Smartcat applies its Column Name Expansion (CNE) enricher [5] to the EDI dataset, leveraging the *Maverick* large language model [42] to generate human-readable expansions for cryptic column names. The resulting dataset contains **3,830 records**, each corresponding to a candidate column-name expansion produced by the enricher. Each record consists of the following attributes:

- `table_name`: the name of the source table in the EDI corpus;
- `column_name`: the original column name extracted from the dataset;
- `gt_label`: the gold-standard expansion of the column name;
- `expansion`: the expansion generated by the Maverick model;
- `gold_label`: a binary indicator, where 1.0 denotes that the generated expansion matches the gold-standard expansion (accounting for synonyms), and 0.0 denotes a mismatch.

```
id,table_name,column_name,gt_label,expansion,gold_label
0,158_1734_tree_density_2002.txt,Site,site,,
1,158_1734_tree_density_2002.txt,Plot,plot,,
2,158_1734_tree_density_2002.txt,Black Spruce Adults,black spruce adults,Black Spruce Adults,1.0
3,158_1734_tree_density_2002.txt,Black Spruce Seedlings,black spruce seedlings,Black Spruce Seedlings,1.0
4,158_1734_tree_density_2002.txt,White Spruce Adults,white spruce adults,White Spruce Adults,1.0
5,158_1734_tree_density_2002.txt,White Spruce Seedlings,white spruce seedlings,White Spruce Seedlings,1.0
6,186_1590_Cones.txt,Site,site,Site,1.0
7,186_1590_Cones.txt,Plot,plot,Plot,1.0
8,186_1590_Cones.txt,Pmar ht,,Pmar Height,
9,186_1590_Cones.txt,Pmar Cones,,Pmar Cones,
10,197_1622_SEED.txt,plot,plot,Plot,1.0
11,197_1622_SEED.txt,AspT,aspen treatment,Aspect Trend/Type,0.0
```

Figure 5.6: Excerpt from the Smartcat-generated EDI column expansion dataset.

Figure 5.6 shows a snapshot of the dataset produced by Smartcat after applying the CNE enricher. As shown, some records contain missing values in the `gold_label` field. This occurs when a gold-standard expansion is unavailable, when the language model fails to generate an expansion,

or when the generated output cannot be reliably extracted. Since such records cannot be used to evaluate annotation correctness, we apply a simple preprocessing step to filter them out. Specifically, we remove all rows with missing values in the `gold_label` column. After filtering, **2,918 valid records** remain. These records form the input dataset for all our experiments reported in the following sections.

5.4.3 Experiments Across Multiple Dimensions

The seven experiments presented in this section perform the evaluation along multiple dimensions, including aggregation policy, worker trust, and interaction modality. Experiments 1–3 establish baseline behavior under bulk curation with increasingly robust aggregation and trust assumptions. Experiments 4–6 progressively introduce drive-by curation through the Smartcat API and evaluate mixed interaction scenarios. Finally, Experiment 7 solicits curation from a large-scale, externally sourced worker population. Together, these experiments demonstrate that Cymphony can be customized to support a broad range of Smartcat curation workflows through a stable, API-driven integration.

5.4.3.1 Experiments 1-3: Bulk Curation with Synthetic Workers and Stewards

Experiment 1 - Bulk Curation with Workers (2-out-of-3): Our first experiment establishes a baseline for bulk curation using only regular workers and a simple majority aggregation policy. The workflow consists of a single `3a_knlm` operator configured with a 2-out-of-3 voting rule for regular workers along with a 1-out-of-1 aggregation policy for stewards, applied to the full preprocessed EDI dataset. Each task asks whether the enricher-generated expansion is a correct semantic expansion of the given column name. This experiment serves as a reference point for evaluating the effects of redundancy and role-aware customization in subsequent runs.

Regular worker arrivals follow a steady process, with one worker arriving every two minutes, up to a maximum of 100 workers. Each worker annotates tasks at a fixed rate of one task every 10 seconds. Worker accuracy is sampled uniformly between 80% and 85%, modeling moderately reliable but noisy contributors.

During execution, a total of 34 workers actively contribute annotations. Workers complete between 3 and 393 annotations each, with an average of 198.2 annotations per worker. Observed worker precision ranges from 0.71 to 1.0, with an overall average precision of 0.8165, closely matching the configured accuracy distribution.

At completion, all 2,918 tasks converge to final labels, with no undecided cases. The aggregated outputs contain 2,333 positive labels and 585 negative labels. Comparing against gold labels yields 2,296 true positives, 37 false positives, 212 false negatives, and 373 true negatives, corresponding to an overall precision (and recall) of 0.9146. The end-to-end execution time for the run is 1 hour, 6 minutes, and 59 seconds.

This experiment confirms that the customized `3a_knlm` operator supports large-scale bulk curation through the CN–SC API boundary, achieving strong accuracy and task convergence under realistic worker behavior.

Experiment 2 - Bulk Curation with Workers (3-out-of-5): The second experiment evaluates the effect of increased annotation redundancy on label quality and execution time. We use the same dataset, worker arrival process, and worker accuracy model as in Experiment 1, but increase the aggregation requirement to a 3-out-of-5 majority policy for regular workers.

The higher redundancy results in greater overall worker participation. A total of 42 workers contribute annotations, with each worker completing between 6 and 489 tasks and an average of 247.8 annotations per worker. Observed worker precision remains stable, ranging from 0.74 to 0.92, with an average precision of 0.8186 across all annotations.

As in Experiment 1, all 2,918 tasks converge to final labels without undecided cases. The final aggregation produces 2,422 positive labels and 496 negative labels. Relative to the gold standard, this corresponds to 2,394 true positives, 28 false positives, 114 false negatives, and 382 true negatives. The resulting precision (and recall) improves to 0.9513, demonstrating the benefit of additional redundancy. However, this improvement comes at the cost of increased execution time: the run completes in 1 hour, 23 minutes, and 25 seconds.

This experiment highlights a fundamental trade-off: higher annotation redundancy improves label quality but increases annotation volume and end-to-end latency. Importantly, these changes

are realized solely by modifying workflow parameters, without requiring any changes to Smartcat-side logic or CN internals.

Experiment 3 - Bulk Curation with Workers and Stewards (2-out-of-3; 1-out-of-1): The third experiment evaluates the impact of role-aware curation by introducing high-trust stewards alongside regular workers. The workflow again uses a 2-out-of-3 aggregation policy for regular workers, along with a 1-out-of-1 aggregation policy for stewards, identical to Experiment 1.

Regular worker behavior follows the same arrival and accuracy model as before. In addition, two stewards participate in the run. The first steward begins annotating immediately and remains active for one hour, while the second steward joins after 30 minutes and annotates for two hours. Both stewards annotate at the same rate as regular workers but with a higher accuracy of approximately 95%.

During execution, 31 regular workers contribute annotations, completing between 7 and 362 tasks each, with an average of 184.2 annotations per worker. Their observed average precision is 0.8301. The two stewards contribute 355 and 183 annotations, respectively, with observed precisions of 0.9352 and 0.9398.

All 2,918 tasks again converge without undecided cases. The final outputs contain 2,361 positive labels and 557 negative labels. Against the gold standard, this yields 2,327 true positives, 34 false positives, 181 false negatives, and 376 true negatives, corresponding to a precision (and recall) of 0.9263. Notably, the precision (and recall) improved from 0.9146 to 0.9263, and the end-to-end execution time decreases to 1 hour, 1 minute, and 38 seconds, despite comparable worker arrival rates.

This experiment demonstrates the effectiveness of role-aware customization. By allowing high-trust stewards to participate alongside regular workers, Cymphony achieves faster convergence and improved accuracy without increasing annotation redundancy. Crucially, this behavior emerges naturally from the customized operator semantics and does not require any special handling in the Smartcat client.

Discussion: Table 5.1 summarizes the highest-level outcomes; all experiments were conducted under identical datasets and worker arrival models, with detailed parameters and statistics reported in the preceding paragraphs.

Table 5.1: Comparison of bulk curation experiments.

	Exp. 1	**Exp. 2**	**Exp. 3**
Aggregation policy	$2/3$ (rg.) + $1/1$ (st.)	$3/5$ (rg.) + $1/1$ (st.)	$2/3$ (rg.) + $1/1$ (st.)
Regular workers (rg.)	34	42	31
Stewards (st.)	0	0	2
Final precision	0.9146	0.9513	0.9263
Run completion time	1:06:59	1:23:25	1:01:38

Taken together, Experiments 1–3 validate the CN–SC customization under a range of bulk curation configurations while holding the dataset, worker arrival process, and Smartcat-facing API contract constant. Across all three runs, Cymphony successfully executes large-scale curation workloads, produces resolved labels for all 2,918 tasks, and exposes results through the API-driven interaction model.

Experiment 1 establishes a baseline using a simple 2-out-of-3 majority policy with regular workers, achieving strong accuracy with moderate latency. Experiment 2 demonstrates that increasing annotation redundancy to 3-out-of-5 further improves precision, at the cost of additional annotations and longer end-to-end execution time. This trade-off emerges naturally from workflow parameterization and does not require any changes to Smartcat-side logic or Cymphony's execution infrastructure.

Experiment 3 illustrates the benefits of role-aware customization. By introducing a small number of high-trust stewards while retaining the same aggregation policy as Experiment 1, the system achieves faster convergence and improved label quality. Notably, these gains are realized without increasing redundancy or annotation volume, demonstrating that Cymphony's customized operators can effectively interleave heterogeneous worker roles within a single run.

Collectively, these experiments confirm that the CN–SC integration supports a broad spectrum of bulk curation strategies. More importantly, they show that Smartcat can explore these design points entirely through workflow configuration and API usage, without embedding curation-specific logic into the catalog frontend. This separation of concerns is central to the customization goals of this chapter and sets the stage for the drive-by and mixed-interaction experiments presented next.

5.4.3.2 Experiments 4–6: Drive-by and Mixed Curation with Synthetic Workers and Stewards

Experiments 4–6 evaluate Cymphony's customization for Smartcat under increasingly complex interaction patterns. These experiments more closely resemble real catalog usage, where annotations arrive opportunistically and are interleaved with structured bulk curation.

Experiment 4 – Pure Drive-by Curation (2-out-of-3): The fourth experiment isolates drive-by curation as the sole source of annotations. Smartcat-side users perform lightweight, opportunistic annotations while browsing the catalog, and their annotations are forwarded to Cymphony through the *Drive-by Curation* API. The workflow uses the same `3a_knlm` operator as in Experiment 3.

Fifty synthetic Smartcat users participate in the simulation, each arriving once, annotating exactly ten tuples, and never returning. Annotations are accumulated client-side and submitted in batches, modeling a simple but realistic drive-by interaction pattern. Worker accuracy is again sampled between 80% and 85%.

Across the run, 500 drive-by annotations are submitted, covering 467 distinct tuples. Because drive-by annotations are sparse and unevenly distributed across tasks, only 20 tuples receive sufficient votes to satisfy the aggregation criterion. All converged tasks are labeled correctly, yielding perfect precision on the small subset of tasks that complete. The remaining tasks do not converge due to insufficient redundancy, rather than conflicting votes.

This experiment demonstrates that the CN–SC integration correctly supports asynchronous, opportunistic annotation submission and incremental aggregation. Importantly, the system remains stable even when the majority of tasks do not converge, and partial results can still be retrieved

without error. This behavior matches expectations for pure drive-by curation and highlights the need for complementary bulk annotation in practice.

Experiment 5 – Mixed Drive-by and Bulk Curation (2-out-of-3): The fifth experiment combines drive-by curation from Smartcat with bulk curation performed by regular Cymphony workers. The workflow and aggregation policy remain unchanged, allowing us to directly observe how the two interaction modes interleave.

Drive-by annotations are generated under the same assumptions as in Experiment 4, while bulk regular workers arrive every two minutes and annotate tasks continuously throughout the run following the same pattern as in Experiment 3. Drive-by annotations interleave with bulk worker annotations as they arrive.

Before run completion, 33 Smartcat users submit 330 drive-by annotations, covering 309 distinct tuples. Across the full simulation, all 50 Smartcat users submit 500 annotations. In parallel, 33 regular Cymphony workers contribute bulk annotations with an average precision of approximately 82%. All drive-by annotations are ingested through the same job-scoped output tables as bulk annotations and are treated uniformly by the aggregation logic.

By the end of the run, all 2,918 tasks converge to final labels with no undecided cases. The final precision of 0.9242 closely matches the bulk-only baseline from Experiment 1, while the run completion time reduces slightly to 1 hour, 6 minutes, and 3 seconds. These results show that drive-by annotations can accelerate early progress on tasks without destabilizing aggregation or degrading final label quality.

This experiment confirms that Cymphony's customization cleanly supports mixed interaction modes. Drive-by and bulk annotations coexist without requiring special handling in either system, validating the decision to unify them at the job-scoped output layer.

Experiment 6 – Mixed Drive-by, Bulk, and Steward Curation (2-out-of-3; 1-out-of-1): The sixth experiment extends the mixed scenario by introducing high-trust stewards in addition to regular bulk and drive-by workers. This configuration represents the most realistic catalog curation setting.

Drive-by annotations are generated under the same assumptions as in Experiment 4, while bulk regular workers and stewards follow the same pattern as they did in Experiment 3. Drive-by annotations interleave with bulk annotations as they arrive.

Drive-by curation proceeds as before, with 50 Smartcat users contributing 500 annotations in total. Bulk curation is performed by 31 regular workers, while two stewards participate with significantly higher accuracy (approximately 95%).

Across the run, steward annotations rapidly resolve many tasks that would otherwise require additional worker votes. Despite the increased complexity of the interaction pattern, all tasks again converge without undecided cases, and with a final precision of 0.9208 that remains high and comparable to earlier experiments, while the run completes faster than the corresponding mixed scenario without stewards.

This experiment demonstrates the full expressive power of the CN–SC customization. Heterogeneous annotation sources with different trust levels are interleaved seamlessly, and the system exploits high-quality annotations to accelerate convergence without increasing redundancy. Crucially, Smartcat remains oblivious to these internal execution details.

Table 5.2: Comparison of drive-by and mixed curation experiments.

	Exp. 4	**Exp. 5**	**Exp. 6**
Curation modes	Drive-by	Drive-by + bulk rg.	Drive-by + bulk rg. & st.
Aggregation policy	$2/3$ (rg.) + $1/1$ (st.)	$2/3$ (rg.) + $1/1$ (st.)	$2/3$ (rg.) + $1/1$ (st.)
SC drive-by users	50	50	50
Regular workers (rg.)	0	33	31
Stewards (st.)	0	0	2
Tasks converged	20 / 2918	2918 / 2918	2918 / 2918
Final precision	1.0000[†]	0.9242	0.9208
Run completion time	1:40:15	1:06:03	1:00:54

[†]Precision computed over the small subset of tasks that received sufficient drive-by annotations to converge.

Discussion: Table 5.2 summarizes the outcomes of Experiments 4–6, which progressively introduce drive-by curation and mixed interaction modes while holding the dataset, aggregation policy, and API contract constant. Together, these experiments evaluate the CN–SC customization under increasingly realistic Smartcat interaction patterns.

Experiment 4 isolates drive-by curation through the Smartcat API. As shown in Table 5.2, only a small fraction of tasks converge when annotations arrive exclusively through sparse, opportunistic drive-by interactions. This outcome is expected and confirms that drive-by signals alone are insufficient for full dataset coverage, but can still produce highly accurate labels for the limited set of tasks that receive enough votes.

Experiment 5 combines drive-by curation with bulk curation by regular workers. In this mixed setting, drive-by annotations contribute signals that are naturally absorbed into Cymphony's aggregation process, while bulk workers ensure full task coverage. As reflected in Table 5.2, this configuration achieves complete convergence with strong precision and reduced end-to-end latency relative to bulk-only baselines, demonstrating that the CN–SC integration cleanly supports interleaving asynchronous annotation streams.

Experiment 6 further introduces high-trust stewards into the mixed interaction scenario. The addition of stewards improves convergence speed while maintaining high precision, despite comparable annotation volumes. Crucially, these benefits arise without changing the Smartcat-side client or the integration protocol; all behavior is driven by workflow configuration and role-aware operator semantics within Cymphony.

Taken together, Experiments 4–6 confirm that the CN–SC integration supports a broad spectrum of realistic catalog curation behaviors. Pure drive-by curation yields correct but sparse results, mixed drive-by and bulk curation achieves full coverage without sacrificing accuracy, and the addition of stewards further improves efficiency and latency. Differences in system behavior emerge solely from workflow configuration and participant composition, rather than from bespoke integration logic. These results reinforce the central claim of this chapter: Cymphony can be customized to support Smartcat-style curation workflows in a principled, modular, and extensible manner, while preserving clean service boundaries and execution semantics. This flexibility is central to the design goals of the customization and sets the stage for utilizing large external worker populations in the next experiment.

5.4.3.3 Experiment 7: Large-Scale Turker Deployment

The final experiment evaluates the CN–SC integration under an externally sourced worker population. Unlike the previous experiments, which rely on synthetic workers and controlled arrival processes, this experiment deploys a Cymphony workflow on Amazon Mechanical Turk (AMT) and measures system behavior, cost, and label quality.

Dataset Preparation: We begin with the Smartcat-generated EDI dataset introduced earlier, containing 3,830 candidate column-name expansions. After filtering records with missing gold labels, 2,918 labeled records remain (2,508 positive and 410 negative). To support a cost-effective yet representative Turker deployment, we construct a curated evaluation dataset of 1,100 records as follows.

First, we create a *tightly representative* subset of 100 records with an approximately balanced label distribution. This subset is obtained by sampling one tuple per unique table name from both positive and negative examples and further subsampling to achieve a 50–50 split. Second, we construct a *loosely representative* subset of 1,000 records by randomly shuffling the remaining data and selecting the first 1,000 rows, preserving the natural skew of the dataset. The two subsets are combined to form a final dataset of 1,100 records, with 915 positive and 185 negative examples. Gold-label columns are removed prior to submission to workers.

Task Design and Workflow: Workers are presented with a tabular view showing a column name and its proposed expansion and are asked to determine whether the expansion is correct. Each task offers three response options: *Yes*, *No*, and *Cannot Determine*. Figure 5.7 shows the worker-facing interface, while Figures 5.8–5.10 present the instruction templates.

The workflow consists of a single `3a_amt` operator configured with a 2-out-of-3 aggregation policy. Each HIT contains one task, pays $0.03 per annotation, and is restricted to US-based Master workers. All integration logic—including task publication, result aggregation, and output materialization—is handled entirely by Cymphony through its standard AMT Manager.

Execution Statistics: The run completes in 6 hours and 7 minutes at a total cost of $132. A total of 15 Turkers participate, contributing between 2 and 759 votes each, with an average of 220

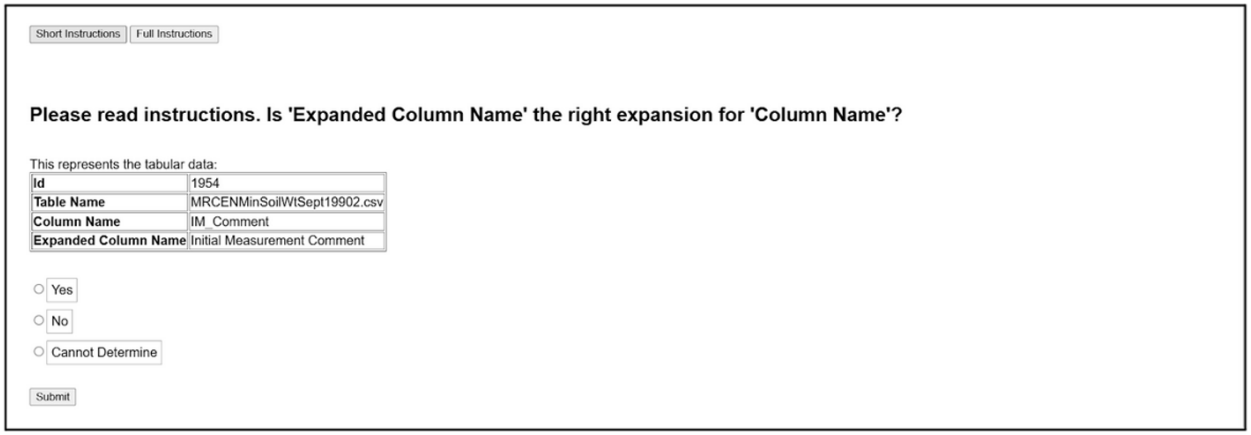

Figure 5.7: Worker Facing Interface.

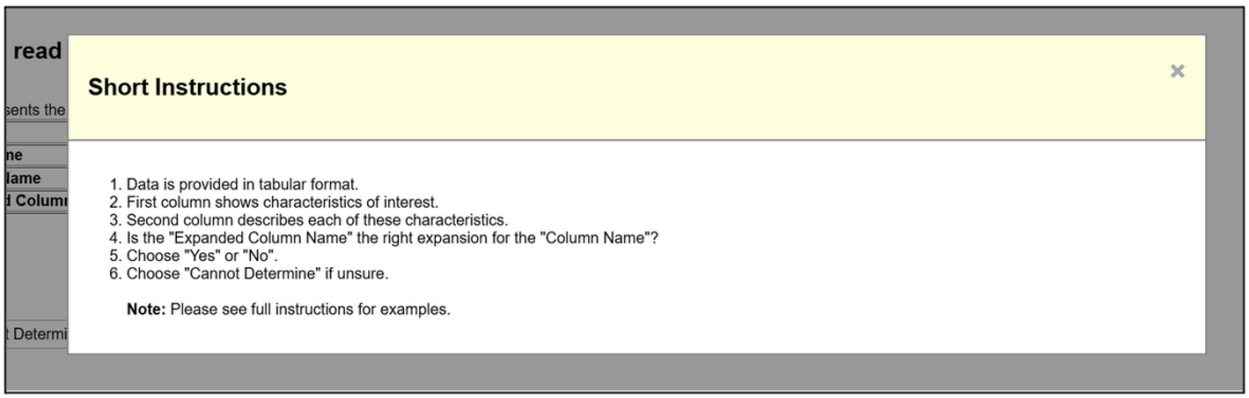

Figure 5.8: Short Instructions.

votes per worker. After excluding *Cannot Determine* responses, the average per-worker prediction precision is 0.8687.

At convergence, 1,074 tasks receive decisive labels, 22 remain undecided, and 4 resolve to *Cannot Determine*. Relative to the gold standard, the final outputs achieve a precision of **0.8771** and a recall of **0.8564**.

More fine-grained analysis reveals a clear asymmetry in worker signals. When Turkers answer *Yes*, they overwhelmingly do so on truly positive examples. Specifically, *Yes* responses correspond to 98.58% of all positive cases in the dataset, closely matching the ideal behavior expected from high-quality workers. At the same time, *Yes* responses also account for 70.81% of the negative

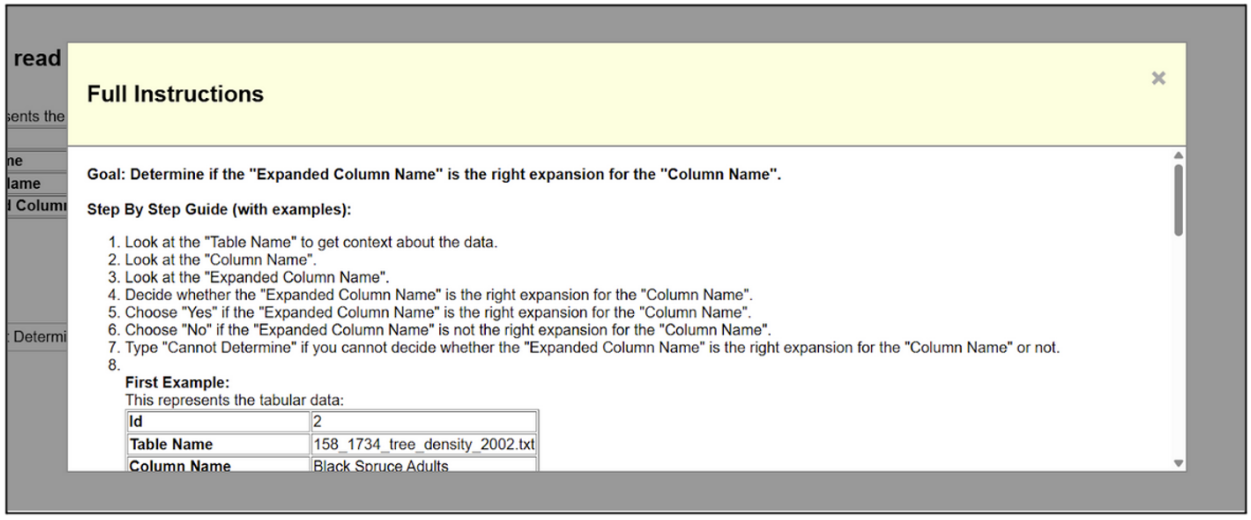

Figure 5.9: Full Instructions.

examples, reflecting the inherent ambiguity of negative cases in this task and the conservative nature of rejecting expansions.

Conversely, when Turkers answer *No*, *Cannot Determine*, or fail to reach consensus, these outcomes predominantly correspond to negative examples. Together, these non-affirmative signals capture 29.19% of all negative cases, while misclassifying only 1.42% of positive examples. This asymmetry indicates that affirmative votes are highly precise, while non-affirmative outcomes serve as a useful—though noisier—filter for problematic or ambiguous expansions.

Discussion: This experiment demonstrates that Cymphony can support a wide range of curation tasks relevant to Smartcat and beyond, including real-world crowdsourcing with heterogeneous worker behavior and explicit uncertainty. Crucially, this capability is achieved without any modification to Cymphony's core execution model. From a systems perspective, the AMT experiment on top of the earlier experiments reinforces the generality and robustness of the Cymphony platform.

5.4.4 Conclusion

Taken together, the experiments in this chapter provide a comprehensive evaluation of the CN–SC integration across aggregation policies, worker roles, interaction modalities, and execution environments. Across synthetic bulk curation, drive-by curation, mixed interaction scenarios,

1. Data is provided in tabular format.
2. First column shows characteristics of interest.
3. Second column describes each of these characteristics.
4. Is the "Expanded Column Name" the right expansion for the "Column Name"?
5. Choose "Yes" or "No".
6. Choose "Cannot Determine" if unsure.

Note: Please see full instructions for examples.

Goal: Determine if the "Expanded Column Name" is the right expansion for the "Column Name".

Step By Step Guide (with examples):

1. Look at the "Table Name" to get context about the data.
2. Look at the "Column Name".
3. Look at the "Expanded Column Name".
4. Decide whether the "Expanded Column Name" is the right expansion for the "Column Name".
5. Choose "Yes" if the "Expanded Column Name" is the right expansion for the "Column Name".
6. Choose "No" if the "Expanded Column Name" is not the right expansion for the "Column Name".
7. Type "Cannot Determine" if you cannot decide whether the "Expanded Column Name" is the right expansion for the "Column Name" or not.
8.

First Example:

This represents the tabular data:

Id	2
Table Name	158_1734_tree_density_2002.txt
Column Name	Black Spruce Adults
Expanded Column Name	Black Spruce Adults

- From the table name, we can infer that the data seems to be about tree density in 2002.
- From the column name, we can infer that the column does not have any abbreviation, and seems to be about Black Spruce Adults.
- From the expanded column name, we can infer that the expanded column name is the same as the column name.
- **Since the column name did not have any abbreviation, and the expanded column name is the same as the column name, we conclude that the expanded column name is the right expansion for the column name.**
- Hence, we will choose "Yes".

Second Example:

This represents the tabular data:

Id	42
Table Name	1988gssabm.csv
Column Name	Average g/m^2
Expanded Column Name	Average Grams per Square Meter

- From the table name, we cannot infer anything since it is cryptic.
- From the column name, we can see some abbreviations, and it seems to be saying Average gram/meter^2.
- From the expanded column name, we can infer that the expanded column name is "Average Grams per Square Meter" which seems to be the correct expansion for the column name.
- **Since the expanded column name seems to be the right expansion for the column name, we will choose "Yes".**

Third Example:

This represents the tabular data:

Id	404
Table Name	410_LogDecompDynamicsIntAK3_DiskData.txt
Column Name	LD0
Expanded Column Name	LD0

- From the table name, we can infer that the data seems to be about log decomposition dynamics for disk data.
- From the column name, we can infer that the column name is LD0 which seems to be abbreviated, but doesn't tell us anything more.
- From the expanded column name, we can infer that the expanded column name is the same as the abbreviated column name. It did not expand the abbreviation correctly.
- **Since the expanded column name does not seem to expand the column name correctly, we will choose "No".**

Fourth Example:

This represents the tabular data:

Id	3643
Table Name	occurrences.csv
Column Name	verbatimEventDate
Expanded Column Name	Unprocessed Event Date

- From the table name, we can infer that the data seems to be about occurrences.
- From the column name, we can infer that the column name is verbatimEventDate, and that it does not have any abbreviation.
- From the expanded column name, we can infer that the expanded column name is Unprocessed Event Date.
- Since the column name was not abbreviated, its expansion should have been "verbatimEventDate" or "verbatim event date", and not "Unprocessed Event Date".
- **Since the expanded column name does not seem to expand the column name correctly, we will choose "No".**

Figure 5.10: Complete View of Full Instructions.

and external crowdsourcing with turkers, Cymphony supports diverse workloads without requiring changes to Cymphony's core execution model.

These results demonstrate that Smartcat can vary redundancy, introduce role-aware trust, interleave drive-by and bulk annotations, and scale from in-house workers to external crowdsourcing entirely through workflow configuration and API usage. At the same time, Cymphony consistently provides deterministic execution, robust aggregation, and asynchronous coordination under realistic timing and cost constraints.

More broadly, this evaluation validates the central thesis of the customization: that our general-purpose crowdsourcing engine can be adapted to a domain-specific data catalog through principled customizations rather than bespoke integration logic. By separating catalog-side orchestration from backend execution semantics, the CN–SC integration enables rapid experimentation on curation strategies while preserving system modularity and long-term maintainability. The lessons drawn from this customization inform not only the Smartcat integration, but also provide guidance for adapting Cymphony to other data management systems.

Chapter 6

Conclusions

Crowdsourcing has become a foundational technique in modern data integration (DI) workflows. Yet despite its widespread practical use, crowdsourcing remains poorly systematized. In many settings, human annotation is treated as an external service, an afterthought, or a collection of scripts layered on top of existing data pipelines. This dissertation has argued that such an approach is fundamentally limiting. Instead, we have proposed that crowdsourcing for data integration should be treated as a first-class systems problem—one that requires explicit abstractions, principled execution semantics, and rigorous evaluation.

This chapter summarizes the main contributions of this work, reflects on the broader lessons that emerged during the design and evaluation of Cymphony, and outlines directions for future research.

6.1 Summary of Contributions

This dissertation introduced **Cymphony**, a general-purpose crowdsourcing platform tailored to data integration workflows. The central goal was not merely to support human annotation, but to elevate it into a structured execution substrate that integrates cleanly with larger data systems.

First, we proposed an *operator-based abstraction* for modeling human–machine workflows. Workflows are expressed declaratively as directed acyclic graphs (DAGs) of operators, where each operator encapsulates well-defined transformations over relational artifacts. Human operators manage assignment, annotation, and aggregation internally, while machine operators perform deterministic transformations such as sampling and SQL-based processing. This design decouples

workflow specification from execution mechanics, enabling complex multi-stage pipelines to be expressed without exposing low-level coordination logic.

Second, we designed and implemented a complete execution stack for human-in-the-loop processing. Cymphony supports task instantiation, worker coordination, annotation collection, aggregation of noisy labels, and output materialization. All execution artifacts are materialized as relational tables, ensuring transparency, reproducibility, and composability with traditional data processing tools. The system also integrates with external marketplaces such as Amazon Mechanical Turk (AMT), enabling heterogeneous worker populations to participate in a unified workflow.

Third, we conducted a comprehensive evaluation of Cymphony along both functional and scalability dimensions. End-to-end experiments on representative DI tasks—including information extraction, column classification, and entity matching with active learning—demonstrated that the system faithfully captures real-world human–machine workflows while achieving high label quality at reasonable cost. Complementary scalability experiments, driven by a simulator that exercised the full execution stack, showed predictable scaling with dataset size and worker concurrency, and sufficient performance under single-node deployment for practical workloads.

Finally, we demonstrated that Cymphony's abstractions extend beyond stand-alone experiments by integrating it into a production-scale data catalog system, Smartcat. Through an API-driven architecture, Smartcat delegates the orchestration of curation workflows to Cymphony while retaining control over user interaction and metadata management. We further introduced role-aware aggregation semantics to support heterogeneous worker types—trusted stewards, organizational users, and external crowd workers—within a unified workflow that supports multiple interaction patterns required by data catalog systems like Smartcat. This case study illustrates that crowdsourcing can serve as a reusable backend service for larger data management systems.

6.2 Key Lessons and Broader Takeaways

Across the design, implementation, and evaluation of Cymphony, several broader insights emerged.

Crowdsourcing is a Systems Problem: A primary lesson is that crowdsourcing cannot be treated solely as an algorithmic or marketplace problem. While aggregation rules and quality-control techniques are important, real-world deployments depend equally on orchestration, state management, integration interfaces, and reproducibility. Many of the practical challenges we encountered—workflow composition, failure handling, intermediate state tracking, and heterogeneous worker coordination—are fundamentally systems challenges. Treating crowdsourcing as a first-class execution substrate provides the right conceptual foundation for addressing these issues holistically.

Declarative Abstractions Enable Composability: The operator-based, DAG-oriented design of Cymphony proved critical for composability and extensibility. By encapsulating coordination logic within operators and exposing clean input/output interfaces, we were able to interleave human and machine steps seamlessly. This mirrors the evolution of relational query processing: once complex operations are expressed declaratively, they can be reasoned about, composed, and potentially optimized in principled ways. Even in the presence of inherently non-deterministic human behavior, a declarative workflow model provides structure and clarity.

Materialization Improves Transparency and Reproducibility: Another important takeaway is the value of materializing all execution artifacts as relational tables. This design decision simplified debugging, enabled downstream analysis, and ensured reproducibility of experiments. In contrast to ad-hoc scripts where annotations and intermediate states are transient or opaque, Cymphony's explicit data model makes every stage of execution inspectable. For research settings in particular, such transparency is essential for scientific rigor.

Heterogeneity is the Norm, Not the Exception: Real-world curation rarely involves a single homogeneous worker pool. Instead, systems must accommodate trusted stewards, casual internal users performing drive-by validation, and external crowd workers with varying reliability. Supporting heterogeneous worker types required rethinking aggregation semantics and interaction modes. The Smartcat integration highlighted that workflow engines must be flexible enough to accommodate both structured bulk sessions and opportunistic, inline interactions within a host system.

Designing abstractions that generalize across these modes was essential to making Cymphony practically usable.

Evaluation Must Reflect End-to-End Reality: Finally, the evaluation experience reinforced that human-in-the-loop systems must be evaluated end-to-end. Measuring only aggregation accuracy or only throughput misses important interactions between workflow design, worker behavior, and system performance. By combining real-worker experiments with controlled simulation-based scalability studies, we were able to isolate and understand different dimensions of performance. This dual methodology provides a template for future evaluation of human–machine systems.

6.3 Future Directions

While this dissertation establishes a foundation for systematized crowdsourcing in data integration, several avenues remain open.

Adaptive and Learning-Aware Workflows: Current workflows rely primarily on user-specified parameters for aggregation thresholds and task sequencing. Integrating learning-based components that adaptively adjust these parameters based on observed worker behavior or task difficulty could further improve cost-quality trade-offs. Combining declarative workflow specification with feedback-driven optimization remains a promising direction.

Deeper Integration with Data Management Stacks: Cymphony currently integrates with external systems via APIs. Exploring tighter integration with query optimizers, metadata managers, and lineage systems could enable cross-layer optimization. For example, workflow cost estimates could inform upstream sampling strategies or downstream data-quality policies.

Richer Interaction Models: Supporting richer forms of interaction—such as iterative refinement, visual analytics for error inspection, or collaborative annotation among workers—could extend the expressive power of the framework. Bringing interactive, human-centered design more deeply into the operator model could be an important step toward fully integrated human–machine data systems.

Generalization Beyond Data Integration: Although Cymphony was designed with DI tasks in mind, its abstractions are applicable to other domains that require structured human coordination, such as model evaluation, content moderation, or knowledge-base construction. Investigating these domains may uncover new operator types and aggregation semantics that further generalize the framework.

6.4 Closing Remarks

This dissertation has argued that crowdsourcing for data integration should not remain an ad-hoc collection of scripts and marketplace interactions. By introducing declarative abstractions, explicit execution semantics, and a complete system implementation, we have demonstrated that human-in-the-loop processing can be treated as a principled data-management substrate.

More broadly, this work suggests a shift in perspective: rather than viewing human annotation as external to the data system, we should treat it as an integral component—one that can be specified, executed, reasoned about, and evaluated with the same rigor as any other part of the data stack. As data systems continue to grow in scale and complexity, such integration will be essential for building robust, extensible, and trustworthy human–machine platforms.

Bibliography

[1] O. Alonso. *The Practice of Crowdsourcing*. Synthesis Lectures on Information Concepts, Retrieval, and Services. Morgan & Claypool Publishers, 2019.

[2] Amazon Web Services. Amazon Mechanical Turk. `https://www.mturk.com/`, 2005. Accessed: 2025.

[3] Appen Limited. Appen: AI Training Data. `https://www.appen.com/`, 2026. Accessed: 2026.

[4] P. A. Bernstein, J. Madhavan, and E. Rahm. Generic schema matching, ten years later. *Proceedings of the VLDB Endowment*, 4(11):695–701, 2011.

[5] T. Cai, S. Sheen, and A. Doan. Columbo: Expanding abbreviated column names for tabular data using large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, 2025. arXiv:2508.09403.

[6] C. Chai, J. Fan, G. Li, J. Wang, and Y. Zheng. Crowdsourcing database systems: Overview and challenges. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 2052–2055, 2019.

[7] X. Chai, B.-Q. Vuong, A. Doan, and J. F. Naughton. Efficiently incorporating user feedback into information extraction and integration programs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 87–100, 2009.

[8] V. Christophides, V. Efthymiou, T. Palpanas, G. Papadakis, and K. Stefanidis. An overview of end-to-end entity resolution for big data. *ACM Computing Surveys*, 53(6):127:1–127:42, 2021.

[9] X. Chu, I. F. Ilyas, S. Krishnan, and J. Wang. Data cleaning: Overview and emerging challenges. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2201–2206, 2016.

[10] Clickworker GmbH. Clickworker. `https://www.clickworker.com/`, 2005. Accessed: 2025.

[11] F. Daniel, P. Kucherbaev, C. Cappiello, B. Benatallah, and M. Allahbakhsh. Quality control in crowdsourcing: A survey of quality attributes, assessment techniques, and assurance actions. *ACM Computing Surveys*, 51(1):7:1–7:40, 2018.

[12] S. Das, P. S. G. C., A. Doan, J. F. Naughton, G. Krishnan, R. Deep, E. Arcaute, V. Raghavendra, and Y. Park. Falcon: Scaling up hands-off crowdsourced entity matching to build cloud services. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1431–1446, 2017.

[13] P. DeRose, X. Chai, B. J. Gao, W. Shen, A. Doan, P. Bohannon, and X. Zhu. Building community wikipedias: A machine-human partnership approach. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 646–655, 2008.

[14] P. DeRose, W. Shen, F. Chen, A. Doan, and R. Ramakrishnan. Building structured web community portals: A top-down, compositional, and incremental approach. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 399–410, 2007.

[15] A. Doan. Human-in-the-loop data analysis: A personal perspective. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics (HILDA@SIGMOD)*, pages 1:1–1:6, 2018.

[16] A. Doan, M. J. Franklin, D. Kossmann, and T. Kraska. Crowdsourcing applications and platforms: A data management perspective. In *Proceedings of the VLDB Endowment*, volume 4, pages 1508–1509, 2011.

[17] A. Doan, A. Y. Halevy, and Z. G. Ives. *Principles of Data Integration*. Morgan Kaufmann, 2012.

[18] A. Doan, P. Konda, P. S. G. C., A. Ardalan, J. R. Ballard, S. Das, Y. Govind, H. Li, P. Martinkus, S. Mudgal, E. Paulson, and H. Zhang. Toward a system building agenda for data integration (and data science). *IEEE Data Engineering Bulletin*, 41(2):35–46, 2018.

[19] A. Doan and R. McCann. Building data integration systems: A mass collaboration approach. In *Proceedings of the IJCAI-03 Workshop on Information Integration on the Web (IIWeb)*, pages 183–188, 2003.

[20] A. Doan, R. Ramakrishnan, F. Chen, P. DeRose, Y. Lee, R. McCann, M. Sayyadian, and W. Shen. Community information management. *IEEE Data Engineering Bulletin*, 29(1):64–72, 2006.

[21] A. Doan, R. Ramakrishnan, and A. Y. Halevy. Crowdsourcing systems on the World-Wide Web. *Communications of the ACM*, 54(4):86–96, 2011.

[22] X. L. Dong and T. Rekatsinas. Data integration and machine learning: A natural synergy. *Proceedings of the VLDB Endowment*, 11(12):2094–2097, 2018.

[23] X. L. Dong and D. Srivastava. *Big Data Integration*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2015.

[24] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):1–16, 2007.

[25] Environmental Data Initiative. Environmental data initiative (EDI) repository. `https://edirepository.org/`, 2026. Accessed: 2026.

[26] J. Fan, G. Li, B. C. Ooi, K.-L. Tan, and J. Feng. iCrowd: An adaptive crowdsourcing framework. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1015–1030, 2015.

[27] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: Answering queries with crowdsourcing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 61–72, 2011.

[28] L. Getoor and A. Machanavajjhala. Entity resolution: Theory, practice & open challenges. *Proceedings of the VLDB Endowment*, 5(12):2018–2019, 2012.

[29] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. W. Shavlik, and X. Zhu. Corleone: Hands-off crowdsourcing for entity matching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 601–612, 2014.

[30] J. Howe. The rise of crowdsourcing. *Wired Magazine*, 14(6):176–183, 2006.

[31] I. F. Ilyas and X. Chu. *Data Cleaning*. ACM Books / Morgan & Claypool, 2019.

[32] iMerit. iMerit: AI Data Solutions. `https://www.imerit.net/`, 2026. Accessed: 2026.

[33] P. G. Ipeirotis, F. Provost, and J. Wang. Quality management on Amazon Mechanical Turk. In *Proceedings of the ACM SIGKDD Workshop on Human Computation (HCOMP)*, pages 64–67, 2010.

[34] A. Kittur, J. V. Nickerson, M. S. Bernstein, E. M. Gerber, A. Shaw, J. Zimmerman, M. Lease, and J. J. Horton. The future of crowd work. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work and Social Computing (CSCW)*, pages 1301–1318, 2013.

[35] P. Konda, S. Das, P. S. G. C., A. Doan, A. Ardalan, J. R. Ballard, H. Li, F. Panahi, H. Zhang, J. F. Naughton, S. Prasad, G. Krishnan, R. Deep, and V. Raghavendra. Magellan: Toward building entity matching management systems. *Proceedings of the VLDB Endowment*, 9(12):1197–1208, 2016.

[36] G. Li, J. Wang, Y. Zheng, and M. J. Franklin. Crowdsourced data management: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 28(9):2296–2319, 2016.

[37] G. Li, Y. Zheng, J. Fan, J. Wang, and R. Cheng. Crowdsourced data management: Overview and challenges. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1711–1716, 2017.

[38] A. Marcus, E. Wu, D. R. Karger, S. Madden, and R. C. Miller. Crowdsourced databases: Query processing with people. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2011.

[39] R. McCann, A. Doan, V. Varadarajan, and A. Kramnik. Building data integration systems via mass collaboration. In *Proceedings of the International Workshop on Web and Databases (WebDB)*, pages 25–30, 2003.

[40] R. McCann, A. Kramnik, W. Shen, V. Varadarajan, O. Sobulo, and A. Doan. Integrating data from disparate sources: A mass collaboration approach. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 487–488, 2005.

[41] R. McCann, W. Shen, and A. Doan. Matching schemas in online communities: A Web 2.0 approach. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 110–119, 2008.

[42] Meta AI. The Llama 4 herd: The beginning of a new era of natively multimodal AI. `https://ai.meta.com/blog/llama-4-multimodal-intelligence/`, 2025. Includes Llama 4 Maverick. Accessed: 2025.

[43] Microworkers. Microworkers: Crowdsourcing Platform. `https://www.microworkers.com/`, 2024. Accessed: 2025.

[44] B. Mozafari, P. Sarkar, M. Franklin, M. Jordan, and S. Madden. Scaling up crowd-sourcing to very large datasets: A case for active learning. *Proceedings of the VLDB Endowment*, 8(2):125–136, 2014.

[45] S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, R. Deep, E. Arcaute, and V. Raghavendra. Deep learning for entity matching: A design space exploration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 19–34, 2018.

[46] A. G. Parameswaran, H. Garcia-Molina, H. Park, N. Polyzotis, A. Ramesh, and J. Widom. CrowdScreen: Algorithms for filtering data with humans. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 361–372, 2012.

[47] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[48] Sama. Sama: AI Training Data. `https://www.sama.com/`, 2024. Accessed: 2025.

[49] S. Sarawagi. Information extraction. *Foundations and Trends in Databases*, 1(3):261–377, 2008.

[50] B. Settles. *Active Learning Literature Survey*. University of Wisconsin–Madison, 2009. Computer Sciences Technical Report 1648.

[51] P. Subramaniam, Y. Ma, C. Li, I. Mohanty, and R. C. Fernandez. Comprehensive and comprehensible data catalogs: The what, who, where, when, why, and how of metadata management, 2023. arXiv:2103.07532.

[52] C. Sun, N. Rampalli, F. Yang, and A. Doan. Chimera: Large-scale classification using machine learning, rules, and crowdsourcing. *Proceedings of the VLDB Endowment*, 7(13):1529–1540, 2014.

[53] Toloka. Toloka: Data Labeling Platform. `https://toloka.ai/`, 2024. Accessed: 2025.

[54] N. Vesdapunt, K. Bellare, and N. Dalvi. Crowdsourcing algorithms for entity resolution. *Proceedings of the VLDB Endowment*, 7(12):1071–1082, 2014.

[55] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. CrowdER: Crowdsourcing entity resolution. *Proceedings of the VLDB Endowment*, 5(11):1483–1494, 2012.

[56] Y. Zheng, G. Li, Y. Li, C. Shan, and R. Cheng. Truth inference in crowdsourcing: Is the problem solved? *Proceedings of the VLDB Endowment*, 10(5):541–552, 2017.

[57] Y. Zheng, J. Wang, G. Li, R. Cheng, and J. Feng. QASCA: A quality-aware task assignment system for crowdsourcing applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1031–1046, 2015.