Welding Natural Language Queries to Analytics IRs with LLMs

Kaushik Rajan
Microsoft Research
Redmond, USA

Aseem Rastogi
Microsoft Research
Bangalore, India

Akash Lal
Microsoft Research
Bangalore, India

Sampath Rajendra
Microsoft Research
Bangalore, India

Krithika Subramanian
Microsoft Azure Data
Bangalore, India

Krut Patel*
University of Illinois
Urbana-Champaign
Urbana, IL, USA

ABSTRACT

From the recent momentum behind translating natural language to SQL (NL2SQL), to commercial product offerings such as Co-Pilot for Microsoft Fabric, Large Language Models (LLMs) are poised to have a big impact on data analytics. In this paper, we show that LLMs can be used to convert natural language analytics queries directly to custom intermediate query representations (IRs) of modern data analytics systems. This has the direct benefit of making IRs more accessible to end-users, but interestingly, it can also result in improved translation accuracy and better end-to-end performance, especially when the query semantics is better captured in the IR rather than in SQL. We build an LLM-based pipeline (NL2WELD) for one instance of this flow, to translate natural language queries to the Weld IR using GPT-4. NL2WELD is carefully designed to harness self-reflection and instruction-following capabilities of GPT-4, providing it various forms of feedback such as domain specific instructions and feedback from the Weld compiler. We evaluate NL2WELD on a subset of the Spider benchmark and compare it against the gold standard SQL and *DIN-SQL*, a state-of-the-art NL2SQL system. We report a comparable accuracy of 77.4% on the dataset, and also demonstrate examples on which NL2WELD produces code that is $1.2-3\times$ faster than the gold standard and *DIN-SQL*.

1 INTRODUCTION

Pre-trained Large Language Models (LLMs) have shown impressive emergent capabilities of generating code from natural language [4]. Such capabilities are poised to disrupt the software engineering industry [5], by making it practical to have natural language-based frontends accessible even to non-experts. Data analytics systems are also picking up on the trend. The task of translating natural language queries to SQL (NL2SQL) has gained traction, with several high-accuracy translation methods [6, 22] as well as commercial product announcements, such as Co-Pilot for Microsoft Fabric [17].

In addition to leveraging LLM capabilities, NL2SQL also has the advantage of keeping the SQL query close to its natural language intent. However, is SQL always the right target to capture this intent? One can attempt similar variants that instead target other popular data analytics frameworks [11] from the *PyData stack* such as numPy and Pandas. However, instead of stopping there, this paper takes the argument one step further and argues for building translations that directly target the *intermediate representations* (IRs) of modern database systems.

Several recent analytics platforms [7–10, 12, 14, 18–21] have designed and advocated for domain-specific IRs. These IRs are supported by a compilation pipeline to generate efficient executable code. The reach of these IRs has, however, been limited because they remain internal to the platforms, with no expectation of a user directly programming in the IR. This limitation also imposes an implicit burden on the IR designer—they must develop a complete frontend for translating SQL (or other user-facing interfaces) to the IR. We argue that these IRs, which are programming languages in their own right, form a rich playground for database research, specifically for directly capturing a user's natural-language-based intent in the IR—a task that we call NL2IR.

This paper presents NL2WELD, an instance of NL2IR where we use the state-of-the-art LLM GPT-4 to translate natural language to Weld [19, 20], a simple and concise analytics IR based on linear data-structures. Weld has syntax similar to that of a functional language, much like Rust. We demonstrate examples of data-analytics queries where the intent is better captured in Weld than in SQL. The fact that each such IR will be better than SQL at capturing user intent for some set of queries should not be surprising. Our work should be taken as evidence that NL2IR is a useful generalization of NL2SQL that will benefit users of data analytics in the long term.

This paper answers key technical challenges required for realizing NL2IR. For instance, the popularity of SQL implies that pre-trained models would already be proficient with its syntax. This is certainly not true for Weld, and would be the case for any new IR. One option would be to invest in fine-tuning of models with lots of code in the IR. This would be expensive and time-consuming, and would be a big hurdle to realize NL2IR. We demonstrate that by leveraging the *instruction-following* capabilities of LLMs, fine-tuning is not required. We describe all the features of Weld in a single prompt and instruct the LLM to follow the rules of the syntax. We show that GPT-4 is good at following such instructions.

Even with these instructions, we find that the zero-shot performance of GPT-4 for NL2IR significantly lags NL2SQL. We demonstrate the design of a prompting-based pipeline that closes this gap. In particular, NL2WELD uses self-reflection capabilities of GPT-4 to let it fix logical errors in the generated code. We use GPT-4 to also fix syntax and typing errors in its code, by feeding it compilation errors as reported by the Weld compiler. Lastly, to increase the chances

---

[*] Work done while working at Microsoft Research.

This paper is published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution, provided that you attribute the original work to the authors and CIDR 2024. 14th Annual Conference on Innovative Data Systems Research (CIDR '24). TODO, Chaminade, USA

of producing valid Weld code, we ask GPT-4 to generate multiple programs per query and apply the above pipeline to each of them.

We evaluate NL2WELD on four evaluation (dev) databases from Spider, a standard NL2SQL benchmark suite [3]. Results indicate that NL2WELD has comparable accuracy to state-of-the-art NL2SQL systems. NL2WELD achieves an accuracy of 58.6% with one completion, and when configured to return multiple completions it attains an accuracy of 77.4%. On these databases *DIN-SQL*, the leading NL2SQL system on Spider, achieves an accuracy of 71.6% with one completion (DIN-SQL does not use multiple completions). These numbers indicate that while the models are more proficient at generating SQL by default, using multiple completions, self-reflection, and complier feedback can close this gap. Further, we find that our system sometimes produces code that runs faster than $(1.2 - 3\times)$ a Weld translation of the query plan for gold standard SQL queries in the benchmark suite[1].

In summary, this paper makes the following contributions.

1. We formulate the NL2IR problem that aims to directly capture natural-language intent in the sea of data-analytics IRs.
2. We build and evaluate NL2WELD, a novel system for NL2IR based on the Weld language. The system crucially relies on LLM capabilities to follow instructions (to encode Weld syntax), as well as self-reflect (on logical errors) and self-fix compilation errors to improve overall accuracy.
3. We demonstrate that NL2WELD can achieve a comparable translation accuracy to NL2SQL, solving more problems in the hardest category, and can produce more efficient programs.

2 MOTIVATING EXAMPLE

This section present an example query from the Spider benchmark and contrasts its SQL representation against a Weld representation.

Weld IR. Weld [19, 20] is a functional language designed for high performance data analytics. In addition to basic data types and closures (written as |bs| e, where bs are the binders and e is the closure body), Weld supports immutable structs and immutable collections such as vectors and dictionaries. The key construct of Weld is an abstraction called *builders* that provide write-only, build-once abstractions for creating collections. Builders support three operations: merge(b,v) for merging a new value into a builder, result(b) for converting a builder into an immutable collection, and for(v,b,f), a looping combinator that folds a builder-returning function f over the vector v, and returns the final builder (b is the initial builder that the loop starts with).

For example, dictmerger is a builder type for dictionaries. It is initialized with the constructor dictmerger(k,v,id,f), where k and v are the key and the value type respectively, id is the identity for v, and f is a binary function to combine the previous value of a key (if it exists) with the new value of the key, when the dictmerger is updated with merge. Invoking result(b) on a dictmerger converts it to a dictionary. An optlookup(d, k) returns a struct value {b, v}, where b is a boolean indicating whether k exists in the dictionary d, and v is the value of k in d when b is true. Weld imposes a linear type system [24] to ensure that the builders are used *exactly once*. This allows the compiler to soundly generate low-level code that mutates the builders in-place and makes result essentially a no-op.

We have added a few basic features to the original Weld language [19, 20] that are relevant for NL2IR. For example, the original Weld supported only fixed set of binary operations (addition, maximum, etc.) as the binary function in dictmerger. We extended it to support custom closures, written as |x, y| e where x and y are the closure parameters and e is the closure body. This increases the expressiveness of the language and allows efficient query implementations. We have also improved the Weld parser and the typechecker to implement missing checks and report better error messages; this enables us to use GPT-4 for fixing syntax and typing errors.

Example query. One of the queries in the Spider benchmark [3], *"find all students that have both dogs and cats as pets"*, operates over three tables Pets, Students, and Has_pet, where the last table contains pairs of students and pet ids. Figure 1 shows the gold standard SQL representation of the query from the dataset. The query computes two different sets of students, those with cats and those with dogs (each with a 3-way Join), and then interesects the results. The optimized plan also perform two 3-way joins, scanning each table twice.

Figure 2 shows the Weld IR for the same query as generated by our NL2IR pipeline. The code, as presented, compiles with the Weld compiler and passes the testcases from the dataset. It also crucially relies on our Weld enhancement for custom closure support in dictmerger. The program takes the 3 tables as inputs, each represented as a vector of structs, and iterates over each of them exactly once. It first iterates over the pets table and creates a dictionary that maps pet id to pet type (lines 2-4). The dictionary is created using a dictmerger, populated inside a for loop, finally converted to a dictionary using result.

The program then iterates over the has_pet table and creates a second dictionary that maps student id to a struct of two booleans, indicating whether the student has at least one cat and one dog, respectively, (lines 6-10). The binary function in the dictmerger constructor takes boolean or (||) of the corresponding struct fields. This step uses the dictionary from the previous step to lookup the pet type. Finally, the program iterates over the student table and creates a vector of their first names when both the fields in the corresponding value in the dictionary from the previous step are true.

This Weld code avoids performing the same set of joins twice. Writing similar code in SQL would require using advance features like user-defined-aggregation functions or complex case-when expressions that most NL2SQL systems struggle with today [13, 22]. Such rewrites are also beyond the scope of existing query optimizers [15]. As we show in Section 4, the Weld IR is roughly 1.5x faster than the gold standard SQL for this query.

3 NL2WELD PIPELINE

Figure 3 illustrates the architecture of NL2WELD, our multi-agent LLM-based pipeline for solving the NL2IR problem for the Weld language. Each of the green shaded boxes are calls to an LLM, which is always GPT-4 in our implementation. We use a conversational interface to GPT-4, where each conversation consists of a single

---

[1] We enhance the sample databases in the benchmark suite with a data generator that preserves the column distributions and key constraints. More on this in Section 4.

```
SELECT T1.Fname FROM student AS T1 JOIN has_pet AS T2 ON T1.stuid=T2.stuid JOIN pets AS T3 ON T3.petid=T2.petid WHERE T3.pettype='cat'
INTERSECT
SELECT T1.Fname FROM student AS T1 JOIN has_pet AS T2 ON T1.stuid=T2.stuid JOIN pets AS T3 ON T3.petid=T2.petid WHERE T3.pettype='dog'
```

Figure 1: SQL gold standard for the query "find all students that have both cats and dogs as pets"

```
|student:vec[{student_id:int, first_name:str,...}], has_pet:vec[{student_id:int, pet_id:int}], pets:vec[{pet_id:int, pet_type:str,...}]|
(* Step 1: Create a dictionary for pets with pet_id as the key and pet_type as the value *)
let pets_dict = for (pets, dictmerger[int, str, |agg, curr| curr,""], |b, _, e| merge(b, {e.pet_id, e.pet_type}));
let pets_dict_res = result pets_dict;
(* Step 2: Iterate over has_pet and find students who have both cats and dogs *)
let students_with_both_pets = for (has_pet, dictmerger[int, {bool, bool}, |agg, curr| {agg.$0 || curr.$0, agg.$1 || curr.$1}, {false, false}], |b, _, e|
  let pet_type = optlookup (pets_dict_res, e.pet_id);
  if (pet_type.$0, merge (b, {e.student_id, {pet_type.$1 == "cat", pet_type.$1 == "dog"}}), b)
);
let students_with_both_pets_res = result students_with_both_pets;
(* Step 3: Iterate over the students and output the first names of students who have both cats and dogs *)
let final_output = for (student, appender[str], |b, _, e|
  let student_pets = optlookup (students_with_both_pets_res,e.student_id);
  if (student_pets.$0 && student_pets.$1.$0 && student_pets.$1.$1, merge (b, e.$1), b)
);
result final_output
```

Figure 2: NL2IR-generated Weld IR for the query "find all students that have both cats and dogs as pets"

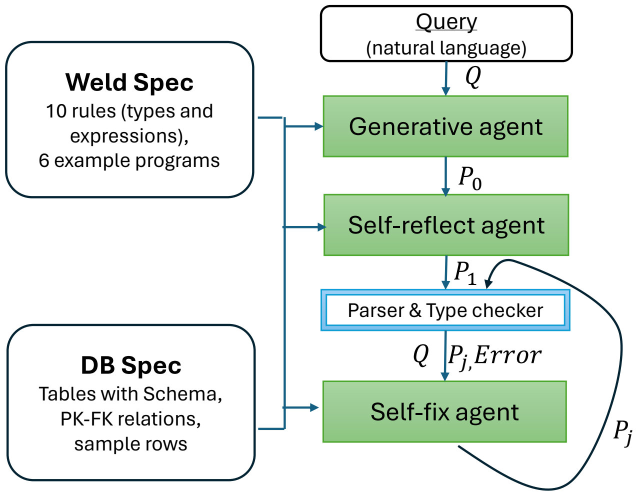

Figure 3: A multi-agent LLM-based NL2IR pipeline.

system prompt that is supplied initially to constrain the entire conversation, followed by a sequence of one or more *user prompts* that each ask for a response.

The specification of the Weld language is supplied as part of the system prompt to inform the model about the features of the language (details later in this section). The system prompt further also includes details about the schemas of all the tables in the database, with the names of any primary-key columns, primary-key/foreign-key (PK-FK) relations between the tables. We also optionally provide a table size hint (small, medium, large) and one or two sample input rows per table. Figure 18 in Appendex shows the complete specification for one of the databases used for evaluation.

Once the system prompt is initialized, NL2WELD uses three different agents that each supply user prompts as part of the same conversation. The first agent (*generative agent*) takes the input natural language query Q and simply asks for a Weld implementation of the query (P_0). The second agent (*self-reflection agent*) takes inspiration from prior work [22] to perform self-reflection, where we ask GPT-4 to check if P_0 incorrectly implements Q? (We find that asking GPT-4 to identify if the program is incorrect is more useful than asking if the program is correct.) Figure 4 shows the complete prompt and an example of self reflection. As can be seen GPT-4 responds with a fixed program and adds detailed comments to the program P_1. These comments can help a user understand the Weld code. (This is why Figure 2 has comments.)

Finally, in the last stage of the NL2WELD (*self-fix agent*), we run the Weld compiler to check for parsing and typing errors. If any errors are reported, the error message is supplied to the model, along with the erroneous program and we ask it to fix the particular error. Figure 5 shows an example of self-fixing. This process is repeated until we get a program that passes the compiler; the number of retries is bounded to 3, after which NL2WELD reports a failure. Figures 19— 21 in the Appendix show a complete end-to-end example that includes all the prompts and responses received.

GPT-4 Prompt:

Given a query and a Weld program that is supposed to implement the query, check if the program is logically incorrect or if it does not follow the rules of the language. Produce the correct program and add detailed comments justifying its correctness.
Query: ...
Program:

```
|airlines:vec[{airline_id:i32, airline_name:str, ...}]|
let count_airlines = COUNT_i32 (airlines.airline_id);
count_airlines
```

GPT-4 Response:

The original program does not follow the rules of Weld as it directly uses COUNT_i32 on airlines.airline_id without iterating over the input vector. The corrected program below uses a for loop to iterate over the input vector and appends the airline_id to an appender collection, calls result and uses COUNT_i32 to count the number of airline_id.

```
|airlines:vec[{airline_id:i32, airline_name:str, ...}]|
let count_airlines = for (airlines, appender[i32],
                          |b, _, e| merge (b, e.airline_id));
let total_airlines = COUNT_i32 (result count_airlines);
total_airlines
```

Figure 4: Example of self-reflection: COUNT_i32 is a Weld macro that expects a vector of type vec[i32] as argument.

GPT-4 Prompt:

Given Weld program and a compiler error in the program, produce the fixed Weld program. Program:

```
for (student, dictmerger[i32, str, |agg, curr| curr], |dm, i, st| ...)
```

Compiler error: Expected ',' before lambda identity value, got ']'. Perhaps you are using reduction functions incorrectly.

GPT-4 Response:

```
for (student, dictmerger[i32, str, |agg, curr| curr, ""], |dm, i, st| ...)
```

Figure 5: Example of self-fixing: the dictmerger declaration expects as additional argument (the identity value).

This document describes my own programming language L with a set of rules. L supports a few scalar and a few collection types. The types and operations can be performed on them are described below.
S1. Basic variable b:bt, has a standard basic type bt ::= bool | i8 | i64 | f64 | str. On these types L supports basic expressions, type casts, for example, i32(b), if(cond,then expr, else expr). Note that variables passed to expressions should have compatible types.
S2. Struct variable s:st, has a struct type st::=bt | (st,bt). The fields of the structure can be named or can be accessed via positional identifiers starting with $0.

Figure 6: Specifying common Weld features in a prompt

Weld-in-a-prompt. The crucial aspect of NL2IR is to ensure that the LLM understands the IR. One cannot expect a pre-trained model to be updated as frequently as an IR is updated. We found that while GPT-4 had some understanding of Weld, it certainly did not know enough details to output correct Weld programs. We expect that our methodology can be used for any IR that is concise, well-defined, and has a small number of key combinators. Weld satisfies these requirements.

The Weld "encoding" consists of four main components: (*a*) description of features that Weld has in common with other functional languages (basic data types, let, conditionals, structs), (*b*) description of Weld-specific abstractions (builders and loops), (*c*) a few small example Weld programs along with a natural language description of the task that they accomplish, and (*d*) instructions for how to encode commonly occurring patterns in data analytics into Weld combinators. We look at each of these components next.

Common features. These instructions are straightforward and specify the basic data types, conditionals, structs etc. Figure 6 shows a snippet of our GPT-4 prompt for the same, where we abstractly refer to Weld as the language *L* so that the model does not pick up its implicit bias for Weld.

A dictmerger collection dr:dictmerger(key:st,value:st,redFn) has a dictionary type, that represents key value pairs where keys and values can themselves be basic types or structs of basic types. In addition it takes a reduction function redFn, of the form |agg:st,curr:st|→(e). Where agg is the current value in the dictmerger and curr is a new value to be folded into agg. e is an expression that evaluates the new agg that has been updated with curr. The identity struct contains the initial values for each element in the agg struct. dr supports the following operations. The full types of dictmerger variables and the complete redFn should be explicitly defined.

(a) dr supports an optlookup function. (flag:bool,v:st) = optlookup(dr,k:st) produces flag:=True only if the key k exists in the dictionary, and when flag is true, v contains the value in the dictionary.
(b) It supports a merge function merge(dr,k:st,v:st). The merge function dr.merge(k,v) checks if the key k already exists and if so updates the current value v with the new value by applying comOp. Note that the keys of a dictionary are to be unique.
(c) dr can be converted into a vector with let v =tovec(dr).
(d) dr supports a result function result(dr). It is required that result() be called before any calls to optlookup ortovec and that merge is not called after result.

Figure 7: Specifying dictmerger in a prompt

Weld-specific abstractions. We follow the methodology of specifying novel language abstractions using concepts that would already be familiar to the model. For instance, we describe builders as collections, with specialized syntax for their construction and use.

Figure 7 shows the prompt snippet for specifying dictmerger. The complete prompt can be found in Figure 14 in the Appendix.

Small examples. This section of the prompt consists of natural language description of small data analytics queries, and their corresponding Weld programs. All examples used in the prompt can be found in Figures 15 — 17 in the Appendix.

Domain-specific instructions. This is the final section of the prompt where we encode domain-specific instructions for writing Weld programs for SQL-like data analytics queries. These instructions encode common optimizations such as filter pushdown, etc. Figure 8 shows a partial snippet from our GPT-4 prompt.

For efficiency reasons you should:
(1) make use of dictmerger to aggregate values per key, or maintain a mapping from a primary key to values. Make sure you use an appropriate reduction function.
(2) combine multiple vectors by inserting the smaller table into a groupmerger or a dictmerger and look it up while iterating over the larger one. A dictmerger should be preferred if the key is the primary key of a table.
(3) apply filters as early as possible, preferably during the first pass on an input vector.
(4) create collections with as few columns as possible. Only columns that are used subsequently should be in the collection.
(5) Iterate over input tables as few times as possible. In general programs with fewer loops and no nesting are preferred.

Figure 8: Specifying domain-specific insights in a prompt

4 EVALUATION

We evaluate NL2WELD on the Spider dataset [3] that has been widely used for NL2SQL benchmarking. It consists of 20 (dev) databases, among which we pick 4: $pets\_1$, $flight\_2$, $singer$, and $network\_1$ for our experiments because our access to GPT-4 was limited at the time of writing this paper. These are some of the more challenging databases for NL2SQL.They together have a total of 208 queries. We consider both translation accuracy as well as query performance in our evaluation.

Accuracy captures how often the generated Weld matches the golden SQL query. We use *execution accuracy* as the metric for accuracy, i.e., we run the queries on the provided databases and check equivalence of the results; the *exact match* metric of Spider is not relevant because the benchmark suite does not have golden Weld queries. Further, we report *pass@k* accuracy, which refers to running NL2WELD for a total of k times and checking if any of the Weld output matches the golden SQL on execution accuracy. We compare our results with DIN-SQL [22], the NL2SQL system with the current highest official execution accuracy on Spider. DIN-SQL also uses GPT-4.

Performance. The goal of the performance evaluation is to determine if NL2WELD generated IRs are more efficient than what can be achieved by a direct translation of the SQL query to the IR. To perform this comparison we take the gold standard SQL query, generate a query plan for it using a state-of-the-art query optimizer, and translate the optimized plan into Weld IR[2]. To generate the plans we use duckDB [23], a popular open-source query engine widely used for performance comparisons [8, 12]. For completeness, we compare not just with the (IR corresponding to) gold standard query in the benchmark, but also *DIN-SQL* generated SQL query.

The datasets used for correctness evaluation in the benchmark are too small (10s of rows) to be used for performance evaluation. We, therefore, make use of a synthetic data-generator [2] that given a schema, constraints, distribution of values and size information, can generate synthetic data that preserves the constraints and distributions. We evaluate performance on the $pets\_1$ database[3]. We use latency of a single-threaded execution as the metric for performance. We extended the Weld runtime to execute the IRs as generated with inputs being read via an in-memory table reader [1].

4.1 Accuracy

As reported in Figure 9, we achieve a pass@1 accuracy of 58.6% and a peak accuracy of 77.4%. Accuracy remains at this level for k in the range $3-8$. In comparison, *DIN-SQL* has a pass@1 accuracy of 71.6%. They do not generate multiple completions.

Figure 10 provides a breakdown of accuracy for each database and hardness category, based on ease of expressing the computation in SQL. The comparison of results per hardness category against DIN-SQL is interesting. While both solve the same set of *easy* queries, they differ in their ability to solve other categories. This is because of the difference in expressiveness between SQL and Weld. For example, the benchmark categorizes SQL queries that use nesting and make use of set operators (in, $intersect$, $exists$) as *extra* hard. Weld does not have these notions, it directly encodes the query using dictionaries and vectors, the data-structures used internally by databases to implement physical operators. This demonstrates that some tasks that are hard for NL2SQL could, in fact, be easier for NL2IR. We find that NL2WELD solves 36 queries (7 extra hard) that *DIN-SQL* fails on, while *DIN-SQL* solves 24 (4 extra-hard) queries that NL2WELD fails on. The accuracy reaches 88.9% if we combine *DIN-SQL* together with NL2WELD!

Ablations. Figure 11 shows the impact on accuracy of removing some components from the system. As can be seen, removing *self-fix* has a significant impact, halving the accuracy on two databases and reducing the overall accuracy by 28%. Our prompts have limited program examples, so the model tends to make syntactic mistakes. Feedback from the compiler is critical for the model to fix such errors. Removing *self-reflect* (Figure 4) or *domain specific instructions* (Figure 8), has a lower but still non-negligible impact.

4.2 Cost of using the LLM pipeline

NL2WELD requires multiple rounds of interaction with GPT-4 per query: an initial generation, self-reflect and then up to three self-fix passes. We find that, on average, one completion requires about 8700 input tokens (worst case 17850) and 1460 (worst case 4550) output tokens. At current pricing of OpenAI endpoints[4], this translates to

---

[2] We do this translation while adhering to the same *domain-specific guidelines* that we provide to the LLM.

[3] With table sizes $2M$, $10M$, $5M$ rows for $pets$, $students$, has_pet, and column value distributions being either from fixed set (low cardinality columns) or *Gaussian.*

[4] See pricing for the model `gpt-4` here: https://openai.com/pricing

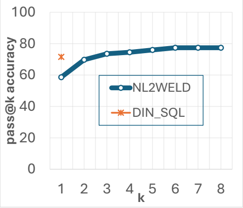

Figure 9: Pass@k

	NL2WELD					DIN-SQL(%)
	Pets	network	singer	flights	all(%)	
Easy	4/4	11/12	6/6	22/26	89.5	89.5
Med	17/22	20/22	18/18	25/30	86.9	69.5
Hard	5/6	8/16	1/6	5/8	52.7	72.2
Extra	8/10	3/6	0/0	8/16	59.4	50.0
per DB	34/42	42/56	25/30	60/80	77.4	71.6
DIN SQL(%)	33/42	38/56	21/30	57/80	71.6	-

Figure 10: Accuracy breakdown

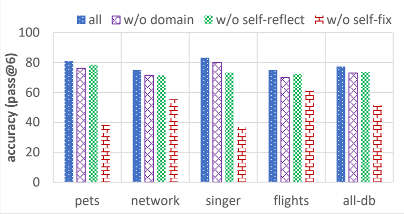

Figure 11: Ablations

an average cost of $0.37 (USD) (worst case $0.80) per completion. The cost scales linearly with the number of completions requested.

OpenAI API allows for the generation of multiple completions in a single GPT-4 request, however our client did not use this setting as the token count would exceed the 4K token limit that the model had at the time of evaluation. GPT-4 also supports 32K token limit now, and that is enough to generate all the completions in one pass of the pipeline. Using this option should bring down the cost per completion by at least half for getting three completions.

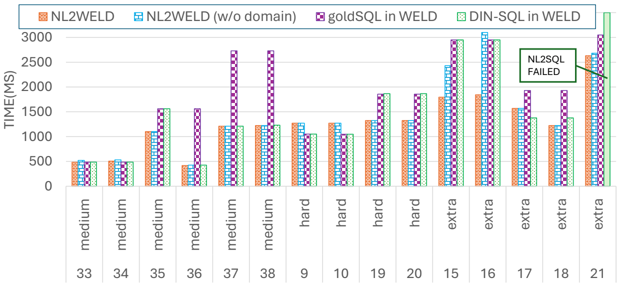

Figure 12: Performance comparison of NL2WELD vs. SQL. The plot shows the performance of NL2WELD generated IRs with and without domain instructions, against the gold standard SQL query translated to Weld, as well the DIN-SQL generated query translated to Weld. The plot only shows queries where there is a performance difference. For other queries the performance is identical and hence not reported.

4.3 Performance evaluation

Figure 12 reports the performance of running the IR corresponding to the gold standard SQL query and the NL2WELD generated IR (when accurate), for *pets*_1. The plot also reports the performance of the *DIN-DQL* generated query, and a variant of NL2WELD where we drop *domain specific instructions* (Figure 8) from the prompt. For about half the queries, all the different IRs are identical. We only report numbers when there is some observable performance difference.

As can be seen, NL2WELD often generates programs (on queries $q15 - q21, q35 - q38$) that are more efficient (at least 1.25× faster) than gold SQL. Figure 13 reports a qualitative comparison of the generated IRs. NL2WELD is often able to produce programs with fewer dictionaries (see Section 2 for an example). In two cases ($q9, q10$) NL2WELD has poorer performance because it picks a less efficient join order.

Interestingly, we find that DIN-SQL generated queries can match the improved performance of NL2WELD for 5 queries ($q17, q18, q36 - q38$), highlighting the usefulness of using LLMs to generate alternate SQL representations. NL2WELD still produces better programs than both gold SQL and DIN-SQL for 6 queries, demonstrating that, even from a performance point of view, there is value in considering IR as a target for LLMs. Finally, as one would expect, *domain specific instructions* are important not just for accuracy but also for performance; dropping them degrades performance on 5 queries.

We also compared SQL queries running on duckDB with their Weld translations and found that for long running queries ($> 100ms$), the Weld version was faster by $1.2 - 3\times$. Several other IR based systems [8, 10, 12] have reported significant improvements over duckDB as well.

Note that the objective of this paper is not to directly compare Weld with duckDB or with other IR based systems, but to only show that NL2IR systems are useful to study as they can achieve better translation accuracy and performance on some queries.

5 DISCUSSION

With the emergence of LLMs, there is an exciting opportunity to build natural language based data analytics systems that can be used even by non-experts.

In this paper, we have shown that these systems can translate natural language queries directly to custom IRs. We build and evaluate an instance of such an NL2IR pipeline that uses GPT-4 to translate natural language intent into Weld IR. We show that this pipeline can solve problems that are challenging for NL2SQL and someteims achieve better performance than NL2SQL. In doing so, our tool brings the community one step closer to democratizing data analytics via natural language.

Future research directions. Our work is a first step towards using LLMs to translate natural language queries to custom IRs. We have some initial evidence that this can be done with reasonable accuracy and efficiency. However, several challenges remain.

First, we have implemented and evaluated only one instance of NL2IR. It remains to be seen whether our methodology extends to other IRs [8, 12, 16, 21] that are concise and well-defined like Weld.

Second, how do we get accuracy of these systems closer to 100%? We show that by combining NL2IR with NL2SQL, one could get closer to this target on a few benchmarks. However, a more thorough, real world, evaluation is needed. Natural language is inherently ambiguous, and there is no easy way to know whether a formally specified query, be it in SQL or some other IR, captures the natural language intent. We envision that a more comprehensive system that lets

Query	Sample SQL Query	How NL2WELD generated IR is different

q9,q10, q33, q34	SELECT count(T1.PetID) FROM Pets AS T1 JOIN Has_Pet AS T2
	ON T1.PetID = T2.PetID JOIN Student AS T3 ON T2.StuID = T3.StuID
	WHERE PetType = 'dog' AND Sex = 'F'
		Generated IR uses a less efficient join order or join algorithm

q15-q21	SELECT StuID FROM Student WHERE StuID NOT IN (SELECT T1.StuID
	FROM Student AS T1 JOIN Has_Pet AS T2 ON T1.StuID = T2.StuID
	JOIN Pets AS T3 ON T2.PetID = T3.PetID WHERE T3.PetType = 'cat')
		Generated IR avoids set operation induced joins (*except, not-in, intersect*), uses one or two dictionaries with custom aggregation. (Figure 2 is q16)

q17-q20, q35,q36	SELECT count(T1.PetID), T1.StuID FROM HAS_Pet AS T1 JOIN Student AS T2 ON T1.StuID = T2.StuID GROUP BY T1.StuID	Generated IR avoids a PK-FK join. For q17,q18 and q36, DIN-SQL also generates a query without the join.
q37, q38	SELECT Fname, Sex FROM Student WHERE StuID IN (SELECT StuID FROM HAS_Pet GROUP BY StuID HAVING count(PetID) > 1)	Generated IR performs aggregation before joins. DIN-SQL is able to match this with an alternative SQL representation.

Figure 13: Summary of differences between NL2WELD generated IR and IR corresponding to the gold standard query

users specify and refine intent in an incremental and interactive way would be needed.

Finally, going from natural language to IRs instead of SQL skips over the SQL query optimizer, and we did find cases of performance degradation because of this in our experiments. Recent research shows how many query plan rewrites can be equivalently expressed as IR-to-IR rewrites in a compiler [10]. This is another interesting future research direction.

REFERENCES

[1] Apache arrow. https://arrow.apache.org/.
[2] Datafiller. https://www.cri.ensmp.fr/people/coelho/datafiller.html.
[3] Spider 1.0. https://yale-lily.github.io/spider.
[4] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv:2303.12712*, 2023.
[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv:2107.03374*, 2021.
[6] Han Fu, Chang Liu, Bin Wu, Feifei Li, Jian Tan, and Jianling Sun. Catsql: Towards real world natural language to sql applications. *Proc. VLDB Endow.*, 16(6):1534–1547, apr 2023.
[7] Ferdinand Gruber, Maximilian Bandle, Alexis Engelke, Thomas Neumann, and Jana Giceva. Bringing compiling databases to risc architectures. In *Proceedings of the VLDB Endowment*, volume 16. VLDB Endowment, 2023.
[8] Tim Gubner and Peter Boncz. Charting the design space of query execution using voila. *Proc. VLDB Endow.*, 14(6):1067–1079, feb 2021.
[9] Andreas Jungmair, Tobias Ziegler, and Alfons Kemper. Designing an open framework for query optimization and compilation. *Proceedings of the VLDB Endowment*, 15:2389–2400, 2022.
[10] Michael Jungmair and Jana Giceva. Declarative sub-operators for universal data processing. *Proc. VLDB Endow.*, 16(11):3461–3474, jul 2023.
[11] Kaggle. State of data science and machine learning 2022. https://www.kaggle.com/kaggle-survey-2022, 2022.
[12] Timo Kersten, Viktor Leis, and Thomas Neumann. Tidy tuples and flying start: fast compilation and fast execution of relational queries in umbra. *The VLDB Journal*, 30:883–905, 2021.
[13] Hyeonji Kim, Byeong-Hoon So, Wook-Shin Han, and Hongrae Lee. Natural language to sql: Where are we today? *Proceedings of the VLDB Endowment*, 13(10):1737–1750, 2020.
[14] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building efficient query engines in a high-level language. In *Proceedings of the VLDB Endowment*, volume 7, pages 853–864. VLDB Endowment, 2014.
[15] Jyoti Leeka and Kaushik Rajan. Incorporating super-operators in big-data query optimizers. *Proc. VLDB Endow.*, 13(3):348–361, nov 2019.
[16] Yinan Li, Jiajie Luo, and Kenny Q. Zhu. Horseir: A uniform vector representation for relational and array languages. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*, pages 1643–1658. ACM, 2018.
[17] Kim Manis. Introducing microsoft fabric and copilot in microsoft power bi. https://powerbi.microsoft.com/en-us/blog/introducing-microsoft-fabric-and-copilot-in-microsoft-power-bi/.
[18] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. In *Proceedings of the VLDB Endowment*, volume 4, pages 539–550, 2011.
[19] Shoumik Palkar, James J. Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimarjan Negi, Malte Schwarzkopf, Holger Pirk, Saman P. Amarasinghe, and Matei Zaharia. Weld: Rethinking the interface between data-intensive libraries. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*, pages 1573–1588. ACM, 2018.
[20] Shoumik Palkar, James J. Thomas, Anil Shanbhag, Deepak Narayanan, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, and Matei Zaharia. Weld: A common runtime for high performance data analytics. In *8th Biennial Conference on Innovative Data Systems Research (CIDR '17)*, 2017.
[21] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. Voodoo - a vector algebra for portable database performance on modern hardware. In *Proceedings of the VLDB Endowment*, volume 9, pages 1707–1718, 2016.
[22] Mohammadreza Pourreza and Davood Rafiei. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *arXiv:2304.11015*, 2023.
[23] Mark Raasveldt and Hannes Mühleisen. Duckdb: An embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1981–1984, New York, NY, USA, 2019. ACM.
[24] Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, 1990.

APPENDIX A : COMPLETE SYSTEMS PROMPT

As discussed in Section 3, the systems prompt has the language description and the database specification. The language description has four parts.

(1) The first part is a description of *common features* and this can be found in Figure 6 of Section 3.
(2) The complete description of Weld collections is shown in Figure 14.
(3) The *input-output examples* that are supplied with the prompt are listed in Figures 15 — 17.
(4) The last part is the *domain specific instructions* and these can be found in Figure 8 secrefPipeline.

Further the system prompt also contains a specification of the database. The database specification for the *pets*_1 databases is shown in Figure 18.

C1. vector collection v:vec st has a vector type, it represents a collection of elements of type st.

(a) v can be iterated over using a for loop. Let res = for (v,collection_type,|b,idx,row| (body)). Where v is a vector being iterated on. b binds to collection_type, is a new collection into which expressions on elements of v are folded. idx is a loop index, row is an element of the vector.

(b) Supports a merge function merge(v, val) that appends val to v.

(c) supports standard reduction operations SUM_[bt](v), MAX_[bt](v), MIN_[bt](v), COUNT(v) to reduce a vector of elements v of a base type $[bt]$ to a scalar value.

(d) v supports a result function. Let w=result(v). It is required that result is called before iterating over a vector, or applying other operations and that no calls to merge are made after result.

Finally note that we have a type-alias for vec called appender which is the collection type that we construct in a for loop.

C2. A dictmerger collection dr:dictmerger(key:st,value:st,redFn) has a dictionary type, that represents key value pairs where keys and values can themselves be basic types or structs of basic types. In addition it takes a reduction function redFn, of the form |agg:st,curr:st|→(e). Where agg is the current value in the dictmerger and curr is a new value to be folded into agg. e is an expression that evaluates the new agg that has been updated with curr. The identity struct contains the initial values for each element in the agg struct. dr supports the following operations. The full types of dictmerger variables and the complete redFn should be explicitly defined.

(a) dr supports an optlookup function. (flag:bool,v:st) = optlookup(dr,k:st) produces flag:=True only if the key k exists in the dictionary, and when flag is true, v contains the value in the dictionary.

(b) It supports a merge function merge(dr,k:st,v:st). The merge function dr.merge(k,v) checks if the key k already exists and if so updates the current value v with the new value by applying comOp. Note that the keys of a dictionary are to be unique.

(c) dr can be converted into a vector with let v =tovec(dr).

(d) dr supports a result function result(dr). It is required that result() be called before any calls to optlookup ortovec and that merge is not called after result.

C3. A groupmerger collection gm:groupmerger(key:st,value: st), represents a dictionary with key k of some struct type and a value which is a vector of some struct type. A groupmerger supports the following operations. The full types of groupmerger variables should be explicitly defined.

(a) gm supports a merge function gm.merge(k:st,v:st). that appends v of type st to the current value vector.

(b) gm supports an optlookup function. output = optlookup(gm,k:st) produces a struct with output.$0:=True only if k exists in the dictionary and when true output.$1 contains the value vector in the dictionary. Note the value vector outputted by optlookup is of type vec.

(c) gm supports a result function.gm:=result(gm). It is required that $result()$ is called before any calls to optlookup and that no calls to append are called after result.

Figure 14: Specifying Weld-specific abstractions in a prompt

example 1 *Given a vector of integers as input, output a vector where each element is doubled.*

```
|x:vec[i32]| #input
let y = for (x,appender[i32],|b,i,e| #C1.a
merge(b,e+e); #C1.b, S1
);
let res=result(y); #C1.d
res
```

example 2 *Given x:vec[measure:f64,wt:i32]. Find all elements in the input vector where wt is greater than 3.*

```
|x:vec[{measure:f64,wt:i32}]| #input
let c = for (x,appender[{i32,f64}],|b,i,row|#C1.a
if(row.wt>3,#S2, S1 cond
    merge(b,{row.wt,row.measure}),#C1.b, S1 then part
    b#S1 else part
)
);
result(c)#C1.d
```

example 3 *Given x:vec[i32], output the sum of absolute values of the elements.*

```
|x:vec[i32]| #input
let c = for (x,appender[i32], |b,i,e|#C1.a
if(e>0,#S1 cond
    merge(b,e),#C1.b, S1 then part
    merge(b,−e)#S1 else part
)
)
let final = SUM_i32(result(c));#C1.c, C1.d
final
```

Figure 15: Input output examples for vectors

example 4 *Given Inp:vec[id:i64,m1:f64,m2:f64,wt:i32] write a program computes the sum of m1 and max of m2 per id for rows with wt greater than 9*

```
|Inp:vec[{id:i64,m1:f64,m2:f64,wt:i32}]|#input
# Iterate over the rows of Inp
let output = for(inp, dictmerger[i64,{f64,f64},
    |agg,curr|{curr.$0+agg.$0,if(curr.$1>agg.$1,curr.$1,agg.$1)},
    {0,f64(−Infinity)}], #S1,C1.a,C2
    |b,i,row|
        if (row.wt > 9,
          merge(b,{row.id, {row.m1, row.m2}}),#C2.b
        b)#S1
);
result(output)#C2.d
```

example 5 *Given parts:vec[id:i32,size:i32,name:str] with id as a primary key and store:[vecpart_id:i32,location:str] with a foriegn key part_id that references parts id. Find part_id, name, size, location and a flag indicating if the size is greater than 15 or not.*

```
|parts:vec[{id:i32,size:i32,name:str}],
store:[vec{part_id:i32,location:str}]| #input
#use dictmerger for parts as id is a primary key
#use custom lambda to remember the values for the key
let parts_dict = for(parts,dictmerger[i32,{sz:i32,nm:str,szGt15:bool},
|agg,curr|if(curr.sz>15,
     {curr.sz,curr.nm,true},
     {curr.sz,curr.nm,false}
     ),
     {i32(−Infinity),ID_str(),false}],|b,i,e| #S1,C1.a,C2
  merge(b,{e.id,{e.size,e.name,false}})#C2.b, S2
);
let parts_dict_res = result(parts_dict);#C2.d
let combined=for(store,
     appender[{id:i32,name:str,size:i32,location:str,szGt15:bool}],
     |app,i,st|#C1.a
          let from_parts=optlookup(parts_dict_res,st.$0);#C2.a
          if(from_parts.$0, #S1
               merge(app,{st.part_id,from_parts.$1.nm,
               from_parts.sz,st.location,from_parts.$1.szGt15}),#C1.b
               app
          )#S1
);
result(combined)#C1.c
```

Figure 16: Input output examples for dictmerger

example 6 *Given a small table X:vec[Okey:i64, commitDate:date, priority:str] and a large table Y:vec[Ykey:i64, shipmode:i8, receivedDate:date, price:f32, cost:f64]. Find the sum of revenue (defined as price - cost) for high priority orders per shipmode that were received before the commit date. From this determine the shipmode with the highest total revenue.*

```
|X:vec [{Okey:i64, commitDate:date, priority:str}],
Y:vec[{Ykey:i64, shipmode:i8, receivedDate:date, price:f32, cost:f64}]|
let high_pri_orders = for (X, groupmerger[i64, date],|gm,i,row|#S1,C3
     if (row.priority == "high" ,
          merge(gm, {row.Okey, row.commitDate}),#C3.a
          gm
     )#S1
);
let high_pri_orders_dict = result(high_pri_orders);#C3.d
let rev_shipmode = for(Y,dictmerger[i8,f64,|agg,curr|agg+curr,f64(0)],
   |dm,i,ys|#C1.a,C2
     let valueOut = optlookup(high_pri_orders_dict,ys.Ykey);#C3.b
     if(valueOut.$0,#S1
     for(valueOut.$1,dm,|b,i,x|#C1.a
          if (ys.receivedDate < x, #S1
                    merge(b,{ys.shipmode,ys.price − ys.cost}),#C2.b
                    b
          )#S1
        ),
        dm
     )
);
let rev_shipmode_vec = tovec(result(rev_shipmode));#C2.c
let max_rev_shipmode = for(rev_shipmode_vec,
     dictmerger[bool,{i8,f64},
     |agg,curr|if(curr.$1>agg.$1,curr,agg),{ID_i8(),f64(−Infinity)}],
     |dm,i,rev_shipmode|#C1.a,C2
          merge(rev_shipmode.$0,rev_shipmode.$1)#C2.b
     );
let final_res = optlookup(result(max_rev_shipmode),ID_bool())#C2.a
final_res
```

Figure 17: Input output examples for groupmerger

Input tables and schema

```
pets(pet_id:i32, pet_type:str, pet_age:i32, weight:f64)
: primary−key(pet_id)
has_pet(student_id:i32, pet_id:i32):
foreign−key(pet_id references pets(pet_id)),
foreign−key(student_id references student(student_id))
student(student_id:i32, last_name:str, first_name:str,
age:i32, sex:string, major:i32, advisor:i32, city_code:str)
: primary−key(student_id)
```

pets is a small sized table.
has_pet is medium sized table
student is a large sized table

Sample data

```
insert into Student values(101, 'Smith', 'Linda', 18, 'F', 60, 121,'BAL');
...
insert into Student values (102, 'Kim', 'Tracy', 19, 'F', 50, 712,'HKG');
...
insert into Has_Pet values ( 1001, 2001 );
...
insert into Has_Pet values ( 1002, 2003 );
...
insert into Pets values ( 2001, 'cat', 3, 12);
...
insert into Pets values ( 2003, 'dog', 1, 9.3);
...
```

Figure 18: Database specification for pets_1

APPENDIX B : END-TO-END EXAMPLE WITH NL2WELD

This section shows an end-to-end example of the different user prompts used by NL2WELD and the output of GPT-4 as well as the compiler.

1. Figure 19 shows the initial prompt that generates a program given a query text.
2. Figure 20 shows the self-reflect prompt and the response.
3. Figure 21 shows the response of the compiler, the *self-fix* prompt and the response from GPT-4.

The program output by the last step successfully compiles and ends the pipeline.

Generative agent request

Given a query text your task is to write multiple programs in L. A program is a single lambda that takes one or more input vectors and returns a single output vector. All programs are to be written with line numbers and end of line comments on what rules were applied. Please use input tables as defined in the database and use table names and column names as is for input vectors.You are going to be evaluated on the correctness of the programs you generate, and on whether you followed the rules of programming carefully or not. Lets work this out step-by-step to be sure we get the right answer.
Find number of pets owned by students who are older than 20.
please output a plain text program. All non program text should be prefixed with #

GPT-4 response

```
|students:vec[{student_id:i32, last_name:str, first_name:str,
        age:i32, sex:str, major:i32, advisor:i32, city_code:str}]
, |has_pet:vec[{student_id:i32, pet_id:i32}]| #input
let older_students = for (students, appender[{student_id:i32, age:i32}],
  |b, i, row| #C1.a
  if (row.age > 20, #S1 cond
    merge(b, {row.student_id, row.age}), #C1.b, S1 then part
    b #S1 else part
  )
);
let older_students_res = result(older_students); #C1.d

let older_students_dict = for (older_students_res,
    dictmerger[i32, i32, |agg, curr| curr, i32(−1)],
    |b, i, e| #C2
        merge(b, {e.student_id, e.age}) #C2.b
);
let older_students_dict_res = result(older_students_dict); #C2.d

let pet_count = for (has_pet, dictmerger[i32, i32,
    |agg, curr| agg + curr, i32(0)],
    |b, i, row| #C1.a, C2
        let is_older_student = optlookup(older_students_dict_res,
            row.student_id); #C2.a
        if (is_older_student.$0, #S1 cond
            merge(b, {row.student_id, i32(1)}), #C2.b, S1 then part
            b #S1 else part
        )
);
let pet_count_res = result(pet_count); #C2.d

let total_pet_count = SUM_i32(tovec(pet_count_res)); #C1.c
```

Figure 19: Generative agent and response

Self-reflect agent request

Given a query and a program that is supposed to implement the query, check if the program is logically incorrect or if it does not follow the rules of the language. Produce the correct program and add detailed comments justifying its correctness.
Query: *Find number of pets owned by students who are older than 20.*
Program:... (Output of generative agent)

GPT-4 response

```
# The original program follows the rules of L and is correct.
# Here's the program with end−of−line comments for each rule applied.

|students:vec[{student_id:i32, last_name:str, first_name:str,
        age:i32, sex:str, major:i32, advisor:i32, city_code:str}]
, |has_pet:vec[{student_id:i32, pet_id:i32}]| #input
let older_students = for (students, appender[{student_id:i32, age:i32}],
  |b, i, row| #C1.a
  if (row.age > 20, #S1 cond
    merge(b, {row.student_id, row.age}), #C1.b, S1 then part
    b #S1 else part
  )
);
let older_students_res = result(older_students); #C1.d


# Step 2: Create a dictionary of older students with student_id as the key
let older_students_dict = for (older_students_res,
    dictmerger[i32, i32, |agg, curr| curr, i32(−1)],
    |b, i, e| #C2
        merge(b, {e.student_id, e.age}) #C2.b
);
let older_students_dict_res = result(older_students_dict); #C2.d

# Step 3: Count the number of pets owned by older students
let pet_count = for (has_pet, dictmerger[i32, i32,
    |agg, curr| agg + curr, i32(0)],
    |b, i, row| #C1.a, C2
        let is_older_student = optlookup(older_students_dict_res,
            row.student_id); #C2.a
        if (is_older_student.$0, #S1 cond
            merge(b, {row.student_id, i32(1)}), #C2.b, S1 then part
            b #S1 else part
        )
);
let pet_count_res = result(pet_count); #C2.d

let total_pet_count = SUM_i32(tovec(pet_count_res)); #C1.c
```

Figure 20: Self reflect agent and response

Weld compiler output

```
InternalError Cannot cast (i32,i32) to i32
```

Self-fix agent request

Given Weld program and a compiler error in the program, produce the fixed Weld program. please only output plain text programs without verbose comments, ...add line numbers as per line comments
Program: ...(Output of self−reflect agent)
Compiler output : InternalError Cannot cast (i32,i32) to i32

GPT-4 response

```
#1
|students:vec[{i32, str, str, i32, str, i32, i32, str}],
        has_pet:vec[{i32, i32}]|#2

let older_students = for (students, appender[{i32, i32}], |b, i, row|#3
    if (row.$3 > 20,#4
        merge(b, {row.$0, row.$3}),#5
        b#6
    )#7
);#8
let older_students_res = result(older_students);#9

let older_students_dict = for (older_students_res,
    dictmerger[i32, i32, |agg, curr| curr, i32(−1)],
    |b, i, e|#10
    merge(b, {e.$0, e.$1})#11
);#12
let older_students_dict_res = result(older_students_dict);#13

let pet_count = for (has_pet, dictmerger[i32, i32,
    |agg, curr| agg + curr, i32(0)],
    |b, i, row|#14
    let is_older_student = optlookup(older_students_dict_res,
        row.$0);#15
    if (is_older_student.$0,#16
        merge(b, {row.$0, i32(1)}),#17
        b#18
    )#19
);#20
let pet_count_res = result(pet_count);#21

let pet_count_vec = tovec(pet_count_res);#22
let pet_counts_only = for (pet_count_vec, appender[i32], |b, i, e|#23
    merge(b, e.$1)#24
);#25
let total_pet_count = SUM_i32(result(pet_counts_only));#26
total_pet_count#27
```

Figure 21: Compiler output, self-fix agent and response