Self-Enhancing Video Data Management System for Compositional Events with Large Language Models

ENHAO ZHANG, University of Washington, USA
NICOLE SULLIVAN, University of Washington, USA
BRANDON HAYNES, Microsoft Gray Systems Lab, USA
RANJAY KRISHNA, University of Washington, USA
MAGDALENA BALAZINSKA, University of Washington, USA

Complex video queries can be answered by decomposing them into modular subtasks. However, existing video data management systems assume the existence of predefined modules for each subtask. We introduce VOCAL-UDF, a novel self-enhancing system that supports compositional queries over videos without the need for predefined modules. VOCAL-UDF automatically identifies and constructs missing modules and encapsulates them as user-defined functions (UDFs), thus expanding its querying capabilities. To achieve this, we formulate a unified UDF model that leverages large language models (LLMs) to aid in new UDF generation. VOCAL-UDF handles a wide range of concepts by supporting both *program-based UDFs* (i.e., Python functions generated by LLMs) and *distilled-model UDFs* (lightweight vision models distilled from strong pretrained models). To resolve the inherent ambiguity in user intent, VOCAL-UDF generates multiple candidate UDFs and uses active learning to efficiently select the best one. With the self-enhancing capability, VOCAL-UDF significantly improves query performance across three video datasets.

CCS Concepts: • **Information systems → Data management systems**; **Multimedia databases**; **Video search**; • **Computing methodologies → Visual content-based indexing and retrieval**.

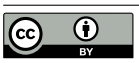
Additional Key Words and Phrases: Video Analytics, Compositional Queries, Scene Graphs, Large Language Models, Vision-Language Models, Program Generation, Knowledge Distillation

ACM Reference Format:
Enhao Zhang, Nicole Sullivan, Brandon Haynes, Ranjay Krishna, and Magdalena Balazinska. 2025. Self-Enhancing Video Data Management System for Compositional Events with Large Language Models. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 215 (June 2025), 29 pages. https://doi.org/10.1145/3725352

1 Introduction

Rapid advances in video analytics have fueled the development of innovative applications across various fields. In medical education, surgery videos enhance students' procedural knowledge by illustrating complex temporal and spatial events [38]. In biology, scientists use wildlife footage to study organism behaviors and interactions in their natural habitats [34, 92]. In transportation, traffic surveillance systems analyze and manage traffic flow, improving urban mobility [43]. Across these applications, analysts seek to query video databases for events characterized by spatio-temporal and semantic interactions. For instance, an analyst might search for "a motorcycle swerving near a silver Subaru and then colliding with it" or "a doctor holding a scalpel before placing it on a table."

Authors' Contact Information: Enhao Zhang, University of Washington, Seattle, USA, enhaoz@cs.washington.edu; Nicole Sullivan, University of Washington, Seattle, USA, nsulliv@cs.washington.edu; Brandon Haynes, Microsoft Gray Systems Lab, Redmond, USA, brandon.haynes@microsoft.com; Ranjay Krishna, University of Washington, Seattle, USA, ranjay@cs.washington.edu; Magdalena Balazinska, University of Washington, Seattle, USA, magda@cs.washington.edu.



ACM 2836-6573/2025/6-ART215
https://doi.org/10.1145/3725352

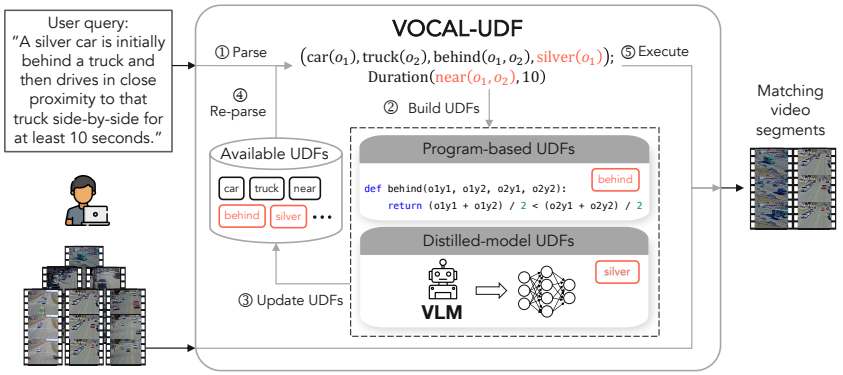


Fig. 1. Given a video dataset and a user query in natural language, VOCAL-UDF ① parses the query into a DSL notation. If the query contains predicates that existing UDFs cannot answer, VOCAL-UDF ② automatically builds new UDFs, ③ updates its available UDF list, ④ reparses the query, and ⑤ executes the query to return matching video segments. VOCAL-UDF supports both program-based UDFs (i.e., Python functions) and distilled-model UDFs (i.e., ML models) for diverse concepts.

Though promising, answering video queries using frontier vision-language models (VLMs) remains underwhelming. Although VLMs have demonstrated notable capabilities on diverse, challenging tasks [68, 69, 87], they struggle to answer *compositional queries* [74] that involve recognizing objects (e.g., "car", "truck"), reasoning about relationships (e.g., "behind", "holding"), and identifying attributes (e.g., "silver color", "Subaru make"). This challenge is further amplified when queries require temporal reasoning [112] (e.g., "X then Y", "X for at least 10 seconds"). While new models continue to improve their ability to reason spatially [19] and compositionally [52], their performance remains low [103]. Additionally, deploying large models at scale is prohibitively expensive [84] and inference is slow [99]. For example, current VLMs can only achieve a throughput of around 100 tokens per second [3, 10], making their use in large-scale video analytics intractable.

Alternatively, *workflow-oriented* video data management systems (VDBMSs) answer compositional queries by decomposing them into granular subtasks [8, 13, 39, 42, 76, 94, 97, 118]. Submodules identify and track objects, attributes, and relationships across frames, forming spatio-temporal scene graphs [53, 63]. These scene graphs can expressively represent many complex visual queries. Various scene graph generation techniques [31, 116] have been proposed to extract scene graphs from images and videos. In these systems, subtasks that extract scene graph elements are solved individually and then composed to answer a compositional query [71].

However, VDBMSs make a critical assumption: the existence of modules capable of executing subtasks to answer a complex query. Systems typically provide a variety of built-in modules [42, 76, 94, 97, 118] and often allow extensibility via user-defined functions (UDFs) for unsupported scenarios [54, 107, 118]. Concerning our motorcycle query, a user might need to supply a UDF to filter for "silver" or nearby objects if the system lacks these capabilities. Despite the availability of pre-trained computer vision models that can be readily integrated as UDFs, users may require solutions for domain-specific applications or seek to identify fine-grained object classes and subjective concepts for which no off-the-shelf models exist. Identifying or adapting domain-specific models in such cases may be possible, though tedious. Additionally, it may be necessary to—on a per-UDF basis—curate datasets and perform extensive training to achieve satisfactory performance [8].

To address these challenges, we present VOCAL-UDF (Figure 1), a *self-enhancing* VDBMS that empowers users to flexibly issue and answer compositional queries, even when the necessary modules are unavailable. To use VOCAL-UDF, a user only needs to provide a video dataset and a

natural language (NL) description of the query. VOCAL-UDF then *automatically* identifies and builds the necessary modules and encapsulates them as new UDFs to expand its querying capabilities. It then compiles the NL query into a declarative one that it executes over the video dataset.

There are several challenges in building VOCAL-UDF. First, given an NL query, VOCAL-UDF converts it into a declarative one that it can execute. Unfortunately, existing methods assume the existence of predefined modules that can be invoked to construct the declarative query [32, 118]. VOCAL-UDF addresses this challenge by identifying any semantic concepts that are not supported by existing modules or UDFs during compilation and leveraging an LLM's reasoning to determine when and which new UDFs are needed.

The second challenge involves transforming the various identified missing semantic concepts into executable modules. While LLMs can produce useful code in various contexts [9, 22, 26, 110], queries on video data are often highly ambiguous and their performance varies significantly (see Section 6.3). To address the challenge of handling a multitude of missing concepts, VOCAL-UDF automatically implements two types of UDFs—*program-based UDFs* and *distilled-model UDFs*. In our system, program-based UDFs are imperative Python functions that operate on relational tables and video pixels. While this class of UDFs can classify many relationships and attributes with high quality [62, 118], we show in Section 4.3 the need for distilled-model UDFs, which are lightweight machine learning (ML) models that are trained on the fly to classify more nuanced concepts [98]. VOCAL-UDF leverages LLMs for both UDF types: program-based UDFs harness an LLM's programming capability, whereas distilled-model UDFs rely on the LLM's ability to annotate and convert visual concepts into ML models [98]. To mitigate potential erroneous LLM-generated UDFs, VOCAL-UDF implements syntax and semantic verification steps.

The third challenge involves managing the inherent ambiguity in user intent when articulating a query. This is especially important for specialized and subjective concepts that are difficult to resolve without user feedback. For example, users may have different interpretations of the "near" relationship. Further compounding this difficulty is our observation that for some inputs a program-based UDF might perform better than a distilled-model UDF, while in other cases the opposite is true. Finally, for the most challenging semantic concepts, *no* UDF may perform well and employing one might risk overall query performance. To address this formidable combination of challenges, VOCAL-UDF generates *multiple* candidate UDFs each with different semantic interpretations, implementations, and properties. VOCAL-UDF then employs active learning to efficiently identify the variant that best matches the user's intent.

Finally, the self-building nature of our system necessitates a consistent and unified UDF model. Typically, a UDF can be an arbitrary function that operates on database tuples. However, one challenge lies in modeling UDFs in a structured way to handle concepts of objects, relationships, and attributes while seamlessly integrating and interacting with various system components. To address this, we propose a unified UDF scheme for different semantic concepts, which enables the LLM to generate UDFs in a structured format and simplifies the compilation process. By incrementally growing the database, future UDFs can be composed using existing ones.

We evaluate VOCAL-UDF on three video datasets from different domains [37, 53, 111] and show that it significantly improves query performance, in terms of F1 score, by automatically selecting, implementing, and executing its automatically-generated UDFs. We finally conduct a thorough analysis of VOCAL-UDF to understand its efficiency in terms of execution time and monetary cost.

Overall, VOCAL-UDF's self-enhancing capability is an important step toward making VDBMSs more practical to deploy and use in a variety of applications.

2 Background

Table 1. Relational schema representation of data model.

`Frames`(vid, fid, pixels)
`Objects`(vid, fid, oid, oname, x_1, y_1, x_2, y_2)
`Relationships`(vid, fid, rid, oid_1, rname, oid_2)
`Attributes`(vid, fid, oid, aname)

Table 2. Relational views for query execution with UDFs.

`ObjView`(vid, video_pixels)
`RelView`(vid, fid, pixels, rid, o1_o2_rnames, o2_o1_rnames, o1_oid, o1_x1, o1_y1, o1_x2, o1_y2, o1_anames, o2_oid, o2_x1, o2_x2, o2_y1, o2_y2, o2_anames)
`AttrView`(vid, fid, pixels, oid, oname, x1, y1, x2, y2, anames)

Listing 1. An example of the query language.

```
% The first graph detects a car far from a truck
g1(vid, fid, fid, oid1, oid2) :- Objects(vid, fid, oid1, 'car', _, _, _, _),
    Objects(vid, fid, oid2, 'truck', _, _, _, _),
    Relationships(vid, fid, _, oid1, 'far', oid2), oid1 != oid2.
% The second graph detects the 'near' relationship
g2(vid, fid, oid1, oid2) :- Relationships(vid, fid, _, oid1, 'near', oid2), oid1 != oid2.
% Use recursive rules to declare duration constraints
g2_star(vid, fid, fid, oid1, oid2) :- g2(vid, fid, oid1, oid2).
g2_star(vid, fid_start, fid_end, oid1, oid2) :- g2_star(vid, fid_start, fid, oid1, oid2),
    g2(vid, fid_end, oid1, oid2), fid_end = fid + 1.
% Finally, define the temporal order of the two region graphs
q(vid) :- g1(vid, fid11, fid12, oid1, oid2), g2_star(vid, fid21, fid22, oid1, oid2),
    fid21 > fid12, fid22 - fid21 + 1 > 10 * 24. % Assume 24 frames per second
```

VOCAL-UDF leverages EQUI-VOCAL's [118] scene graph data model and query language, which models compositional video events as spatio-temporal scene graphs. This approach draws from cognitive foundations in human perception [14, 64, 115] and enables a variety of compositional queries [118]. This section summarizes key background information about these concepts.

Data model. In VOCAL-UDF, each video comprises a series of N frames $\{f_1, \ldots, f_N\}$. The visual content of each frame is represented by a *scene graph* $g_i = (\mathbf{o}_i, \mathbf{r}_i)$, capturing all *objects* $\mathbf{o}_i$ and all *relationships* $\mathbf{r}_i$ between those objects within the frame at some time. Objects may also possess *attributes*. While a relationship links two objects, an attribute is attached to one object. A *region graph* g_{ij} is a subgraph of g_i, i.e., $g_{ij} \subseteq g_i$, that contains information critical for identifying an event. Finally, an *event* e is a sequence of region graphs $e = \{g_1, \ldots, g_k\}$, where region graphs with a smaller index occur earlier in time than those with a larger index, but they do not need to be contiguous or distinct. The relational schema in Table 1 captures the scene graphs data model.

Query language. In VOCAL-UDF, a query identifies video segments that match a user-specified event. VOCAL-UDF supports relational queries over the schema in Table 1, which are of the following form. Using Datalog notation: $q(vid)$:- $g_1, \ldots, g_k, \mathbf{p}, \mathbf{d}, w$, where $g_1, \ldots, g_k$ is a temporally ordered sequence of region graphs specifying that a matching event consists of g_1, followed by g_2, followed by g_3, etc. Each g_i can persist for multiple frames and there can be other frames between g_i and g_{i+1}. $\mathbf{p}$ is a set of predicates that are applied to objects, relationships, and attributes in region graphs. $\mathbf{d}$ is a set of constraints on the duration for which a region graph must remain valid before transitioning to the next one. Lastly, w is the maximum number of frames between g_1 and g_k. As an example, the event "A car is initially far from a truck, then remains close to the truck for more than 10 seconds" can be expressed as in Listing 1.

DSL. VOCAL-UDF adopts EQUI-VOCAL's domain-specific language (DSL) [118], which encapsulates query logic while abstracting away SQL details. The query executor compiles these DSL queries into efficient SQL for execution over relational tables. Using the DSL, the same event can be expressed as (assuming 24 frames per second): $(\texttt{Car}(o_1), \texttt{truck}(o_2), \texttt{far}(o_1, o_2); \texttt{Duration}(\texttt{near}(o_1, o_2), 240)$. In this DSL, the *variable* o represents an arbitrary object in a query, with distinct subscripts indicating objects with different *oid*'s. All predicates in a region graph are separated by commas. Region graphs are then sequenced in temporal order using semicolons. Each region graph can

Listing 2. SQL query to identify frames where a silver car is behind a truck.

```
SELECT DISTINCT f.vid, f.fid, o1.oid, o2.oid
FROM frames f, objects AS o1, objects AS o2
WHERE f.vid = o1.vid AND f.fid = o1.fid AND o1.vid = o2.vid AND o1.fid = o2.fid
    AND o1.oid <> o2.oid AND car(o1.oname) = TRUE AND truck(o2.oname) = TRUE
    AND behind(o1.y1, o1.y2, o2.y1, o2.y2) = TRUE
    AND silver(f.pixels, o1.x1, o1.y1, o1.x2, o1.y2) = TRUE
```

persist for multiple frames and there can be other frames between two adjacent region graphs. Finally, `Duration`(g, d) stipulates that the region graph g exists in *at least* d consecutive frames. The query returns a set of video segment identifiers.

3 A UDF-based data model

In this section, we formalize the types of UDFs supported by VOCAL-UDF. These UDFs allow users to define custom objects, relationships, and attributes. We then describe how UDFs are compiled and executed in VOCAL-UDF.

Table 1 shows VOCAL-UDF's relational schema. The `Frames` relation includes a virtual `pixels` column that stores the frame pixel values in a 3D array ($H \times W \times 3$, where H is height, W is width, and 3 represents the color channels). The `Objects`, `Relationships`, and `Attributes` relations store corresponding detected frame elements.

A UDF extends database functionality. A typical UDF takes columns as input, returns a scalar value or a row set, and is used in SQL statements, e.g., in `WHERE` clauses. When querying video databases, VOCAL-UDF supports generating and executing UDFs to identify custom objects, relationships, and attributes, enabling users to find complex, compositional events. Listing 2 shows an example SQL query with UDFs for the objects `car` and `truck`, the relationship `behind`, and the attribute `silver`. VOCAL-UDF supports imperative and declarative Python-based UDFs, and are categorized into the following classes: relationship, attribute, object, and value-lookup.

A **relationship UDF** or **attribute UDF** is a deterministic, scalar predicate that indicates whether an input exhibits a specified relationship or attribute. It accepts zero (i.e., a dummy UDF; see Section 4.4.2) or more columns as arguments and produces a boolean result. The parameters can include any of the following columns from each table in the SQL query's `FROM` clause: `pixels`, `oname`, `x1`, `y1`, `x2`, `y2`, `rname`, and `aname`. VOCAL-UDF restricts relationship UDFs and attribute UDFs to be frame-level, i.e., they operate on object(s) within the same frame. Therefore, input arguments are all from the same video frame (identified by `vid` and `fid`) and are associated with one or two distinct objects. As an example, we might define a relationship UDF to indicate whether object o_1 is behind another object o_2 by comparing their centroid y-coordinates:

```
def behind(o1_y1, o1_y2, o2_y1, o2_y2):
    return (o1_y1 + o1_y2) / 2 < (o2_y1 + o2_y2) / 2
```

We could also define an attribute UDF to indicate if a detected car is silver by running an ML model over the frame pixels:

```
def silver(pixels, x1, y1, x2, y2):
    cropped_img = pixels[y1:y2, x1:x2]
    is_silver = awesome_color_classifier(cropped_img)
    return is_silver
```

An **object UDF** requires localizing, classifying, and tracking objects in videos. Instead of returning a boolean value, it is a table-valued function that takes a video segment, `video_pixels`, as input. A video segment comprises frames with the same `vid` concatenated into a 4D array with an additional

dimension for the frame index. An object UDF detects and tracks objects of a specific class. Given a video segment, the UDF makes calls to a custom object detection and tracking model and returns a row set, which follows the `Objects` schema listing the detected and tracked objects. As an example, we can define an object UDF that detects all cars in a video:

```
def car(video_pixels):
    obj_tuples = []
    car_detector, tracker = load_models()
    for frame in video_pixels:
        detected_cars = car_detector(frame)
        tracked_objs = tracker.update(detected_cars)
        obj_tuples.extend(tracked_objs)
    return obj_tuples
```

In this paper, we assume that object UDFs are given and focus on proposing and generating relationship and attribute UDFs. We leave the extension to object UDFs for future work.

A **value-lookup UDF** is a class of UDFs that simply encapsulate a predicate over existing column values. As an example, suppose the value "car" is in the domain of the object `oname`. Then, a value-lookup UDF can be defined as:

```
def car(oname):
    return oname == 'car'
```

While value-lookup UDFs are not strictly necessary, as they can be directly and easily expressed in SQL statements, we wrap all predicates of our DSL queries in UDFs to simplify the compilation from the DSL to SQL.

UDFs can have different lists of parameters in their signatures. To execute a query with UDFs, VOCAL-UDF first constructs the relational views as shown in Table 2 derived from the relations in Table 1. The views–`ObjView`, `RelView` and `AttrView`–contain all attributes that an object UDF, relationship UDF, and attribute UDF can potentially accept as input arguments, respectively. For simplicity, VOCAL-UDF generates UDFs with a uniform list of parameters for each class.

Object, relationship, and attribute UDFs can be expensive to evaluate, as they may operate on image pixels and invoke ML models. To optimize query execution, VOCAL-UDF caches the results of all UDFs and replaces them with value-lookup UDFs. When executing a UDF over a video corpus for the first time, VOCAL-UDF materializes the results to make it available as a value in the corresponding column and substitutes the UDF with a value-lookup UDF of the same name. For example, when running a query with the `silver` attribute UDF for the first time, VOCAL-UDF evaluates the UDF for each object in the video corpus. If an object `OID1` in a frame `F1` of a video segment `V1` is classified as silver, VOCAL-UDF inserts a new row with values (`V1`, `F1`, `OID1`, 'silver') into the `Attributes` relation. Then, VOCAL-UDF replaces the `silver` attribute UDF with a value-lookup UDF that checks whether the `aname` is 'silver'. Reusing the results of predicate evaluation for query optimization is a long-standing research topic [36, 78, 89, 107] and is not the focus of this paper. Batch inference can further accelerate ML-based UDFs.

4 VOCAL-UDF approach

VOCAL-UDF needs to address several challenges. First, it must determine whether existing UDFs can adequately answer a user query, or if new UDFs should be created (C1). Second, VOCAL-UDF should support implementing UDFs drawn from diverse range of semantic concepts (C2). Third, since VOCAL-UDF utilizes error-prone LLMs to generate UDFs, it is crucial to ensure high quality in the produced UDFs (C3). We now discuss our solutions to each challenge.

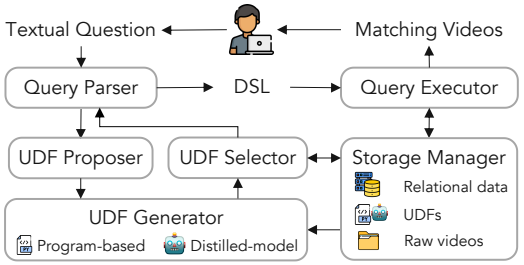


Fig. 2. VOCAL-UDF system overview.

Figure 2 shows the architecture of VOCAL-UDF. The user initializes the system with a video dataset and an optional set of UDFs. After preprocessing (Section 5), the user can issue NL queries to identify compositional events within the videos. The *Query Parser* (Section 4.1) parses the query into the DSL notation described in Section 2. If successful, the DSL query is passed to the *Query Executor* to find all matching videos in the dataset. If the query contains predicates that existing UDFs cannot resolve, the *UDF Proposer* (Section 4.1) is invoked to propose the names and descriptions of new UDFs. The *UDF Generator* (Sections 4.2 and 4.3) creates executable candidates for each proposed UDF. Next, the *UDF Selector* (Section 4.4) solicits user labels to select the implementation that best aligns with the user's intent. With updated UDFs, the *Query Parser* re-parses the query for execution. The *Storage Manager* maintains raw videos, relational data, and available UDFs.

4.1 Query parsing and UDF proposal

The Query Parser converts NL queries into our DSL and determines the need for new UDFs. VOCAL-UDF must understand the semantics of the user's query and available UDFs, mapping each part of the query to an existing UDF or suggesting the creation of a new one. Moreover, VOCAL-UDF needs to be resilient to the linguistic ambiguities and synonymous terms in NL queries.

VOCAL-UDF utilizes LLMs to convert complex NL to a consistent DSL format. LLMs show strong capabilities in SQL and program generations [42, 83, 100] via in-context learning without fine-tuning. We incorporate domain-specific constraints into LLM prompts to ensure adherence to DSL grammar and conduct post-verification to ensure syntax correctness. Inspired by Wang et. al. [100] and Hsieh et. al. [50], VOCAL-UDF provides the DSL definition, UDF format, and descriptions of available UDFs as NL documentation. This enables VOCAL-UDF to generate grammatically correct DSL queries, determine whether new UDFs need to be created, and interact with LLMs in a zero-shot manner (C1). While VOCAL-UDF focuses on zero-shot prompting, introducing few-shot examples [15] could potentially further improve UDF proposal performance. However, referencing unavailable UDFs in examples may lead LLMs to incorrectly assume their availability at query time. Since the available UDFs can change as the database evolves, examples must be carefully curated for different database states. Figuring out the optimal prompting strategy, though beneficial, is not the focus of this paper. An example prompt is shown in our technical report [119].

To enhance LLM response reliability, VOCAL-UDF performs post-verification of the generated DSL query to ensure it is syntactically valid and only uses available UDFs. If parse errors occur, VOCAL-UDF appends the error message to the context and asks the LLM to make another attempt.

When the query contains predicates that cannot be resolved using the available UDFs, we prompt the LLM to identify this and propose new UDFs. The output from this process is the function signature and textual description of the proposed UDF generated by the LLM. In case of a query

with multiple missing UDFs, VOCAL-UDF returns a list of proposed UDFs and will generate them one by one. An example proposed UDF of `behind` is:

```
{"signature": "behind(o0, o1)", "description": "Whether o0 is behind o1"}
```

Since both the Query Parser and UDF Proposer utilize LLMs, their outputs may sometimes be inaccurate. If the LLM proposes fewer UDFs than needed, it may make performance improvements less noticeable but will not degrade performance. Conversely, proposing more UDFs can increase system runtime and monetary costs; however, as detailed in Section 4.4, we have implemented techniques to ensure that generated UDFs (whether superfluous or inaccurate) do not adversely affect performance. Finally, users may also rephrase their NL query if they are not fully satisfied with the query results.

We show the effectiveness of our approach empirically in Section 6 for queries with detailed and explicit descriptions. Semantic parsing (e.g., text-to-SQL) is a highly active research area with significant recent advances [35, 40, 58, 114, 120]. While current methods still fall short of human-level performance [67], improving text-to-SQL is not the goal of our paper. VOCAL-UDF focuses on the efficient generation of missing UDFs, relies on LLMs currently adequate performance for text-to-SQL in our context, and will benefit from future advances in this area.

4.2 Program-based UDF generation

The UDF Generator implements executable UDFs based on the LLM-proposed UDF signatures and descriptions. As discussed, VOCAL-UDF should produce high-quality UDFs (C3) for a wide range of semantic concepts (C2). While VOCAL-UDF leverages the programming capabilities of LLMs [16, 20, 24, 90] to generate UDFs as Python programs, more work is required. Some tasks require complex visual understanding, rendering it challenging to solve via programs (e.g., determining a car's make). Additionally, even the most advanced LLMs remain error-prone and their programming performance for nuanced tasks is not yet on par with humans.

To address the first problem, VOCAL-UDF supports two types of UDF implementations: program-based UDFs, which are Python programs generated by LLMs (Section 4.2), and distilled-model UDFs, which are lightweight vision models distilled from strong pretrained models (Section 4.3). To improve generated UDF quality, VOCAL-UDF uses a two-step approach by verifying both syntactic (Section 4.2.2) and semantic (Section 4.4) correctness.

At a high level, given a UDF signature h and description d, a video database instance I over schema R, VOCAL-UDF performs the following steps to generate a UDF p:

1. *Generate UDF candidates using an LLM*. Given h, d, I, and R, VOCAL-UDF prompts an LLM to generate a set of k candidate Python functions $\{p_1, ..., p_k\}$.
2. *Syntactically verify candidates*. VOCAL-UDF executes each candidate p_i on a small dataset sampled from I to verify syntactic correctness, ensuring that all passing candidates are executable.
3. *Semantically verify candidates*. VOCAL-UDF finally evaluates the semantic correctness of the remaining candidates and selects the best one (see Section 4.4).

We next describe each step in further detail.

4.2.1 *Generating candidate programs using an LLM.*

In its most basic form, VOCAL-UDF prompts the LLM to generate a program p based on the UDF signature h and description d. To ensure p can be eventually expressed as a SQL predicate, VOCAL-UDF rewrites h in the DSL format to h' that accepts columns from the schema R as inputs. For example, the `behind` function is rewritten from `behind(o0, o1)` to `behind(o1_y1, o1_y2, o2_y1, o2_y2)`, where `o1_y1`, `o1_y2`, `o2_y1`, `o2_y2` are columns of R. The generated Python program's input is a set of attribute values that correspond to one or two objects in the video

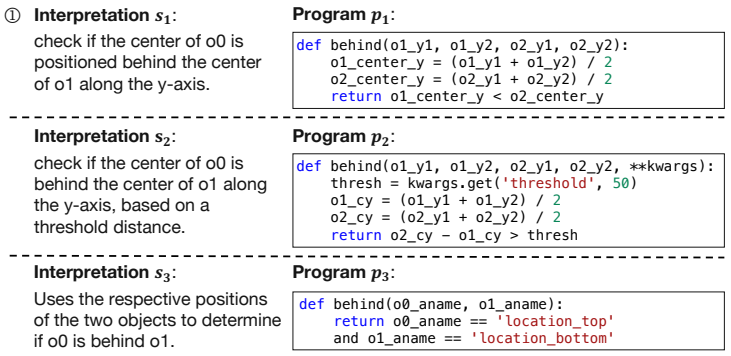


Fig. 3. Program candidates, using behind as an example.

database, depending on the number of variables in h. It generates code that operates on those values, including the pixels column, and returns a boolean indicating if the predicate is satisfied or not. Figure 3 (program p_1) shows an example program UDF for the behind(·) predicate.

To generate quality UDFs, VOCAL-UDF must resolve three challenges. First, NL descriptions are often ambiguous and may not entirely capture the user intent [65] (e.g., user-specific definition of "far"). Second, predicates in video compositional queries often include hyperparameters that require tuning [76, 77] for different datasets and user intents, e.g., determining the threshold distance for the "far" relationship. Lastly, as VOCAL-UDF expands its collection of UDFs and database incrementally, it is essential that new UDFs be able to utilize results from previously established ones.

To address linguistic ambiguity, we provide the LLM with h' and d and ask it to generate a list of k candidate programs with a variety of semantic interpretations: $C = \{(p_1, s_1), \cdots, (p_k, s_k)\}$, where p_i and s_i respectively denote a Python program and its semantic interpretation. [119] shows an example prompt used by VOCAL-UDF in this step. Later, VOCAL-UDF verifies and selects one program for each proposed UDF. See Figure 3 for candidate program examples.

To resolve the challenge of parametric predicates, VOCAL-UDF additionally prompts the LLM to generate UDFs with optional numeric hyperparameters and their valid ranges. More formally, VOCAL-UDF prompts the LLM to produce a (possibly empty) list $\Theta = \{(\theta_1, \mathrm{df}_{\theta_1}, \min_{\theta_1}, \max_{\theta_1}), \cdots, (\theta_i, \mathrm{df}_{\theta_k}, \min_{\theta_k}, \max_{\theta_k})\}$ of parameter names θ_i, default value df_{θ_i}, and range $[\min_{\theta_i}, \max_{\theta_i}]$. In later steps, VOCAL-UDF instantiates each hyperparameter with its default value as well as values sampled from the range and then selects the best program. See Figure 3② for an example of a program that relies on a hyperparameter.

To resolve the final challenge of incrementally building the video database, our approach is to additionally include in the LLM prompt, the active domain of all attributes populated by UDFs, which include oname, rname, and aname. This additional information enables the LLM to leverage the results of existing UDFs as building blocks to dynamically compose more complex UDFs. Figure 3③ shows an example program inlining an existing UDF.

Overall, program-based UDFs are well-suited for concepts involving bounding box-like spatial relationships. In addition, other attributes and the pixels column can also be used to reason about existing concepts and perform statistical analysis of the pixels in a frame. A program-based UDF could also invoke a pretrained model over each frame, but this is in general expensive and slow. As discussed in Section 4.3, a distilled-model UDF is a more efficient approach for concepts that require visual understanding of videos.

4.2.2 Syntax verification.

Prior works have proposed various approaches to improve the performance of LLM-generated programs. One line of work leverages unit tests to verify the functional correctness of generated programs [18, 22, 27, 52]. This approach is not suitable for VOCAL-UDF because users do not know in advance what UDFs will be generates, and thus cannot provide labeled data before issuing a query. Another line of work leverages LLMs to automatically generate test cases [20], evaluate the generated programs [24, 95], and select the best one [104]. However, the ambiguity of semantic concepts in video queries means that the correct program is not always unique and often many of the generated programs are reasonable. As a result, the best program cannot be easily identified without user feedback and a working dataset.

Additionally, the quality of an LLM is not guaranteed. To address this issue, VOCAL-UDF uses a two-step approach to verify and select the best program: syntax verification and semantic verification. In the first step, VOCAL-UDF focuses solely on the syntax correctness of the programs. VOCAL-UDF executes each candidate program on a small sample of data from the database I and checks whether: (i) the number and types of inputs and outputs are correct, (ii) the program can be executed with the data samples, and (iii) Θ, if any, can be parsed successfully. If the verification fails, VOCAL-UDF appends the error message to the context and prompt the LLM to make another attempt. VOCAL-UDF draws samples by constructing tuples that contain attributes and values that respectively correspond to one or two objects in the database. If the program still fails after a few trials, VOCAL-UDF discards the program. In our prototype, we empirically set the number of trials to five. Recent research has demonstrated the potential for formally verifying the correctness of generated UDFs based on a given semantic interpretation [23], which could be incorporated into VOCAL-UDF to further enhance the reliability of its generated Python programs. After generating multiple candidate programs, VOCAL-UDF next utilizes user labels to select the one that best aligns with user intent, which we discuss in Section 4.4.

4.3 Distilled-model UDF generation

While program-based UDFs are powerful and flexible for predicates that reason about existing concepts, bounding box coordinates, or perform a simple statistical analysis of pixel values, they struggle with tasks that require understanding the visual contents of frames. Even though pretrained models like VLMs can be used in a program-based UDF to classify relationships and attributes in a zero-shot manner, running such models over the entire video dataset is expensive. For instance, applying GPT-4o to every frame of the 10,000 five-second videos in the CLEVRER dataset [111] to evaluate a predicate (e.g., "color-red" or some more complex predicate) would cost approximately $1,413. This cost remains high even when applying techniques such as downsampling or predicate push-down: There are about 1.28M video frames and 5.44M objects to consider. Even if 99% of the image patches corresponding to these objects were filtered, the cost of evaluating a predicate on the remaining 1% of objects would still be around $14—far higher than generating one UDF (about $0.15; see Section 6.1). VOCAL-UDF's goal is to generate cheaper UDFs with visual understanding capabilities. However, training a lightweight image classifier from scratch would require the user to spend a lot of time labeling for just one concept.

Model distillation is a common technique in machine learning to transfer knowledge from a large model to a smaller, more efficient model [17, 21, 48, 49, 98]. Modeling Collaborator [98] is a newly proposed framework that leverages foundation models to train image classifiers for visual concepts using minimal user effort. To do so, given a target concept and description, the system (i) mines relevant images from the public domain, (ii) uses foundation models to annotate sampled images, (iii) trains a lightweight classifier using features extracted from a pretrained model (e.g.,

Fig. 4. Data labeling by a VLM, using `behind` as an example.

CLIP) and labels annotated by the foundation models, and (iv) performs multiple rounds of active learning to further improve its performance.

VOCAL-UDF adopts a similar approach to automatically construct lightweight image classifiers for new concepts without requiring user labeling but with the modifications needed to resolve three unique challenges presented in our compositional query setting. First, random sampling of the user dataset might not give enough positive samples for training, especially for rare concepts. Second, VOCAL-UDF generates UDFs for relationships and attributes, which differ from the concepts in [98]. An image usually includes multiple objects, and VOCAL-UDF needs a different prompting strategy to guide a VLM in classifying specific objects or pairs of objects*. Finally, for the same reason, merely extracting features from the entire image is insufficient to train a good classifier.

4.3.1 Image sampling. To generate a distilled-model UDF, VOCAL-UDF initially randomly samples frames from the user's video dataset for annotation. However, when the target visual concept is infrequent in the dataset, random sampling does not effectively collect enough positive samples for training. For instance, only 0.83% human-object pairs have an "eating" relationship in the Charades [53] dataset. Our solution is to apply *object-aware sampling* to bootstrap the sampling process. Since all objects of interest are already detected and tracked in the dataset, VOCAL-UDF can filter out irrelevant objects and sample only those likely to be involved in the target concept. For example, when labeling the "eating" relationship, object classes like "food" and "person" are more relevant than "car" and "window". To do this, VOCAL-UDF first asks an LLM for relevant object classes, and then only samples objects belonging to these classes. [119] shows an example prompt of object-aware sampling.

4.3.2 Data labeling. VOCAL-UDF uses VLMs to automatically label sampled video frames as positive or negative based on a UDF description. A VLM takes as input an image-text pair and outputs a textual response. However, using a video frame and the UDF description as a direct query is ineffective, since the concepts we are interested in target specific objects or pairs of objects, rather than the frame as a whole. Thus, VOCAL-UDF applies the following prompting strategy, with the goal of encouraging the VLM to focus on particular objects or interactions between two objects. For attribute concepts, we use the UDF description proposed in Section 4.1 as the text input, and create the image input by sampling an object from the `Objects` relation and cropping the video frame to include only the object. For relationship concepts, VOCAL-UDF augments the text input with the class names and bounding box coordinates of the relevant objects in the video frame to provide more context to the VLM. It then generates an image patch cropped from the video frame that includes a pair of objects in the same frame from the `Objects` relation. VOCAL-UDF further augments the image patch by overlaying a red box around the subject and a blue box around the target, thereby providing the VLM with directional information about the relationship. Figure 4 shows an example prompt for labeling the `behind` relationship.

4.3.3 Model training. Similar to [33, 96, 98], VOCAL-UDF leverages a pretrained vision model (e.g., CLIP) as the feature extractor and uses the feature-label pairs to train a multi-layer perceptron (MLP). However, for relationship classification, directly extracting features from the image patch

containing two objects would perform poorly, as the feature extractor is not aware of the object locations and cannot capture directional relationships (e.g., o_1 holding o_2 vs. o_2 holding o_1). To address this, VOCAL-UDF concatenates features from three versions of an image patch: the original one containing both objects, one where everything except the subject is masked, and one where everything except the target is masked. For attribute UDFs, VOCAL-UDF simply extracts features from the image patch containing the object. In addition, VOCAL-UDF incorporates text features of object class names, derived from the `Objects` relation, to enhance the MLP's performance. After training an initial model, VOCAL-UDF uses an active learning approach similar to [96, 98] to iteratively improve the model. During each iteration, the trained MLP is run over the unlabeled dataset, VOCAL-UDF selects a batch of samples with the highest uncertainty for labeling by the VLM. VOCAL-UDF then retrains the MLP with the updated labeled dataset.

VOCAL-UDF uses VLMs to automatically label data and the cost is bounded by the training data size, which is at most 500 frames in our evaluation. This is only 0.04% of the 1.28M frames in the entire database for CLEVRER, 0.8% of the 65K frames for CityFlow-NL, and 0.2% of the 289K frames for Charades (see Section 6), which correspond to extremely low sampling rates. This cost is thus far lower than directly applying an LLM to the entire video dataset, even with downsampling. Although VOCAL-UDF's distilled model process may occasionally deliver unsatisfactory performance, its UDF Selector component, described next, mitigates this by selecting the best candidate UDFs and filtering low-performing models.

4.4 UDF Selection

For a given UDF, VOCAL-UDF can generate both a distilled-model and a set of program-based implementations. When there are multiple implementation candidates, VOCAL-UDF needs to select the best one. VOCAL-UDF does not have any initial data to validate semantic correctness, only the user NL query. If VOCAL-UDF were to collect labeled data for validation before query execution, the user would need to manually go through the video dataset to find positive and negative examples. Instead, VOCAL-UDF utilizes active learning at query time to strategically request binary labels on carefully selected samples that are most likely to resolve disagreements among UDF candidates. This approach minimizes labeling effort while effectively guiding the UDF selection process, which we now describe in detail.

4.4.1 UDF selection and active learning. To best align with the user's intent, VOCAL-UDF strives to select the candidate UDF that yields the best F1 score for a given set of user labels. As summarized in Algorithm 1, VOCAL-UDF uses active learning [57, 81, 118] to reduce the number of labeled examples needed from the user. Key hyperparameters include: a labeling budget b, thresholds t_p and t_n giving the number of positive and negative samples needed to initiate active learning, and the number of tuples n_s sampled per iteration. Section 6 details the values used in our experiments.

During each iteration, the algorithm randomly samples n_s tuples from the database (Line 4). It operates in two phases: *bootstrapping* and *active learning*. The bootstrapping phase collects at least t_p positive and t_n negative samples. While the number of labeled positives ($|L_p|$) or negatives ($|L_n|$) remains below its respective threshold, VOCAL-UDF selects samples most likely to be positive or negative (Lines 5 to 8). Once enough initial labeled data has been collected, VOCAL-UDF transitions to active learning, adopting the margin and positive strategy from [81]. Specifically, if $|L_p| < |L_n|$, it asks the user to label samples most likely to be positive; otherwise, it picks the sample with the greatest disagreement among UDF candidates to differentiate them [118] (Lines 9 to 12).

In PickPositive(U_s, C, W) and PickNegative(U_s, C, W), the algorithm priorities VLM annotations obtained in Section 4.3.2 over the UDF candidates' majority vote, as VLM labels are empirically

Algorithm 1: UDF selection using active learning.

Input : Set of unlabeled data U, set of UDF candidates C, and hyperparameters b, n_s, t_p, t_n
Output: Selected UDF with highest score

1 $L_p \leftarrow \{\}, L_n \leftarrow \{\}$
2 $W \leftarrow \{w_i \mid w_i = 1/|C|, i = 1, 2, \ldots, |C|\}$
3 **for** $i = 1$ ***to*** b **do**
4 $\quad U_s \leftarrow \textsc{SampleSubset}(U, n_s)$
$\quad$ `/* Phase 1: Bootstrapping */`
5 $\quad$ **if** $|L_p| < t_p$ **then**
6 $\quad\quad L'_p, L'_n \leftarrow \textsc{PickPositive}(U_s, C, W)$
7 $\quad$ **else if** $|L_n| < t_n$ **then**
8 $\quad\quad L'_p, L'_n \leftarrow \textsc{PickNegative}(U_s, C, W)$
$\quad$ `/* Phase 2: Active learning */`
9 $\quad$ **else if** $|L_p| < |L_n|$ **then**
10 $\quad\quad L'_p, L'_n \leftarrow \textsc{PickPositive}(U_s, C, W)$
11 $\quad$ **else**
12 $\quad\quad L'_p, L'_n \leftarrow \textsc{PickDisagreed}(U_s, C, W)$
13 $\quad L_p \leftarrow L_p \cup L'_p, L_n \leftarrow L_n \cup L'_n, U \leftarrow U - (L'_p \cup L'_n)$
14 $\quad W \leftarrow \textsc{ComputeScore}(L_p, L_n, C)$

more reliable. In $\textsc{PickDisagreed}(U_s, C, W)$, it computes a disagreement score for each sample in U_s based on UDF candidates C, selecting the one with the highest disagreement [57, 118]. Each sample's disagreement score is the weighted disagreement among UDF candidates, with candidate weights initialized to $1/|C|$ (Line 2) and updated to its performance (F1 score in our prototype) over L_p and L_n after each iteration (Line 14). At the end of each iteration, L_p, L_n and U are updated with the new user labels (Line 13).

4.4.2 Dummy UDFs. Several factors can contribute to the unsatisfactory performance of generated UDFs, including the complexity of the target concept, the low quality of VLM labels, and an insufficient number of positive training samples for rare events. When no UDF candidate performs well, VOCAL-UDF should not apply a UDF that can hurt the performance. To prevent this, VOCAL-UDF appends a *dummy UDF* to its list of UDF candidates. A dummy UDF is a constant-valued function that produces `True` and is equivalent to omitting the predicate. When all other candidates perform poorly, VOCAL-UDF opts for the dummy UDF during its UDF selection process.

4.4.3 UDF generation strategy. Program-based UDFs and distilled-model UDFs offer different trade-offs in terms of performance, interpretability, generation cost, and inference throughput. VOCAL-UDF provides four strategies depending on the user's preference and the system's performance requirements: `program` only generates program-based UDFs, `model` only generates distilled-model UDFs, `llm` asks the LLM to decide whether to generate program-based or distilled-model UDFs, and `both` generates both classes of UDFs. [119] shows an example prompt of using the `llm` UDF generation strategy. Depending on the user specification, one or more program-based UDFs may be generated for each proposed UDF, and one distilled-model UDF is generated for each proposed UDF. By default, VOCAL-UDF uses the `both` strategy to maximize query performance. In Section 6.4, we also evaluate the performance of using the `llm` strategy to automatically choose UDF types.

4.5 Comparing with direct LLM methods

As an alternative to VOCAL-UDF, a multimodal LLM can directly answer user queries. However, as shown in prior work [74, 103] and as we confirm in Section 6.1.2, LLMs still struggle with

compositional queries, and directly providing the LLM with the video frames and the user query in a single prompt (the "**vanilla LLM**" approach in Section 6.1.2) leads to low F1 scores. Furthermore, the cost scales with the dataset size, making it expensive for large video collections today, as we discussed in Section 4.3 and evaluate in Table 3.

A higher performance LLM baseline is to parse the query into our DSL, then use a combination of existing cheap models for common concepts and an LLM for application-specific concepts (i.e., VOCAL-UDF without UDF generation, or "**LLM for concepts**" in Table 3). While it may yield good F1 scores, this approach is only practical for short videos and a small number of queries. For larger workloads, the cost and latency of this method are even greater than "vanilla LLM" due to repeated LLM calls to identify concepts for each object or object pair. As Table 3 shows, executing just one query with three missing UDFs over the CLEVRER dataset using GPT-4o costs about $1,504 USD. Reducing costs through sampling is also non-trivial because low sampling rates can degrade query performance, especially for compositional events involving temporal dimensions. As we show in Figure 6b, even a small downsampling factor of 8 can greatly reduce the upper bound of achievable F1 scores from 1.0 to 0.51.

VOCAL-UDF overcomes these issues by using the multimodal LLM to learn each new application-specific concept once, over a fixed-size sample of the data, and generate cheaper, domain-specific models (either program-based or distilled-model UDFs) that can then be applied to the remaining videos and to all future queries, avoiding repeated reliance on the LLM and reducing overall costs.

5 Implementation

VOCAL-UDF is implemented in Python using the AutoGen framework [105]. The Query Executor converts queries in DSL notation into SQL and uses a relational engine (DuckDB [86] in our prototype) to execute them. We apply the same query translation algorithm as in [118] to optimize query execution. To reduce system latency, we parallelize API calls to the LLM.

VOCAL-UDF requires a preprocessing stage per dataset before queries can be issued. The user provides a video dataset and optionally a set of UDFs. If these UDFs include object detection and tracking models, VOCAL-UDF uses them to populate the `Objects` relation. Otherwise, VOCAL-UDF uses a predefined object detection and tracking model to identify common objects. VOCAL-UDF pre-extracts features into Parquet files for use by distilled models at query time (Section 4.3.3), and we utilize NVIDIA DALI [5] to accelerate this process. VOCAL-UDF also executes the initial UDFs ahead of time to populate the `Relationships` and `Attributes` relations and compute the active domains for `oname`, `rname`, and `aname`. These operations happen as soon as VOCAL-UDF receives the videos and UDFs and before the user issues any queries. If no initial UDFs are supplied, the `Relationships` and `Attributes` relations will be empty. Similarly, when a new UDF is later created, our prototype runs it over the entire dataset and materializes the results in the database. It then converts the UDF into a value-lookup UDF so that subsequent queries can reuse the materialized results. All UDFs are implemented in Python.

There is preprocessing overhead in the initial setup. The cost of running object detection, object tracking, and feature extraction is relatively modest compared to that of LLMs and VLMs, and they can be run locally. The cost of executing a UDF depends on its implementation. In this paper, we materialize all results in the video dataset before issuing new queries. Several works have focused on optimizing query execution with UDFs [54, 107], which could be integrated into our system.

6 Evaluation

Baselines. We compare VOCAL-UDF against VisProg [42] and EQUI-VOCAL [118], both of which are limited to predefined UDFs when answering queries. VisProg uses a different set of modules for different tasks, so we manually write new modules for the system to be able to answer compositional queries. These include both logical (e.g., `Eval`, `Event`, `Before`) and conceptual (e.g., `Red`, `Holding`) modules. VisProg also requires in-context examples, which we create separately from the evaluation queries. While EQUI-VOCAL does not need such examples, it does not accept NL queries as input. To address this, we adapt VOCAL-UDF's approach of converting NL into the EQUI-VOCAL DSL using an LLM. We further compare VOCAL-UDF with two LLM baselines as described in Section 4.5.
Metrics. We evaluate query answering performance using F1 scores, precision, and recall. We propose and generate UDFs using training data and report query performance over the test set. We evaluate each dataset using 30 queries, and each query is run three times. We additionally evaluate the system efficiency in terms of latency and cost. For latency, we report the wall-clock time as perceived by the user from the moment they submit a query to the moment when VOCAL-UDF returns the query results. For cost, we measure both the LLM API invocation cost using the OpenAI pricing model [6] and the resource cost using the AWS pricing model [1, 2].
Datasets: We evaluate VOCAL-UDF on three datasets: CLEVRER [111], CityFlow-NL [37] and Charades [53]. CLEVRER is a benchmark that facilitates testing queries with varying complexities. CityFlow-NL and Charades represent real-world applications—traffic monitoring and human activities—typical for video data management systems. UDFs in CityFlow-NL include fine-grained vehicle types, while Charades comes with complex human interactions. The evaluation queries are based on concepts from these datasets, which are labeled by the original annotators and reflect events from those domains.
CLEVRER: The CLEVRER [111] dataset consists of 10,000, 5-second synthetic videos of moving objects. To determine ground-truth information, following [111], we use a Mask R-CNN [46] to locate objects and predict their colors, shapes, and materials. We write rule-based functions to extract spatial relationships and attributes. For our experiments, we use the same Mask R-CNN and rule-based functions as UDFs, for a total of six relationships and 17 attributes. Among them, we select two relationships and nine attributes as the *base UDFs*, which are available to all systems during query evaluation. We use templates to automatically generate 30 target queries with seven predicates, up to three variables (i.e., three distinct objects), up to three region graphs, and duration constraints with three possible values. We ensure there are at least 5% positive examples in the dataset for each target query (we ensure the same for all datasets). We then rewrite the DSL queries as NL queries using GPT-4 and manually verify the correctness.
CityFlow-NL: The CityFlow-NL [37] dataset contains traffic videos captured from multiple cameras and NL descriptions for vehicle tracks. Following [66], we extract vehicle colors and types from the NL descriptions. Since only sampled vehicle tracks are annotated, we only consider them in our evaluation. We extract six spatial relationships using rule-based predictors and create 1,473 non-overlapping 50-frame video segments from the original dataset. To create UDFs for attributes, we train binary image classifiers. For relationships, we use the same rule-based functions as UDFs. Then, we select three relationships and four attributes as *base UDFs*. We automatically generate 30 queries with up to four predicates, up to three variables, up to three region graphs, and duration constraints with three possible values. An example query is "a vehicle o0 with type sedan is to the right of another vehicle o1 for at least 15 frames. Then, o0 is to the left of o1 for at least 15 frames", which could be used to find sedans changing their positions with other vehicles.
Charades: The Charades [53] dataset contains 30-second indoor activity videos. We focus exclusively on frames with scene graph annotations provided by Action Genome [53], which only provide human-object relationships but do not cover attributes or object-object relationships. Thus, our evaluation queries focus on human interactions with objects. At query time, only human-object

pairs are sampled to collect more positive examples. Incomplete annotations is a common issue in real-world datasets [19] like Charades. To address this, we augment the dataset by automatically generating dense spatial relationship annotations using rule-based predictors to replace the existing ones. For semantic relationships, we train binary image classifiers. In total, we have 20 relationships (five spatial, 15 semantic) and 35 object classes. Among them, we select two spatial and seven semantic relationships as the base UDFs. We automatically generate 30 target queries with up to four predicates (one is always `object(`o_i`,'person')`), up to three variables, and up to two region graphs, but without duration constraints. An example query is "a person is eating something, and then the same person is holding another object while being in front of it", which could retrieve videos including a person taking a bite from an apple, and then picking up a cookbook in front of them to read a recipe.

For each dataset, we use half of the videos as training data and the rest as test data. To generate ground truth labels, we run each target DSL query on the dataset.

Evaluation setup. We conduct all experiments on a compute cluster. For each experiment, we request one node with eight Intel Xeon Gold 6230R CPUs at 2.10GHz, 200GB of RAM, and one NVIDIA A40 GPU. We use the GPT-4o model (`gpt-4o-2024-08-06`) as the LLM and VLM across all systems. We configure VOCAL-UDF as follows. For the CLEVRER dataset, we use a labeling budget of 20 for each UDF during selection. We generate 10 candidate programs, allowing numeric parameters and frame pixels as inputs. During UDF selection, we bind each numeric parameter to its default value as well as five random values. When generating distilled-model UDFs, we ask the VLM to annotate 100 sampled frames per UDF. Given their increased complexity, for the CityFlow-NL and Charades datasets, we set the labeling budget per UDF to 50 and the number of VLM-annotated frames per UDF to 500. 100 frames are annotated for the initial training, and then active learning is performed to select 100 additional frames in each round. Frame pixels are disallowed in program-based UDF generation due to their limited impact on performance. To focus our evaluation on relationships and attributes, we assume objects are pre-detected and tracked.

6.1 End-to-end performance

6.1.1 F1 score, precision, and recall. We first evaluate the end-to-end performance of VOCAL-UDF and baseline systems by varying the number of missing UDFs from zero (i.e., all required UDFs are available) to three. In each experiment, all base UDFs are available, but X supplemental UDFs are randomly removed. We execute thirty distinct queries three times. Figure 5 presents the F1 scores, precision, and recall. **VOCAL-UDF maintains high performance even when UDFs are missing.** When all required UDFs are available, all systems perform similarly. For CLEVRER, F1 scores with no missing UDFs reach 1.0 since predefined UDFs are also used to generate ground-truth scene graph annotations. However, CityFlow-NL and Charades contain less accurate ML models in their UDFs, leading to lower F1 scores. When UDFs are missing, VisProg and EQUI-VOCAL experience significant F1 score drops, as they cannot generate new UDFs, with VisProg performing worse due to its higher likelihood of including unavailable UDFs. VOCAL-UDF mitigates F1 score degradation by generating new UDFs as needed, though performance drops more on Charades as its missing UDFs involve complex, harder-to-generate semantic relationships. Interestingly, while EQUI-VOCAL achieves higher recall than VOCAL-UDF on CLEVRER and CityFlow-NL when UDFs are missing, it does so at the expense of substantially lower precision, for an overall lower F1 score.

6.1.2 Comparing with direct LLM methods.

To understand the performance gap between VOCAL-UDF and using an LLM directly, we evaluate both approaches on CLEVRER. We use the same queries from the end-to-end experiments, but under a simplified setting due to the high expense of GPT-4o. We select 500 videos and

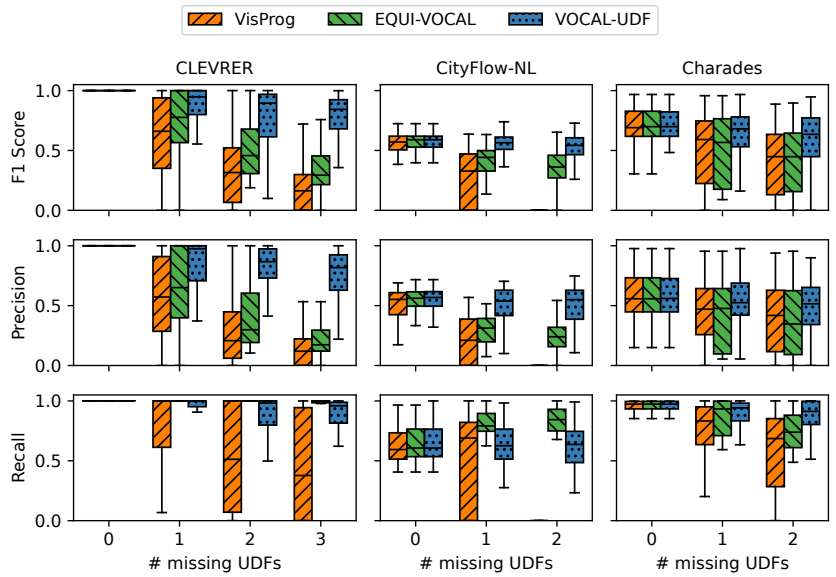
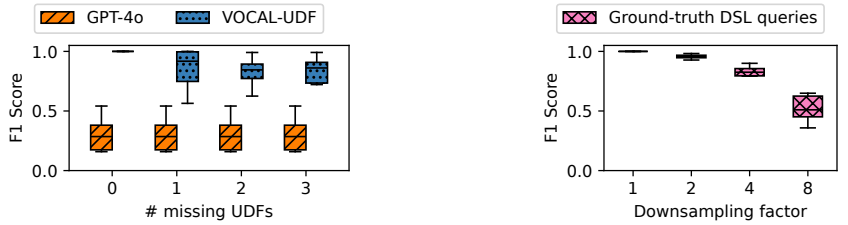


Fig. 5. End-to-end performance of generated queries with various number of missing UDFs.



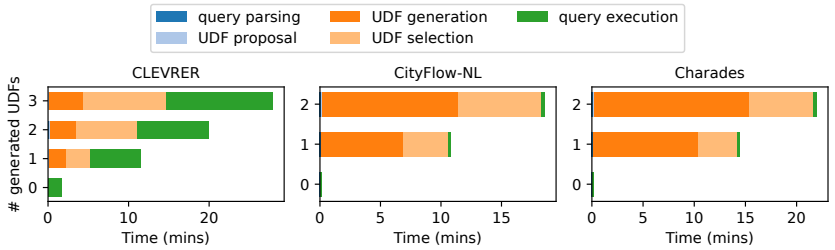
(a) Vanilla LLM vs. VOCAL-UDF.

(b) Downsampling reduces F1 scores.

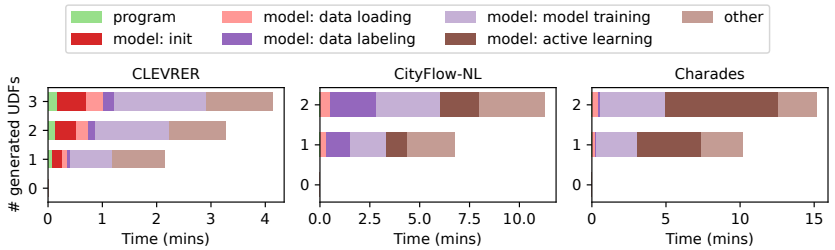
Fig. 6. (a) F1 scores of the vanilla LLM approach and VOCAL-UDF on CLEVRER under a simplified setting. (b) F1 scores of running ground-truth DSL queries over sampled videos.

downsample each by 75%, retaining every fourth frame. We use the first 10 queries as evaluated in Section 6.1.1 but remove any duration constraints due to the downsampling. We compare VOCAL-UDF to a vanilla LLM method in which we provide GPT-4o with a sequence of video frames and ask with the prompt: "Examine the sequence of frames from a video and determine if the following event occurs? <user_query>. Answer with 'yes' or 'no'." We use OpenAI's Batch API that offers 50% lower cost. Figure 6a shows that **VOCAL-UDF achieves significantly higher F1 scores than the vanilla LLM approach.** The cost of the vanilla LLM method scales with the dataset size, averaging $5.17 USD per query under this simplified setting. If extended to the full 5,000 videos in the test set of CLEVRER, the estimated cost would be $207 USD per query.

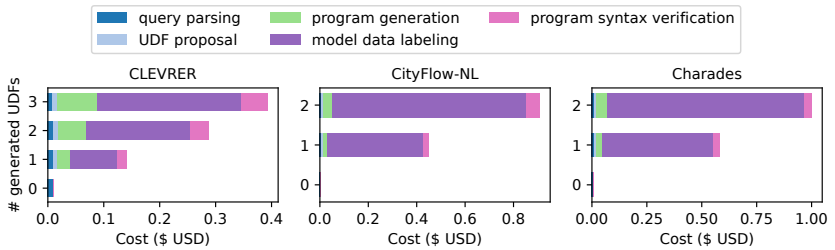
As an alternative baseline, one can use an LLM to classify new concepts instead of generating UDFs. However, as discussed later in Section 6.1.4, the cost of doing so without downsampling is prohibitively high. While downsampling reduces LLM costs, it can also degrade query performance. Figure 6b shows the F1 scores of executing ground-truth DSL queries (i.e., assuming perfect LLM predictions) over the simplified CLEVRER with various downsampling factors. **Downsampling**



(a) Total time.



(b) UDF generation time.



(c) Monetary cost of OpenAI API invocation.

Fig. 7. Detailed breakdown of efficiency and cost.

greatly reduces the upper bound on achievable F1 scores, with performance dropping to 0.51 at a downsampling factor of just 8.

6.1.3 Execution time. To analyze VOCAL-UDF's efficiency, we perform an end-to-end evaluation of execution time. Figure 7a breaks down the wall-clock time for each system component, with each stacked bar showing the mean execution time for queries with different numbers of UDFs that VOCAL-UDF *decides to generate*. VOCAL-UDF may occasionally decide to generate extra, unnecessary UDFs. For clarity, we omit CLEVRER with four generated UDFs and CityFlow-NL with three generated UDFs from Figure 7, as each has only one data point—arising from the LLM's inaccurate decision—and does not reflect an average execution time for that number of generated UDFs. **The average execution time is under 100 seconds when all required UDFs are available and increases proportionally with the number of generated UDFs.** In CLEVRER, query execution takes longer when new UDFs need to be created, requiring several minutes to process the full 14-hour video dataset as VOCAL-UDF materializes new UDF outputs. Subsequent queries using the same UDFs, however, are as fast as the case with no missing UDFs. CityFlow-NL and Charades are faster due to smaller datasets and the exclusion of frame pixels in program-based UDF generation, which eliminates the need for video loading and reduces the program execution cost. For CityFlow-NL and Charades, UDF generation dominates the execution time, as the number

Table 3. LLM cost estimation for executing one query over CLEVRER's full test set, with three missing UDFs.

Method	GPT-4o	Llama 3.1 70B
VOCAL-UDF	\$0.4	\$0.03–0.11
Vanilla LLM	\$207	\$59
LLM for concepts	\$1504	\$430

Table 4. Natural language to DSL correctness.

Dataset	F1 = 1	F1 ≥ 0.98
CLEVRER	79%	100%
CityFlow-NL	92%	99%
Charades	83%	97%

of VLM-annotated frames per UDF increases to 500, with active learning applied after labeling every 100 samples. UDF selection requires non-negligible time across all datasets, involving tasks such as frame loading, feature extraction, program execution, and model prediction.

Figure 7b provides a detailed breakdown of wall-clock time for the UDF generation process. Since VOCAL-UDF asynchronously parallelizes calls to the LLM, we measure the time breakdown as follows. All categories except for "other" capture the synchronous portions of each sub-step (i.e., excluding LLM calls), while "other" represents the remaining observed time, which includes the time spent explicitly waiting on asynchronous LLM calls. Program-based UDF generation is labeled as "program," while distilled-model UDF generation is further divided into sub-steps ("init", "data loading", etc.). **Distilled-model UDF generation takes significantly longer than program-based UDFs, but no single sub-step consistently dominates across all cases.** Specifically, training distilled models takes the longest in CLEVRER and CityFlow-NL, whereas active learning takes the longest in Charades due to a larger dataset. The active learning step could be accelerated by performing it on a sample of the unlabeled dataset. The results also suggest that **LLM calls do not dominate UDF generation time**, thanks to asynchronous execution. For example, during data labeling, VOCAL-UDF submits batches of 100 asynchronous tasks to maximize parallelization.

6.1.4 Monetary cost. To understand the query execution cost, we consider both the LLM API invocation cost and the compute resource cost. Figure 7c shows the mean monetary cost of OpenAI LLM calls per query. **VOCAL-UDF costs less than \$1.0 USD per query, with data labeling for distilled-model UDFs being the primary cost factor.** Moreover, GPT-4o can be replaced by open-source models. For instance, Amazon Bedrock offers Llama 3.1 70B for a price of \$0.72 USD per 1M tokens [1], which is 3.5× cheaper for input and 14× cheaper for output tokens than GPT-4o. Other open-source models and hosting options [4] are available to further reduce costs, with a potential trade-off in query F1 scores. Compute resource costs depend on deployment scenarios. For example, on a `g4dn.12xlarge` EC2 spot instance, the estimated average resource cost of running a query with the largest number of missing UDFs is \$0.7 USD for CLEVRER, \$0.4 for CityFlow-NL, and \$0.5 for Charades [2]. Thus, **the total per-query cost for each dataset remains under \$1.5.**

Table 3 compares the estimated LLM costs of VOCAL-UDF, "vanilla LLM", and "LLM for concepts" (as described in Section 4.5) using both GPT-4o and Llama 3.1 70B. Due to the high expense of the baselines, we estimated costs rather than running full experiments. We estimate the cost of "LLM for concepts" as follows: Each CLEVRER video contains an average of 544 objects and 1,933 object pairs. Each LLM call requires 300 input tokens (255 for the image and 45 for the task) and one output token ("yes" or "no"). We first run the query with missing UDFs removed, then invoke the LLM only on the matching videos to minimize costs. As shown in Table 3, **VOCAL-UDF is much more cost-effective than direct LLM baselines.** While further optimizations—e.g., evaluating immutable attributes on a single frame and propagating them to other frames for the same object—could reduce the cost of "LLM for concepts", the cost remains proportional to the dataset size, in contrast to VOCAL-UDF.

6.2 UDF proposal

Table 5. Proposing UDFs. FPs are incorrectly proposed UDFs, and FNs are missed UDFs.

Dataset	# New UDFs	# proposed UDFs	# FP	# FN
CLEVRER	540	441	27	126
CityFlow-NL	270	261	4	14
Charades	270	229	19	60

Table 6. Program-based and distilled-model UDFs performance.

Dataset	Program-based UDFs performance				Model-based UDFs performance			
	best = "program"		best ≠ "program"		best = "model"		best ≠ "model"	
	best UDFs	all UDFs	best UDFs	all UDFs	model	dummy	model	dummy
CLEVRER	0.995	0.203	0.664	0.024	0.889	0.519	0.556	0.662
CityFlow-NL	1.000	0.831	0.349	0.203	0.687	0.289	0.735	0.664
Charades	1.000	0.092	0.162	0.030	0.724	0.207	0.302	0.728

Translating NL queries to DSL is not a contribution of this paper, but we report it as it affects the end-to-end performance. For this experiment, all UDFs are available, and the LLM translates the same queries from the end-to-end experiment, running each three times. We measure the F1 score of the generated queries. As shown in Table 4, **VOCAL-UDF can effectively parse the complex textual queries into the DSL notation**, returning correct video segments at least 79% of the time, which increases to ≥97% if we consider an F1 score ≥ 0.98. Common translation errors include missing and redundant predicates, misuse of UDFs with similar names, and incorrect temporal ordering of predicates.

We evaluate VOCAL-UDF's UDF proposal using the same set-up as the end-to-end evaluation. A proposed UDF is considered correct if its name matches one of the target query's ground truth UDFs, with a manual check for synonyms. As shown in Table 5, **VOCAL-UDF proposes fewer UDFs than expected, with more false negatives (FNs) than false positives (FPs).** This occurs because some new UDFs are equivalent to available UDFs (e.g., "above(o1, o2)" vs. "beneath(o2, o1)"). However, this is beneficial as it avoids unnecessary UDF generation. Most other FNs are mainly because VOCAL-UDF decides to approximate using other available UDFs (e.g., "touching" for "holding"). **Most FPs arise from proposing compositional UDFs**, such as "behind_and_near" for "behind" and "near".

6.3 UDF generation

We now evaluate the performance of program-based UDFs. We consider all correctly proposed program-based UDFs across the 270 experiments with the largest number of new UDFs (3 datasets × 30 queries × 3 runs). We classify UDFs into two categories, one where at least one of their best implementations is program-based, and another where the best is not program-based. Table 6 shows the median F1 score of the best generated program for each UDF (best UDFs) and all generated programs for each UDF (all UDFs). **When the best UDF generation is program-based, VOCAL-UDF consistently produces high-quality programs.** However, not all generated UDF candidates are of high quality; for instance, the median F1 score of all program-based UDFs for Charades is only 0.092. Thus, VOCAL-UDF must carefully select the best candidate during the UDF selection phase. When the best UDF is not program-based, F1 scores decrease significantly, indicating that certain relationships or attributes may not be well-suited to program-based UDFs.

Using the same method as above, we evaluate the performance of distilled-model UDFs. UDFs are classified into two categories, one where at least one of their best implementations is distilled-model, and another where none are distilled-model. Table 6 shows the median F1 score for both distilled-model generation and dummy generation (as the baseline) for each UDF. **When the best**

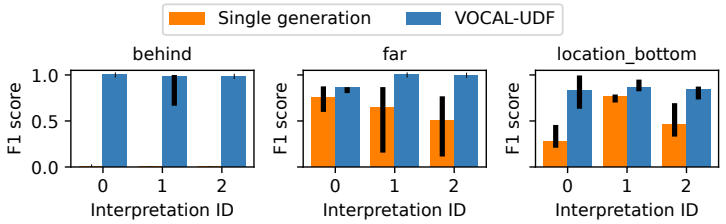


Fig. 8. Median F1 scores of VOCAL-UDF vs. a single-generation baseline on queries with one predicate under various interpretations, each run 20 times. Error bars show the IQR.

Table 7. VLM labeling quality.

CityFlow-NL		Charades			
suv	0.843	holding	0.776	sitting on	0.897
white	0.906	standing on	0.846	covered by	0.749
grey	0.797	carrying	0.761	eating	0.737
van	0.956	wiping	0.685	touching	0.632
sedan	0.903	leaning on	0.816	wearing	0.854
black	0.819	drinking from	0.916	lying on	0.857
red	0.973	writing on	0.894	above	0.683
blue	0.904	in front of	0.644	beneath	0.561
pickup truck	0.956	behind	0.599	in	0.444

UDF generated is distilled model, VOCAL-UDF shows a significant improvement in F1 scores over the baseline. When the best UDF is not distilled model, the F1 scores of distilled models are comparable or worse than the baseline, indicating that certain relationships or attributes may not be suitable for distilled-model UDFs. For CityFlow-NL, F1 scores are higher for UDFs whose best implementations are not distilled models, likely due to the complexity of semantic attributes that are more challenging to classify.

To access VOCAL-UDF's ability to handle ambiguous intent in program-based UDFs, we test three concepts from CLEVRER—"behind", "far", and "location_bottom". For each concept, we execute the same query with one predicate but three different interpretations. For instance, "behind" is interpreted as: (1) o1's center above o2's center, (2) o1's center below o2's center, and (3) o1's bottom edge above o2's bottom edge. We compare VOCAL-UDF to a baseline that uses an LLM to generate a single program without numeric hyperparameters or UDF reuse. Figure 8 shows the F1 scores for each concept across varying interpretations, showing that **VOCAL-UDF can more robustly adapt to different definitions of the same concept**. The baseline struggles with "behind" since it uses an interpretation based on bounding box overlap that mismatches all three ground truths. While it works well for one interpretation of the other concepts, its single-generation method fails to accommodate varying interpretations, leading to significant performance drops. One limitation of VOCAL-UDF is that a concept becomes fixed once a UDF is generated. The UDF would have to be explicitly deleted or disabled if a user wanted a different interpretation for an already-existing concept. Additionally, VOCAL-UDF currently does not handle intent ambiguity for distilled-model UDFs, but this can be extended by asking LLMs to generate diverse descriptions of the target concept and distilling multiple models.

We further examine the VLM's labeling quality during distilled-model UDF generation on the CityFlow-NL and Charades datasets. We evaluate nine attributes in CityFlow-NL and 18 relationships in Charades. For each attribute or relationship, we use GPT-4o to label a randomly selected, balanced set of 500 samples. Table 7 shows the average F1 score of each concept over three runs. **GPT-4o generally achieves high labeling quality for a wide range of concepts**:

Table 8. UDF selection performance and selected UDF distribution, using "both" strategy.

Dataset	No.	best	80% of best	program	model	dummy
CLEVRER	232	163 (70%)	218 (94%)	183 (79%)	37 (16%)	12 (5%)
CityFlow-NL	174	144 (83%)	154 (89%)	94 (54%)	76 (44%)	4 (2%)
Charades	145	93 (64%)	123 (85%)	81 (56%)	23 (16%)	41 (28%)

Table 9. Number of correctly selected UDF types. "No." represents the number of proposed UDF instances.

Dataset	best ≠ "dummy"				best = "dummy"	
	No.	both	llm (gpt-4o)	llm (gpt-4-turbo)	No.	both
CLEVRER	225	203 (90%)	208 (92%)	145 (64%)	7	3 (43%)
CityFlow-NL	174	154 (89%)	170 (98%)	102 (59%)	0	0 (—)
Charades	125	90 (72%)	104 (83%)	57 (46%)	20	18 (90%)

seven out of the 27 concepts achieve F1 scores of at least 0.9, and 15 attain F1 scores of at least 0.8. However, GPT-4o struggles to label the five spatial relationships, suggesting that **VLMs like GPT-4o are still limited in spatial reasoning**.

6.4 UDF selection

Table 8 reports the UDF selection results. The "best" column shows the number of UDFs selected by VOCAL-UDF that achieve the highest F1 score among all UDF candidates, while the "80% of best" column shows how many attain at least 80% of the best F1 score. **VOCAL-UDF effectively selects better-performing UDFs from candidates, even with a labeling budget as low as 20.** Although VOCAL-UDF does not always pick the best UDF due to similar scores for many candidate UDFs, it can still select a good UDF implementation at least 85% of the time (with an F1 score of at least 80% of the best implementation). Table 8 also breaks down the types of UDFs that VOCAL-UDF selects. **To handle diverse semantic concepts, VOCAL-UDF generates and selects UDFs of various types**, underscoring the necessity for VOCAL-UDF to support two different types of UDFs. Interestingly, VOCAL-UDF selects more dummy UDFs (41 instances, 28%) on the Charades dataset than the other datasets. Notably, 20 instances are "holding", which is difficult to distinguish under the current labeling budget. Another 17 cases are "in", which our rule-based predictor identifies by checking bounding box overlap. This method classifies most object pairs as having this relationship, thereby allowing even the dummy UDFs to perform exceptionally well.

VOCAL-UDF supports four UDF generation strategies, as described in Section 4.4.3. We now compare the "both" and "llm" strategies in selecting the correct UDF type. When multiple candidates of different types share the highest F1 score, any of these types is considered correct. Using the same end-to-end experiment setting with the largest number of missing UDFs, Table 9 shows the number of correctly selected UDF types. The results are divided into two categories: one where the best UDF type is not "dummy" and one where it is "dummy." When the best UDF type is not "dummy," **VOCAL-UDF correctly selects the UDF type at least 72% of the time with the "both" strategy**, reaching up to 90% on CLEVRER. Interestingly, **using "llm" with GPT-4o yields a higher accuracy than "both"**. However, this strategy is highly sensitive to the model used; switching to an earlier GPT-4 Turbo model reduces the accuracy to 46–64%. Despite this, the "llm" strategy demonstrates potential to further improve performance while reducing latency and cost. When the best UDF type is "dummy," VOCAL-UDF can also select the correct UDF type 43% to 90% of the time using the "both" strategy.

We study the impact of active learning and dummy UDFs to the UDF selection process, using the same end-to-end experiment setting with the largest number of new UDFs. Figure 9 compares

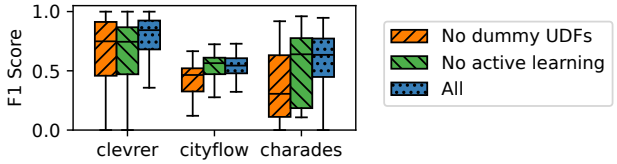


Fig. 9. F1 scores of VOCAL-UDF without active learning and without dummy UDFs in UDF selection.

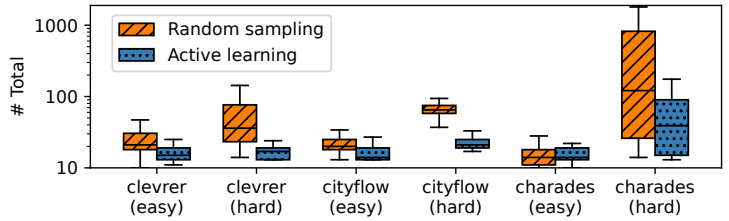


Fig. 10. Number of samples required to get at least 10 positive examples (y-axis in log scale; lower is better).

F1 scores across three system variants: the complete system with both active learning and dummy UDFs (All), the system with random sampling and dummy UDFs (No active learning), and the system with active learning but without dummy UDFs (No dummy UDFs). **Both active learning and dummy UDFs help achieve higher F1 scores.** For CityFlow-NL, the best UDF implementations are easily distinguishable from candidates, so active learning does not improve F1 scores but also does not degrade system performance.

VOCAL-UDF reduces user effort by using active learning to request labels at query time rather than asking the user to provide all examples upfront. Figure 10 compares the number of samples needed under two approaches—active learning during UDF selection and random sampling before query execution—to obtain at least ten positive examples for a given UDF. We consider all generated UDFs from the end-to-end experiment and classify them as "easy" (spatial relationships and attributes with relatively balanced labels) or "hard" (other concepts with greater class imbalance). **Active learning allows VOCAL-UDF to collect positive samples with reduced labeling effort, particularly for "hard" concepts.** Since most object pairs exhibit "in" relationships in Charades, random sampling requires labeling only 11.5 samples on average to yield 10 positives. As a result, the overall labeling effort for "easy" concepts in Charades is similar between random sampling and active learning.

7 Related work

Video analytics. Numerous video analytics systems have been developed to support a wide range of data management tasks [32, 33, 60, 61, 88]. Query execution over videos typically involves running expensive ML models. Thus, many techniques have been proposed to accelerate query processing, including indexing [47, 51, 56], sampling [11, 12, 80], pre-filtering frames [44, 45, 73, 106], reusing results [107], and building specialized models [7, 55]. VOCAL-UDF can incorporate existing methods to optimize query execution.

Compositional video query processing. VOCAL-UDF is most related to systems designed for compositional video queries [13, 25, 30, 39, 70, 76, 107, 108, 113, 118]. However, these systems often require users to have a certain level of database expertise to manually construct compositional queries [25, 39, 70, 107, 113] or to provide examples to learn a query from [76, 118]. In contrast, VOCAL-UDF leverages advances in LLMs and allows expressing queries in NL.

LLMs with tools. LLMs are widely used to tackle challenging text tasks across a variety of applications [22, 26, 41, 59, 82, 93, 101, 102]. By integrating external tools, LLMs can address even more complex reasoning tasks [72, 75, 85, 91, 121], including vision tasks [29, 42, 79, 94, 97]. However, these systems generally rely on the availability of existing tools or modules. For instance, VisProg [42] and ProViQ [29] utilize LLMs to transform complex tasks into executable programs that invoke predefined tools. VOCAL-UDF also employs LLMs to handle compositional queries over videos but extends this capability by generating new UDFs. Several systems explore the capacity of LLMs to create new tools. LATM [18] generates reusable code snippets for NL tasks, while GENOME [27] generates and reuses code-based modules to solve visual tasks. VOCAL-UDF distinguishes itself as an end-to-end VDBMS that generates both program-based and distilled-model UDFs which significantly enhances performance.

8 Limitations

VOCAL-UDF does not currently support relationship and attribute UDFs involving state changes over time, such as identifying a car "moving close" to a truck, which requires distance comparisons at different timestamps. As an approximation, multiple frame-level UDFs can represent each state separately; for instance, detecting a "far" relationship followed by a "near" one.

Materializing and reusing UDF results can be problematic when the definition of an attribute or relationship is not objective (e.g., "near") and varies from query to query. To address this, VOCAL-UDF could allow users to access and manage generated UDF descriptions, implementations, and results, enabling them to dynamically enable or disable specific UDFs before issuing queries.

VOCAL-UDF's performance is affected by its ability to semantically understand the user queries and available UDFs. We assume that predefined UDFs are relevant to the target domain and that an LLM is capable of disambiguating each UDF effectively. The performance of VOCAL-UDF thus heavily depends on the underlying LLM, which is why we used the latest LLMs (e.g., GPT-4o) to ensure high-quality semantic understanding. Furthermore, VOCAL-UDF currently supports queries with detailed and explicit descriptions. Extending support to vague queries is left for future work.

VOCAL-UDF leverages LLMs in various system components, making the quality of query results dependent on the underlying LLM performance. To enhance the reliability of LLM-generated answers, VOCAL-UDF employs syntax verification, generates multiple candidate UDFs, and leverages user annotations via active learning to select the best implementation. Introducing dummy UDFs further ensures that newly generated UDFs do not negatively impact query results. While users already provide annotations to guide UDF selection, VOCAL-UDF could be extended to incorporate additional human interventions, allowing users to access intermediate results for manual examination [28, 109, 117]. Our experiments empirically demonstrate the promising potential of our LLM-based approach, and we posit that as LLMs advance, VOCAL-UDF performance will correspondingly improve.

9 Conclusion

This paper presents VOCAL-UDF, a new system that supports compositional video queries with the capability to generate new UDFs. VOCAL-UDF utilizes LLMs to parse natural language queries and automatically determine the need for new UDFs. It supports both program-based and distilled-model UDF generations and improves UDF quality through syntax verification and semantic verification.

Acknowledgments

This work was funded in part by NSF award 2211133. Nicole Sullivan was supported in part by an NSF graduate fellowship (DGE-2140004).

References

[1] 2024. Amazon Bedrock Pricing. https://aws.amazon.com/bedrock/pricing.

[2] 2024. Amazon EC2 Spot Instances Pricing. https://aws.amazon.com/ec2/spot/pricing.

[3] 2024. Analysis: GPT-4o vs GPT-4 Turbo. https://www.vellum.ai/blog/analysis-gpt-4o-vs-gpt-4-turbo.

[4] 2024. Llama 3 70B vs GPT-4: Comparison Analysis. https://www.vellum.ai/blog/llama-3-70b-vs-gpt-4-comparison-analysis.

[5] 2024. NVIDIA DALI. https://developer.nvidia.com/dali.

[6] 2024. OpenAI API pricing. https://openai.com/api/pricing.

[7] Michael R. Anderson, Michael J. Cafarella, Germán Ros, and Thomas F. Wenisch. 2019. Physical Representation-Based Predicate Optimization for a Visual Analytics Database. In *ICDE*. 1466–1477.

[8] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2016. Neural Module Networks. In *CVPR*. 39–48.

[9] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732 [cs.PL]

[10] Jinze Bai, Shuai Bai, Shusheng Yang, Shijie Wang, Sinan Tan, Peng Wang, Junyang Lin, Chang Zhou, and Jingren Zhou. 2023. Qwen-VL: A Versatile Vision-Language Model for Understanding, Localization, Text Reading, and Beyond. arXiv:2308.12966 [cs.CV]

[11] Jaeho Bang, Gaurav Tarlok Kakkar, Pramod Chunduri, Subrata Mitra, and Joy Arulraj. 2023. SEIDEN: Revisiting Query Processing in Video Database Systems. *PVLDB* 16, 9 (2023), 2289–2301.

[12] Favyen Bastani, Songtao He, Arjun Balasingam, Karthik Gopalakrishnan, Mohammad Alizadeh, Hari Balakrishnan, Michael J. Cafarella, Tim Kraska, and Sam Madden. 2020. MIRIS: Fast Object Track Queries in Video. In *SIGMOD*. 1907–1921.

[13] Favyen Bastani, Oscar R. Moll, and Samuel Madden. 2020. Vaas: Video Analytics At Scale. *PVLDB* 13, 12 (2020), 2877–2880.

[14] Irving Biederman. 1987. Recognition-by-components: a theory of human image understanding. *Psychological review* 94, 2 (1987), 115.

[15] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *NeurIPS*.

[16] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. arXiv:2303.12712 [cs.CL]

[17] Cristian Bucila, Rich Caruana, and Alexandru Niculescu-Mizil. 2006. Model compression. In *SIGKDD*. 535–541.

[18] Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. 2024. Large Language Models as Tool Makers. In *ICLR*.

[19] Boyuan Chen, Zhuo Xu, Sean Kirmani, Brian Ichter, Dorsa Sadigh, Leonidas J. Guibas, and Fei Xia. 2024. SpatialVLM: Endowing Vision-Language Models with Spatial Reasoning Capabilities. In *CVPR*. 14455–14465.

[20] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code Generation with Generated Tests. In *ICLR*.

[21] Guobin Chen, Wongun Choi, Xiang Yu, Tony X. Han, and Manmohan Chandraker. 2017. Learning Efficient Object Detection Models with Knowledge Distillation. In *NeurIPS*. 742–751.

[22] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[23] Tianyu Chen, Shuai Lu, Shan Lu, Yeyun Gong, Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Hao Yu, Nan Duan, Peng Cheng, Fan Yang, Shuvendu K. Lahiri, Tao Xie, and Lidong Zhou. 2025. Automated Proof Generation for Rust Code via Self-Evolution. In *ICLR*.

[24] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching Large Language Models to Self-Debug. In *ICLR*.

[25] Yueting Chen, Nick Koudas, Xiaohui Yu, and Ziqiang Yu. 2022. Spatial and Temporal Constrained Ranked Retrieval over Videos. *PVLDB* 15, 11 (2022), 3226–3239.

[26] Zui Chen, Lei Cao, Sam Madden, Tim Kraska, Zeyuan Shang, Ju Fan, Nan Tang, Zihui Gu, Chunwei Liu, and Michael Cafarella. 2024. SEED: Domain-Specific Data Curation With Large Language Models. arXiv:2310.00749 [cs.DB]

[27] Zhenfang Chen, Rui Sun, Wenjun Liu, Yining Hong, and Chuang Gan. 2024. GENOME: Generative Neuro-Symbolic Visual Reasoning by Growing and Reusing Modules. In *ICLR*.

[28] Jaemin Cho, Yushi Hu, Jason M. Baldridge, Roopal Garg, Peter Anderson, Ranjay Krishna, Mohit Bansal, Jordi Pont-Tuset, and Su Wang. 2024. Davidsonian Scene Graph: Improving Reliability in Fine-grained Evaluation for Text-to-Image Generation. In *ICLR*.

[29] Rohan Choudhury, Koichiro Niinuma, Kris M. Kitani, and László A. Jeni. 2023. Zero-Shot Video Question Answering with Procedural Programs. *CoRR* abs/2312.00937 (2023).

[30] Pramod Chunduri, Jaeho Bang, Yao Lu, and Joy Arulraj. 2022. Zeus: Efficiently Localizing Actions in Videos using Reinforcement Learning. In *SIGMOD*. 545–558.

[31] Bo Dai, Yuqi Zhang, and Dahua Lin. 2017. Detecting Visual Relationships with Deep Relational Networks. In *CVPR*. 3298–3308.

[32] Maureen Daum, Enhao Zhang, Dong He, Magdalena Balazinska, Brandon Haynes, Ranjay Krishna, Apryle Craig, and Aaron Wirsing. 2022. VOCAL: Video Organization and Interactive Compositional AnaLytics. In *CIDR*.

[33] Maureen Daum, Enhao Zhang, Dong He, Stephen Mussmann, Brandon Haynes, Ranjay Krishna, and Magdalena Balazinska. 2023. VOCALExplore: Pay-as-You-Go Video Data Exploration and Model Building. *PVLDB* 16, 13 (2023), 4188–4201.

[34] Justin Dellinger, Carolyn Shores, Apryle Craig, Shannon Kachel, Michael Heithaus, William Ripple, and Aaron Wirsing. 2021. Predators reduce niche overlap between sympatric prey. *Oikos* (12 2021).

[35] Naihao Deng, Yulong Chen, and Yue Zhang. 2022. Recent Advances in Text-to-SQL: A Survey of What We Have and What We Expect. In *COLING*. 2166–2187.

[36] Iman Elghandour and Ashraf Aboulnaga. 2012. ReStore: Reusing Results of MapReduce Jobs. *PVLDB* 5, 6 (2012), 586–597.

[37] Qi Feng, Vitaly Ablavsky, and Stan Sclaroff. 2021. CityFlow-NL: Tracking and Retrieval of Vehicles at City Scale by Natural Language Descriptions. arXiv:2101.04741 [cs.CV]

[38] Reinhard Friedl, Helmut Höppler, Karl Ecard, Wilfried Scholz, Andreas Hannekum, Wolfgang Öchsner, and Sylvia Stracke. 2006. Multimedia-Driven Teaching Significantly Improves Students' Performance When Compared With a Print Medium. *The Annals of Thoracic Surgery* 81, 5 (2006), 1760–1766.

[39] Daniel Y. Fu, Will Crichton, James Hong, Xinwei Yao, Haotian Zhang, Anh Truong, Avanika Narayan, Maneesh Agrawala, Christopher Ré, and Kayvon Fatahalian. 2019. Rekall: Specifying Video Events using Compositions of Spatiotemporal Labels. arXiv:1910.02993 [cs.DB]

[40] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. *PVLDB* 17, 5 (2024), 1132–1145.

[41] Taicheng Guo, Kehan Guo, Bozhao Nan, Zhenwen Liang, Zhichun Guo, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. 2023. What can Large Language Models do in chemistry? A comprehensive benchmark on eight tasks. In *NeurIPS*.

[42] Tanmay Gupta and Aniruddha Kembhavi. 2023. Visual Programming: Compositional visual reasoning without training. In *CVPR*. 14953–14962.

[43] Patrick Hammer, Tony Lofthouse, Enzo Fenoglio, Hugo Latapie, and Pei Wang. 2020. A Reasoning Based Model for Anomaly Detection in the Smart City Domain. In *IntelliSys (AISC, Vol. 1251)*. 144–159.

[44] Dong He, Maureen Daum, Walter Cai, and Magdalena Balazinska. 2021. DeepEverest: Accelerating Declarative Top-K Queries for Deep Neural Network Interpretation. *PVLDB* 15, 1 (2021), 98–111.

[45] Dong He, Jieyu Zhang, Maureen Daum, Alexander Ratner, and Magdalena Balazinska. 2024. MaskSearch: Querying Image Masks at Scale. arXiv:2305.02375 [cs.DB]

[46] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick. 2017. Mask R-CNN. In *ICCV*. 2980–2988.

[47] Wenjia He, Michael R. Anderson, Maxwell Strome, and Michael J. Cafarella. 2020. A Method for Optimizing Opaque Filter Queries. In *SIGMOD*. 1257–1272.

[48] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the Knowledge in a Neural Network. arXiv:1503.02531 [stat.ML]

[49] Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alex Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. 2023. Distilling Step-by-Step! Outperforming Larger Language Models with Less Training Data and Smaller Model Sizes. In *Findings of ACL*. 8003–8017.

[50] Cheng-Yu Hsieh, Sibei Chen, Chun-Liang Li, Yasuhisa Fujii, Alexander J. Ratner, Chen-Yu Lee, Ranjay Krishna, and Tomas Pfister. 2023. Tool Documentation Enables Zero-Shot Tool-Usage with Large Language Models.

arXiv:2308.00675 [cs.CL]
[51] Bo Hu, Peizhen Guo, and Wenjun Hu. 2022. Video-zilla: An Indexing Layer for Large-Scale Video Analytics. In *SIGMOD*. 1905–1919.
[52] Yushi Hu, Otilia Stretcu, Chun-Ta Lu, Krishnamurthy Viswanathan, Kenji Hata, Enming Luo, Ranjay Krishna, and Ariel Fuxman. 2023. Visual Program Distillation: Distilling Tools and Programmatic Reasoning into Vision-Language Models. arXiv:2312.03052 [cs.CV]
[53] Jingwei Ji, Ranjay Krishna, Li Fei-Fei, and Juan Carlos Niebles. 2020. Action Genome: Actions As Compositions of Spatio-Temporal Scene Graphs. In *CVPR*. 10233–10244.
[54] Gaurav Tarlok Kakkar, Jiashen Cao, Aubhro Sengupta, Joy Arulraj, and Hyesoon Kim. 2024. Hydro: Adaptive Query Processing of ML Queries. arXiv:2403.14902 [cs.DB]
[55] Daniel Kang, Peter Bailis, and Matei Zaharia. 2019. BlazeIt: Optimizing Declarative Aggregation and Limit Queries for Neural Network-Based Video Analytics. *PVLDB* 13, 4 (2019), 533–546.
[56] Daniel Kang, John Guibas, Peter D. Bailis, Tatsunori Hashimoto, and Matei Zaharia. 2022. TASTI: Semantic Indexes for Machine Learning-based Queries over Unstructured Data. In *SIGMOD*. 1934–1947.
[57] Mohammad Reza Karimi, Nezihe Merve Gürel, Bojan Karlas, Johannes Rausch, Ce Zhang, and Andreas Krause. 2021. Online Active Model Selection for Pre-trained Classifiers. In *AISTATS*, Vol. 130. 307–315.
[58] George Katsogiannis-Meimarakis and Georgia Koutrika. 2023. A survey on deep learning approaches for text-to-SQL. *VLDB J.* 32, 4 (2023), 905–936.
[59] Moe Kayali, Anton Lykov, Ilias Fountalis, Nikolaos Vasiloglou, Dan Olteanu, and Dan Suciu. 2024. CHORUS: Foundation Models for Unified Data Discovery and Exploration. *PVLDB* 17, 8 (2024), 2104–2114.
[60] Chanwut Kittivorawong, Yongming Ge, Yousef Helal, and Alvin Cheung. 2024. Spatialyze: A Geospatial Video Analytics System with Spatial-Aware Optimizations. *PVLDB* 17, 9 (2024), 2136–2148.
[61] Ferdinand Kossmann, Ziniu Wu, Eugenie Lai, Nesime Tatbul, Lei Cao, Tim Kraska, and Sam Madden. 2023. Extract-Transform-Load for Video Streams. *PVLDB* 16, 9 (2023), 2302–2315.
[62] Ranjay Krishna, Vincent S. Chen, Paroma Varma, Michael S. Bernstein, Christopher Ré, and Li Fei-Fei. 2019. Scene Graph Prediction With Limited Labels. In *ICCV*. 2580–2590.
[63] Ranjay Krishna, Yuke Zhu, Oliver Groth, Justin Johnson, Kenji Hata, Joshua Kravitz, Stephanie Chen, Yannis Kalantidis, Li-Jia Li, David A. Shamma, Michael S. Bernstein, and Li Fei-Fei. 2017. Visual Genome: Connecting Language and Vision Using Crowdsourced Dense Image Annotations. *Int. J. Comput. Vis.* 123, 1 (2017), 32–73.
[64] Christopher A Kurby and Jeffrey M Zacks. 2008. Segmentation in the perception and memory of events. *TiCS* 12, 2 (2008), 72–79.
[65] Shuvendu K. Lahiri, Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, Madanlal Musuvathi, Piali Choudhury, Curtis von Veh, Jeevana Priya Inala, Chenglong Wang, and Jianfeng Gao. 2023. Interactive Code Generation via Test-Driven User-Intent Formalization. arXiv:2208.05950 [cs.SE]
[66] Huy Dinh-Anh Le, Quang Qui-Vinh Nguyen, Duc Trung Luu, Truc Thi-Thanh Chau, Nhat Minh Chung, and Synh Viet-Uyen Ha. 2023. Tracked-Vehicle Retrieval by Natural Language Descriptions with Multi-Contextual Adaptive Knowledge. In *CVPR Workshops*. 5511–5519.
[67] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chen-Chuan Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs. In *NeurIPS*.
[68] Junnan Li, Dongxu Li, Silvio Savarese, and Steven C. H. Hoi. 2023. BLIP-2: Bootstrapping Language-Image Pre-training with Frozen Image Encoders and Large Language Models. In *ICML*. 19730–19742.
[69] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. 2023. Visual Instruction Tuning. In *NeurIPS*.
[70] Xiaochen Liu, Pradipta Ghosh, Oytun Ulutan, B. S. Manjunath, Kevin S. Chan, and Ramesh Govindan. 2019. Caesar: cross-camera complex activity recognition. In *SenSys*. 232–244.
[71] Cewu Lu, Ranjay Krishna, Michael Bernstein, and Li Fei-Fei. 2016. Visual relationship detection with language priors. In *ECCV*. 852–869.
[72] Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. 2023. Chameleon: Plug-and-Play Compositional Reasoning with Large Language Models. In *NeurIPS*.
[73] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. 2018. Accelerating Machine Learning Inference with Probabilistic Predicates. In *SIGMOD*. 1493–1508.
[74] Zixian Ma, Jerry Hong, Mustafa Omer Gul, Mona Gandhi, Irena Gao, and Ranjay Krishna. 2023. CREPE: Can Vision-Language Foundation Models Reason Compositionally?. In *CVPR*. 10910–10921.
[75] Zixian Ma, Weikai Huang, Jieyu Zhang, Tanmay Gupta, and Ranjay Krishna. 2024. m &m's: A Benchmark to Evaluate Tool-Use for multi-step multi-modal Tasks. In *ECCV*, Vol. 15068. 18–34.

[76] Stephen Mell, Favyen Bastani, Steve Zdancewic, and Osbert Bastani. 2023. Synthesizing Trajectory Queries from Examples. In *CAV*, Vol. 13964. 459–484.
[77] Stephen Mell, Steve Zdancewic, and Osbert Bastani. 2024. Optimal Program Synthesis via Abstract Interpretation. *Proc. ACM Program. Lang.* 8, POPL (2024), 457–481.
[78] Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi Ramamritham. 2001. Materialized View Selection and Maintenance Using Multi-Query Optimization. In *SIGMOD*. 307–318.
[79] Chancharik Mitra, Brandon Huang, Trevor Darrell, and Roei Herzig. 2024. Compositional Chain-of-Thought Prompting for Large Multimodal Models. In *CVPR*. 14420–14431.
[80] Oscar Moll, Favyen Bastani, Sam Madden, Michael Stonebraker, Vijay N. Gadepally, and Tim Kraska. 2020. ExSample: Efficient Searches on Video Repositories through Adaptive Sampling. *ICDE* (2020), 3065–3077.
[81] Ravi Teja Mullapudi, Fait Poms, William R. Mark, Deva Ramanan, and Kayvon Fatahalian. 2021. Learning Rare Category Classifiers on a Tight Labeling Budget. In *ICCV*. 8403–8412.
[82] Rock Yuren Pang, Sebastin Santy, René Just, and Katharina Reinecke. 2024. BLIP: Facilitating the Exploration of Undesirable Consequences of Digital Technologies. In *CHI*. 290:1–290:18.
[83] Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction. In *NeurIPS*.
[84] Wei Qiao, Tushar Dogra, Otilia Stretcu, Yu-Han Lyu, Tiantian Fang, Dongjin Kwon, Chun-Ta Lu, Enming Luo, Yuan Wang, Chih-Chun Chia, Ariel Fuxman, Fangzhou Wang, Ranjay Krishna, and Mehmet Tek. 2024. Scaling Up LLM Reviews for Google Ads Content Moderation. In *WSDM*. 1174–1175.
[85] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. In *ICLR*.
[86] Mark Raasveldt and Hannes Muehleisen. [n. d.]. *DuckDB*. https://github.com/duckdb/duckdb
[87] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning Transferable Visual Models From Natural Language Supervision. In *ICML*. 8748–8763.
[88] Francisco Romero, Caleb Winston, Johann Hauswald, Matei Zaharia, and Christos Kozyrakis. 2023. Zelda: Video Analytics using Vision-Language Models. arXiv:2305.03785 [cs.DB]
[89] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. 2000. Efficient and Extensible Algorithms for Multi Query Optimization. In *SIGMOD*. 249–260.
[90] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]
[91] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. In *NeurIPS*.
[92] Kyra Schlining, Susan von Thun, Linda Kuhnz, Brian Schlining, Lonny Lundsten, Nancy Jacobsen Stout, Lori Chaney, and Judith Connor. 2013. Debris in the deep: Using a 22-year video annotation database to survey marine litter in Monterey Canyon, central California, USA. *Deep Sea Research Part I: Oceanographic Research Papers* 79 (2013), 96–105.
[93] Ashish Sharma, Inna W. Lin, Adam S. Miner, David C. Atkins, and Tim Althoff. 2021. Towards Facilitating Empathic Conversations in Online Mental Health Support: A Reinforcement Learning Approach. In *WWW*. 194–205.
[94] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Face. In *NeurIPS*.
[95] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *NeurIPS*.
[96] Otilia Stretcu, Edward Vendrow, Kenji Hata, Krishnamurthy Viswanathan, Vittorio Ferrari, Sasan Tavakkol, Wenlei Zhou, Aditya Avinash, Enming Luo, Neil Gordon Alldrin, MohammadHossein Bateni, Gabriel Berger, Andrew Bunner, Chun-Ta Lu, Javier A Rey, Giulia DeSalvo, Ranjay Krishna, and Ariel Fuxman. 2023. Agile Modeling: From Concept to Classifier in Minutes. In *ICCV*. 22266–22277.
[97] Dídac Surís, Sachit Menon, and Carl Vondrick. 2023. ViperGPT: Visual Inference via Python Execution for Reasoning. In *ICCV*. 11854–11864.
[98] Imad Eddine Toubal, Aditya Avinash, Neil Gordon Alldrin, Jan Dlabal, Wenlei Zhou, Enming Luo, Otilia Stretcu, Hao Xiong, Chun-Ta Lu, Howard Zhou, Ranjay Krishna, Ariel Fuxman, and Tom Duerig. 2024. Modeling Collaborator: Enabling Subjective Vision Classification with Minimal Human Effort via LLM Tool-Use. In *CVPR*. 17553–17563.

[99] Zhongwei Wan, Xin Wang, Che Liu, Samiul Alam, Yu Zheng, Jiachen Liu, Zhongnan Qu, Shen Yan, Yi Zhu, Quanlu Zhang, Mosharaf Chowdhury, and Mi Zhang. 2024. Efficient Large Language Models: A Survey. *Trans. Mach. Learn. Res.* 2024 (2024).
[100] Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A. Saurous, and Yoon Kim. 2023. Grammar Prompting for Domain-Specific Language Generation with Large Language Models. In *NeurIPS*.
[101] Huichen Will Wang, Larry Birnbaum, and Vidya Setlur. 2025. Jupybara: Operationalizing a Design Space for Actionable Data Analysis and Storytelling with LLMs. arXiv:2501.16661 [cs.HC]
[102] Huichen Will Wang, Mitchell Gordon, Leilani Battle, and Jeffrey Heer. 2025. DracoGPT: Extracting Visualization Design Preferences from Large Language Models. *IEEE Trans. Vis. Comput. Graph.* 31, 1 (2025), 710–720.
[103] Jiayu Wang, Yifei Ming, Zhenmei Shi, Vibhav Vineet, Xin Wang, Sharon Li, and Neel Joshi. 2024. Is A Picture Worth A Thousand Words? Delving Into Spatial Reasoning for Vision Language Models. In *NeurIPS*.
[104] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *ICLR*.
[105] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2023. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. arXiv:2308.08155 [cs.AI]
[106] Ioannis Xarchakos and Nick Koudas. 2019. SVQ: Streaming Video Queries. In *SIGMOD*. 2013–2016.
[107] Zhuangdi Xu, Gaurav Tarlok Kakkar, Joy Arulraj, and Umakishore Ramachandran. 2022. EVA: A Symbolic Approach to Accelerating Exploratory Video Analytics with Materialized Views. In *SIGMOD*. 602–616.
[108] Piyush Yadav and Edward Curry. 2019. VidCEP: Complex Event Processing Framework to Detect Spatiotemporal Patterns in Video Streams. In *Big Data*. 2513–2522.
[109] Junran Yang, Alex Bäuerle, Dominik Moritz, and Çagatay Demiralp. 2023. VegaProf: Profiling Vega Visualizations. In *UIST*. 94:1–94:11.
[110] Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. 2023. SatLM: Satisfiability-Aided Language Models Using Declarative Prompting. In *NeurIPS*.
[111] Kexin Yi, Chuang Gan, Yunzhu Li, Pushmeet Kohli, Jiajun Wu, Antonio Torralba, and Joshua B. Tenenbaum. 2020. CLEVRER: Collision Events for Video Representation and Reasoning. In *ICLR*.
[112] Shoubin Yu, Jaemin Cho, Prateek Yadav, and Mohit Bansal. 2023. Self-Chained Image-Language Model for Video Localization and Question Answering. In *NeurIPS*.
[113] Shan Yu, Zhenting Zhu, Yu Chen, Hanchen Xu, Pengzhan Zhao, Yang Wang, Arthi Padmanabhan, Hugo Latapie, and Harry Xu. 2024. VQPy: An Object-Oriented Approach to Modern Video Analytics. In *MLSys*.
[114] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *EMNLP*. 3911–3921.
[115] Jeffrey M Zacks, Barbara Tversky, and Gowri Iyer. 2001. Perceiving, remembering, and communicating structure in events. *Journal of experimental psychology: General* 130, 1 (2001), 29.
[116] Rowan Zellers, Mark Yatskar, Sam Thomson, and Yejin Choi. 2018. Neural Motifs: Scene Graph Parsing With Global Context. In *CVPR*. 5831–5840.
[117] Enhao Zhang, Maureen Daum, Dong He, Manasi Ganti, Brandon Haynes, Ranjay Krishna, and Magdalena Balazinska. 2023. EQUI-VOCAL Demonstration: Synthesizing Video Queries from User Interactions. *PVLDB* 16, 12 (2023), 3978–3981.
[118] Enhao Zhang, Maureen Daum, Dong He, Brandon Haynes, Ranjay Krishna, and Magdalena Balazinska. 2023. EQUI-VOCAL: Synthesizing Queries for Compositional Video Events from Limited User Interactions. *PVLDB* 16, 11 (2023), 2714–2727.
[119] Enhao Zhang, Nicole Sullivan, Brandon Haynes, Ranjay Krishna, and Magdalena Balazinska. 2025. Self-Enhancing Video Data Management System for Compositional Events with Large Language Models [Technical Report]. arXiv:2408.02243 [cs.DB]
[120] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. arXiv:1709.00103 [cs.CL]
[121] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. WebArena: A Realistic Web Environment for Building Autonomous Agents. In *ICLR*.

Received October 2024; revised January 2025; accepted February 2025