ChatCache: A Hierarchical Semantic Redundancy Cache System for Conversational Services at Edge

Lanyu Xu
Wayne State University
xu.lanyu@wayne.edu

Arun Iyengar
Intelligent Data Management and Analytics, LLC
aki@akiyengar.com

Weisong Shi
Wayne State University
weisong@wayne.edu

***Abstract*—The spatial-temporal locality has been observed in various scenarios for conversational services with either voice or text requests. Given the current cloud-based processing mechanism, integrating such a service with caching is a promising way to improve responsiveness, reduce in-network transmission, and avoid computational redundancy. Goes beyond precise redundancy and fuzzy redundancy, semantic redundancy adapts to the diversity in command expression, and is considered as a practical solution for conversational services. In this paper, we introduce a hierarchical cache design inspired by semantic redundancy for conversational services. We propose a scalable edge system *ChatCache* to incorporate the hierarchical cache design and serve single or multiple users. We discussed the cache efficiency with different similarity match policies, and evaluate the responsiveness and scalability of *ChatCache* on heterogeneous edge platforms. On most of the evaluated platforms, *ChatCache* reduces user-perceived latency by more than 91.7% for voice requests, more than 81.6% for text requests. The throughput of *ChatCache* reaches 42.6 throughput tps for voice requests, and 64.4 tps for text requests, which is comparable with mainstream cloud cognitive services. The promising evaluation results show the capability of *ChatCache* in reducing the user-perceived latency and computation redundancy with high response accuracy for conversational services.**

***Index Terms*—Cache, Semantic Similarity, Conversational Service, Edge Computing, Model Compression**

I. Introduction

Natural language has become the emerging way people interact with the machine. Conversational services that are based on the natural language have been introduced into various fields to provide an intelligent, professional, and less labor-involved experience for users. When integrating a conversational user interface into the products (e.g., a mobile/web app, a web application, an interactive voice response system), it is common for developers to use cloud resources due to the computation-intensive requirements. Although it provides sufficient accuracy supported by powerful computation resources, the cloud-based conversational services mechanism [1]–[5] brings low efficiency as all requests have to be processed after sending to the cloud. It also affects the reliability of the performance caused by an unstable Internet connection.

There have been prior efforts that are mainly divided into two directions to address the unstable and low computational efficient problem brought from cloud-based cognitive services. One is offloading to the edge; another is integrating with the cache. Offloading to the edge relies on efficient neural network design to enable intelligence on the resource-constraint edge devices [6]–[8]. Another is integrating with the cache. Cache reuses previous request results to eliminate the computational redundancy. It observes the precise redundancy and fuzzy redundancy in cognitive services to reduces the processing time by a proximate searching. Precise redundancy finds the exact match from previously processed requests for the search item to reuse the computation results. Fuzzy redundancy measures the similarity between search item and cached items and determines the threshold to reuse the computation results. With this idea, many efforts have been devoted to image processing and achieved good performance [9]–[11], as the similarity of the two images is intuitive to be captured.

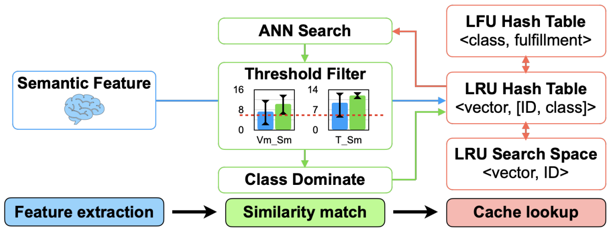


Fig. 1: The designed workflow of *ChatCache*.

The cache in conversational services, however, is not well leveraged for computation efficiency with precise or fuzzy redundancy. In *ChatCache*, we innovatively capture the *semantic redundancy* and design a unique and efficient cache system (Figure 1) with it. We observe that in practical conversational services, people with diverse backgrounds have different means of expression using various vocabulary or phrases (syntactic different) when describing the same meaning (semantic duplicated). Statistics reveal that even for a single person, he or she likes changing the syntax of the semantic command in different circumstances [12]. To effectively capture the redundancy in conversational service requests, the problem is identifying the semantic duplication. For example, *"what is the waiting time for rides?"* and *"Rides' waiting time?"* are both syntactically and semantically duplicated. The request *"How long I need to wait for rides?"* is syntactically different from the former two, while semantically, they have the same meaning. In this case, the precise redundancy is barely functional, as it requires the exact match of the input content. The fuzzy redundancy can eliminate the computation redundancy for syntactic duplication by computing the word similarity. For example, term frequency is a traditional way to quantify the similarity of two sentences [13]. Two sentences are close to each other in the vector space if the composed words' frequency is similar. However, it focuses on the word similarity while omits the semantic meaning. Above mentioned three semantic duplicated statements belong to the blue bar in Figure 2(b), while the redundancy measurements have a considerable overlap with the green bar (statements with different semantic meanings), resulting in a high false-positive rate. For voice requests, the situation is more complex as syntactic duplication is getting harder to be captured, not to mention semantic duplication. Usually, Mel-Frequency Cepstral Coefficients (MFCC) captures the voice feature, and fuzzy redundancy compares the distance of vectors [10]. However, this mechanism only works when the same statement is delivered by the same voice ($V_s\_S_s$ in Figure 2(a)). If the same command is issued by multiple speakers ($V_m\_S_s$ in Figure 2(a)), it is hard to distinguish whether fuzzy redundancy exists. Never to mention when the two commands are only semantically duplicated. For example, two voice commands: "louder, please" and "turn sound up," both are requesting the smart speaker increases the volume, while the acoustic feature-based fuzzy redundancy cannot capture this duplication ($V_m\_S_m$ in Figure 2(a)).

ChatCache is distinguished from previous work on two aspects. First, as a caching system, it innovatively captures the semantic redundancy by introducing an intelligent feature extraction module. Second, it is appliable for both voice and text requests, while previous work cannot efficiently cover both situations. Figure 1 presents the basic workflow of *ChatCache*. It maintains a hierarchical cache design, consisting of one least recent used (LRU) searching space S, one LRU hash table H_1, and one least frequently used (LFU) hash table H_2. S, H_1, H_2 are synchronized with read-with-write consistency. *ChatCache* takes a modularized design for main functions: feature extraction, similarity match, and cache lookup. For each request, *ChatCache* first evokes an intelligent model to extract its semantic feature. With the extracted feature *ChatCache* then looks up H_1 for an exact match and determines to start a similarity match if no exact match is found. To search quickly and accurately, we propose t-HNSW built on top of hierarchical navigable small world (HNSW) [14] to accelerate the indexing and searching to find the dominant semantic class in approximate k nearest neighbors (ANN). If the dominant class is determined, *ChatCache* looks up H_2 to obtain the class's fulfillment, and updates S, H_1 to synchronize the hierarchical cache. If no dominant class is found, *ChatCache* will forward the request to the computation unit for processing. Leverages the spatial-temporal locality, *ChatCache* is flexible to serve a single client or multiple clients to share the computation results with a well-designed caching system. We compress the intelligent model with knowledge distillation technology for efficient system implementation and enable running on resource-constrained edge platforms.

We summarize the contribution of this paper as follows.

- We analyze the semantic distance of requests in conversational services and reveal the drawback of using fuzzy redundancy in such cache systems. Instead, we find semantic redundancy is a potential solution for an efficient cache design.
- We propose *ChatCache*, a unique cache design to capture semantic redundancy for conversational services and modularize its functionalities with the general usage purpose for both voice and text request scenarios.
- We deploy *ChatCache* on heterogenous edge platforms and demonstrate its promising performances. With a flexible design to different usage scenarios, *ChatCache* can reduce user-perceived latency by at least 81.6% and 91.7% for text and voice requests, respectively, with a high precision rate and high throughput.

The remainder of this paper is organized as follows. Section II discusses the semantic redundancy in conversational services. Section III explains the semantic reuse approach in *ChatCache* for. Section IV introduces the design of *ChatCache*, followed by the implementation in Section V. Section VI evaluates the performance of *ChatCache*. Section VII reviews the related work. The paper concludes in Section VIII.

II. MOTIVATION

A. Status-quo and motivating scenarios

The workflow of a typical cloud cognitive service has three steps: (a) the client transmits the voice command transmitted to the cloud, (b) the cloud-based intelligent model recognizes and parses the received voice command, (c) the service executes the cloud fulfillments and sends the response to the client. This integrated workflow design limits the flexibility and reliability of the cloud-based cognitive service, especially when the cloud or the network is unstable or unreachable. Moreover, due to the spatial-temporal locality widely in the cognitive service, it is inefficient for the cloud to compute all these requests.

There are some typical scenarios where the spatial-temporal locality exists. One example is the smart home, where the smart speaker is used to control the home automation. Statistics [12] show that the voice commands are short in length, limit in topics, active in a certain period. We can design a caching system in the smart home to provide more efficient, stable service with privacy to users. The second example is the self-driving cars, where voice is the primary interaction interface between the driver and the vehicle. When the driver requests the nearby traffic information from the navigation system, instead of uploading and parsing all the requests to the cloud for the entire processing, the road site unit can maintain a caching system, and share the traffic information across vehicles, improve the computation efficiency. Such a caching system can be leveraged for many types of voice assistant services as well.

There are three main challenges to designing an edge-based caching system for conversational services. First, conversational service is computation-intensive. It is hard for the edge nodes to process the powerful off-the-shelf model directly, as the edge infrastructures are usually less powerful than the cloud. Second, the semantic features (introduce in Section II-C) extracted from the requests are high-dimensional vectors, how to store and correctly classify these vectors is another concern. With the high-dimensional vectors and different semantic classes, the third challenge is how to design an efficient cache lookup solution. To address these challenges, we discuss the light-weighted model design and dominant class selection in Section III, and introduce the hierarchical cache design in Section IV.

B. Redundancy in cognitive services

Precise redundancy and fuzzy redundancy have been discussed in the caching system for different services, such as web services and cognitive services. Precise redundancy finds the exact match from previously processed requests to reuse the computation results. One example is function cache [15], which caches the function results for the same input arguments to reuse. Precise redundancy functions well when the two language commands are the same.

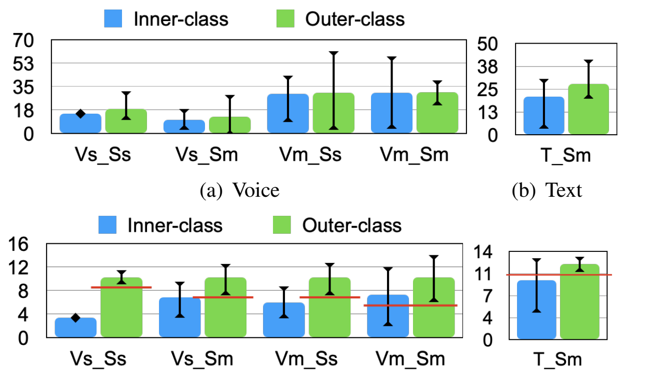
Given two similar inputs in cognitive service, fuzzy redundancy considers the input similarity and determines the threshold to reuse the previously computation results. In practical, fuzzy redundancy is more reasonable, especially in cognitive-related services, such as recommendation system [16], [17] and machine learning training and inference [18], [19]. The similarity observed by fuzzy redundancy focuses on the superficial about the contextual similarity of the inputs. For example, suppose the two pictures contain the same object but shoot from different views, or the two voice commands are issuing the same command. In that case, they can be represented with corresponding features with a general solution. A distance function computes the similarity between the input features, and previous results will be reused if fuzzy redundancy exists. However, generic feature representations, such as SIFT for images [20], word2vec for text, and MFCC for audio [21], are able to extract the fuzzy similarity while failing to capture the semantic redundancy.

In the above three scenarios, model inference will be computed repeatedly when receiving the input language data with semantic redundancy. Semantic redundancy is determined by the intrinsic of language. Exposed to diverse backgrounds, users have different expressions, vocabulary, and grammar when describing the same intention.

C. Feature Distance in Redundancy

To quantitatively present the difference between fuzzy redundancy and semantic redundancy, we compute the feature distance. First, we extract features from the text and audio data to map the unstructured data to the vector space. For fuzzy redundancy, we use CounterVector [22] on text, *MFCC* on audio. For semantic redundancy, we take the output from the first layer of deep learning models. For text, we use a pre-trained DistilBERT model [23]; for audio, we use a spoken language understanding (SLU) model [24]. Then, we compute the similarity for extracted features with L2 distance. The reason we use L2 distance is that it captures the magnitude information of features in the vector space. In contrast, cosine distance, another widely used method, only computes the angle between vectors.

Figure 2 considers five scenarios based on the dataset *Fluent Speech Commands* (*FSC* in Section VI): (a) $V_s\_S_s$: the same voice (*i.e.,* same speaker) with the same statement in each semantic class, (b) $V_s\_S_m$: the same voice with multiple statements, (c) $V_m\_S_s$: multiple voices with the same statement in each semantic class, (d) $V_m\_S_m$: multiple voices with multiple statements, and (e) T_S_m: multiple statements in the text. The same statement only applies to the same semantic class. The blue bar shows the average distance of statements within each semantic class; the green bar shows the average distance between different classes. Error bar marks the redundancy measurements range in each scenario. In fuzzy redundancy (Figure 2(a), 2(b)), the difference between feature distances of inner- and outer-classes are not distinguished well, especially for voice data input. MFCC captures the acoustic features from voice data; thus, the $V_s\_S_s$ (blue bar) shows no variance. However, when different speakers say the same statement, MFCC fails to capture the similarity well (in scenario $V_m\_S_s$). In more practical scenarios, when the same speaker or different speakers express the same semantic meaning with different statements (scenario $V_s\_S_m$, $V_m\_S_m$), fuzzy redundancy cannot distinguish either. The reason behind this is, different speakers have different voiceprints, and the same speaker can have different statements to issue the same intent. As a result, fuzzy redundancy turns out to be restricted for natural-language-based cognitive services in practice. In contrast, semantic redundancy (Figure 2(c), 2(d)) captures semantic features and distinguishes semantic classes well. The minimum inter-class distance is notably higher than the minimum, or even the average intra-class distance. In other words, we can always find a distance value to differentiate between semantic classes in different scenarios. Therefore for natural-language-based cognitive services, especially those taking voice command as input, semantic redundancy, rather than fuzzy redundancy, is a more practical solution when designing the caching system for computation reuse purposes.



(a) Voice (b) Text

(c) Voice (d) Text

Fig. 2: The performance of distinguishing statements with fuzzy redundancy ((a)(b)), and semantic redundancy ((c)(d)). The similarity of statements is quantized as the L2 distance by each approach. Voice V, text T, statement S, same expression s, multiple expressions m. Inner class means the statements are semantically duplicate. Outer class means the statements are semantically different. The red line on (c)(d) shows that semantic redundancy can distinguish statements.

TABLE I: Time and space cost of understanding models.

	Model size (MB)	Inference (ms)	Cloud (ms)
SLU	15.9	737.0	1681.5
NLU	438.1	34.9	206.0
Distilled NLU	123.8	17.9	

III. Semantic Similarity

A. *Semantic feature extraction*

Figure 2 has shown the results when extracting features in semantic redundancy. Different from fuzzy redundancy, where the traditional feature extraction methods work, semantic redundancy requires the model to understand the meaning of the input data and capture the semantic feature. For voice data, the understanding model for spoken language has two ways. Conventionally, a two-step mechanism is adopted: automatic speech recognition (ASR) first recognizes audio to text, natural language understanding (NLU) then extracts the feature to compute the similarity. Another mechanism is more intuitive: an end-to-end SLU model extracts the semantic feature from audio. With either method, the qualified model should eliminate the impact of different speakers and capture the semantic information. Similarly, for text data, a qualified NLU model captures the semantic information.

To deploy the understanding model on resource-constrained edge devices, storage and computation cost are a big concern. Besides, at the critical path, semantic feature extraction requires a short response time. Table I lists the time and space cost for running the SLU and NLU models discussed in Section II. Both of the models run with an Intel Fog Reference Design (Table III). As a benchmark, we evaluate the same understanding process's responsiveness when the request is sent to the cloud. The comparison shows that deploying an understanding model for either voice or text request is computationally expensive, and brings significant overhead if the computation reuse is not applied.

To efficiently deploy *ChatCache*, we compress the understanding model. There have been many different model compression technologies, such as pruning [25], quantization [26], knowledge distillation [27], *etc.*. From Table I we observe that the SLU model is small in model size while complex in model inference, revealing its compact and informative network structure. Unlike pruning or quantization, knowledge distillation is not restricted by the model structure or the system architecture [28]. It also requires low training efforts comparing to other compression methods [29]. It imitates the teacher and students' roles by distilling the pre-trained huge model (i.e., the teacher model) to transfer knowledge to train a model with a much smaller size (i.e., the student model). During the training process, a distillation temperature T controls the probability distribution's smoothness over classes. Section V discusses the compression efficiency of the compact and informative SLU model with knowledge distillation.

B. *Semantic similarity search algorithm*

Given the semantic duplicated features are significantly closer to each other in the vector space, *ChatCache* conceives semantic reuse happens when the request has neighbors with enough close distance. Similar searching can be quite expensive regarding feature encoding and searching [30]. To have a scalable and efficient search, k-nearest neighbor (k-NN) searching is a widely used approach. By defining a distance function, k-NN finds the k elements with the shortest distance to the given object. The semantic features are high in dimension and will result in k-NN ends in the curse of dimensionality. To effectively address the curse of dimensionality problem with a tradeoff between accuracy and speed, HNSW [14] has been proposed. But it can not be used for semantic redundancy directly because the returned ANN items contains one or more classes. To identify the dominant class for the search key given ANN, we propose t-HNSW (Algorithm 1) to (1) produce a weighted distance of each class, and (2) find the dominant class.

t-HNSW builds the search index space with HNSW, which combines the idea of the graph and skip-table and stores the indexes with a layered architecture. The created index space can be serialized to store in the system and deserialize for a warm start. For the search key, if there is no exact match found in H_1, t-HNSW first added the key to the index space, and start searching from the bottom level to find the K-NN for the search key. The returned K NN reflects the number of items close to the search key. To further identify the neighbors with close semantic meaning, t-HNSW filters K' candidate from K with a distance threshold D. D represents the distance that can distinguish similar from dissimilar with high confidence. We set the value of D from the empirical study and discussed it in Section VI.

Given the filtered k items belong to C classes (Algorithm 1 line 1), the goal of t-HNSW is to find the correct class that the search key belongs to. t-HNSW classifies K' items by semantic classes to form two sets: one is the distance set A, recording the distance between the search key and each neighbor in class c_i; another is the counting set B, recording the number of items in each class (line 2-3). t-HNSW computes the weighted distance W between the search key to each class (line 4). $\overline{\sum A_i}$ is the sum average of items in A_i. It is added by 1 to avoid invalid before the logarithmic operation. The logarithm magnifies the marginal effect. Therefore, the closer distance class weighted more. Finally, line 5 identifies the dominant class that has the shortest distance. In a high dimension vector space, inspired by cosine similarity, we convert the comparison of distance to the comparison of intersection angles between the search key to classes. Therefore, the class has the shortest distance is the one has the maximum cosine value with the search key.

IV. *ChatCache* Design

To reuse the semantic redundancy and accelerate the request handling for conversational services, *ChatCache* performs as the middleware locating at the critical path between the end-user and the cloud service. It is adaptive to different end-user scales: it can be deployed for a single user (*e.g.,* the smart speaker in a smart home); or be shared by multiple users (*e.g.,* a roadside unit communicating with moving vehicles). It

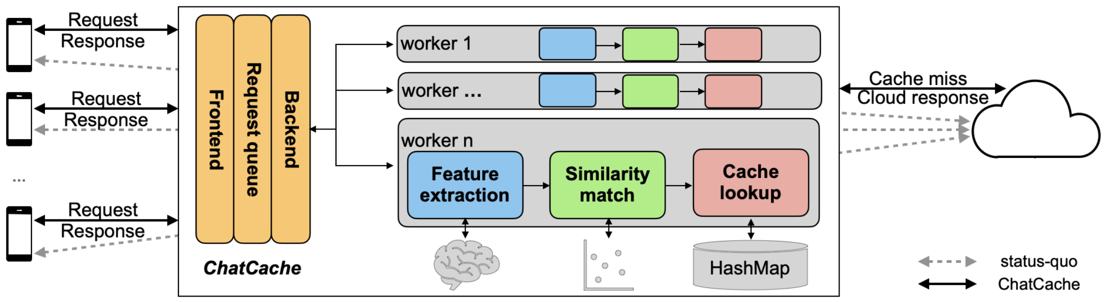


Fig. 3: The design of *ChatCache*.

Algorithm 1 The workflow of t-HNSW

```
1: The semantic class set $C = \{c_i|i = 1, ..., m\}$. For $k \in K'$ has class $c_k$, $d_k$ distance to the search key.
2: Define a distant set $A_i = \{d_k|c_k = c_i\}, k \in K'$.
3: Define a counting set $B = \{b_i|i = 1, ..., m\}$, where $b_i = |\{k|c_k = c_i\}|$.
4: Compute the weighted distance for each class $w_i = \frac{b_i}{\log(\sum A_i + 1)}$, where $b_i \in B$.
5: Find dominant class max $\frac{w_i}{||W||^2}$, where $w_i \in W$.
```

is worth mentioning that we propose *ChatCache* as a general approach regardless of where the cache resides. In other words, the semantic similarity techniques are of value whether the cache resides on the client, server, or edge within the network. In the status-quo approach, requests are sent directly to the cloud to understand the meaning behind the audio, followed by the designed fulfillment, and then the processed response is sent back. For example, the *turn light on* command receives the device control syntax to trigger the connected light off through the control layer; the *tell me the history of this sculpture* command receives the detailed explanation that is prepared for the sculpture. The current cloud service providers process all the audio or text commands by first uploading them to the cloud. In other words, no on-device understanding is supported except the hot-word detection to trigger the cloud understanding service. The design of *ChatCache* is aimed to reduce the transmission path for requests and response, with high accuracy and reusability.

A. Overview

Instead of sent to the cloud directly, commands in *ChatCache* are routing to the edge server first. The edge server is a modularized design containing *feature extraction*, *semantic match* and *cache lookup*. To adapt to different users' scales, we use *ZeroMQ* [31] as the message protocol, and takes the router/dealer pattern. The *frontend* is a router that receives the request from clients, and stores them in a *request queue*. The *backend* is a dealer that assigns requests to launched *workers*. For scalability purposes, sending requests and receiving responses are designed asynchronously. A unique ID is prepended on each request to distinguish them. As an N to 1 schema, *workers* in *ChatCache* running at different threads to execute the core three modularized functions. *Backend* receives the processed response and sends them back to the clients based on the assigned ID. For each request received by a *worker*, it will first be stripped from the ID for processing. As the semantic duplication reuse described in Section III, a compressed model extracts semantic feature and generates a feature vector that mapping to a high-dimension space. t-HNSW then searches semantically similar items in an efficient and approximate way, followed by a weighted ranking to filter the dominant class with a predefined threshold. A cache hit request will receive the cached value as the response. A cache miss request will invoke the cloud understanding processing as the status quo approach, and the cloud response will be used to update the cache and prepared to forward to the *backend*. In the end, the *worker* prepends the ID hold for the request and sends the respond back to the *backend*.

B. Hierarchical Cache Design

ChatCache is designed with a hierarchical cache on edge, as shown in Figure 1: an LRU search space S, an LRU cache H_1 and an LFU cache H_2. S is the searching space for t-HNSW to find the nearest neighbors. It stores each item with a unique universal ID. For each search key, S returns its ANNs in the pair of <ID, distance>, where distance is the L2 distance between the selected item and the search key. H_1 is an extension of S in the form of the hash table. It stores <vector, [ID, class, expire]> for each item, where the semantic class and expire time are recorded. H_2 differs from H_1 and S in the design purpose and eviction policy. H_2 stores <class, [fulfillment, vectors, expire]> in a hash table. Fulfillment records the response to this semantic class from previous cloud processing. The vectors (requested items) have the same semantic meaning map to the same class. For each request, after the feature extraction, *ChatCache* first looks for an exact match in H_1. If there is no exact match, the system evokes t-HNSW to start the similarity match.

Now we explain how *ChatCache* maintains the consistency of the hierarchical cache in different operations.

Add For each request, *ChatCache* first converts it to the feature vector. If no exact match is found in H_1, *ChatCache* will add it to the searching space as the search key for similarity matching. When the request hits the cache (a dominant class is found) with class C_i, *ChatCache* adds the search key to H_1 as a pair of <vector, [ID, C_i, expire]>. Synchronously, H_2 updates C_i item by adding the search key ID to the value. The frequency of C_i is updated as well for the eviction policy. If no dominant class is found, the system forwards the request to the cloud for processing and updates the H_1, H_2 after receiving the cloud response. Under both circumstances, the system returns the fulfillment of the request to the end-user. Besides, if the matching item in H_1, H_2 is expired, the system will update the item by forwarding the request to the cloud.

Update *ChatCache* designs a timestamp for each item in H to record the expiration time. When the request $ID_k \in C_i$ is expired in H_1, it is marked as a dirty item and ready for an update. If an exact match hits this item, *ChatCache* calls the cloud understanding service to process the command and update the response for C_i in H_2. Then both the expire timestamp in H_1 for ID_k and H_2 for C_i are updated. Similarly, when the item C_i is expired in H_2, it is marked as a dirty item and update when a new search key hits the cache with the dominant class C_i. The update of C_i in H_2 will update the expire timestamp not only for itself, but the corresponding IDs in H_1. The update in either H_1 or H_2 will not affect S, as the feature vector is not changed.

Delete Designing hierarchical cache with different cache keys, *ChatCache* adopts two eviction policies. H_1 uses vector as the key, it adopts the LRU eviction policy. S is a proportion of H_1, and adopts the same eviction policy to keep consistency. In other words, the item that has been evicted by H_1 is labeled as deleted in S. Items marked as deleted will not be visited later in the item search in S. H_2 adopts the LFU eviction policy. H_2 maintains a timestamp and an importance score (IS_i). The importance score IS_i is represented by the requested class's frequency. Evicting an item from H_2 is the most costly operation, as the items $id_k \in C_i$ are required to be evicted from H_1 one by one, and marked as deleted in S.

The time cost of *ChatCache* involves creation and item searching. When inserting one element e, similar to skip list, S takes $\mathcal{O}(\log n)$ to identify the number of layers to insert e and find e's neighbors. In system creation, for n elements, S takes $\mathcal{O}(n \log n)$ in index building. H_1, H_2 both take $\mathcal{O}(n)$ in warmup initialization. The item searching in S is $\mathcal{O}(\log n)$, in H_1, H_2 are $\mathcal{O}(1)$. Therefore, the time complexity of *ChatCache* creation and searching is $\mathcal{O}(n \log n)$, $\mathcal{O}(\log n)$.

C. Scalable Support for Clients

ChatCache is designed adaptive to different scales of clients and deployment locations. It can be deployed on the device to serve a single client. The cache is stored on the device, and the data transmission route is unchanged from the device to the cloud service. One thread of *worker* is enough for *ChatCache* to process requests for a single user. For multiple clients where the cache is sharing among them, *ChatCache* provides *frontend* and *backend* with asynchronous interface to handle multiple requests. Multiple threads are launched as *workers* to support scalable processing. Searching space S and cache H_1, H_2 are global parameters shared by *workers*, and each *worker* maintains their own semantic extraction model.

ChatCache is a general solution for conversational services for input commands with either text or audio format. The semantic matching algorithm t-HNSW works for different usage scenarios with different types of input. *ChatCache* handles different end-user scales and can be deployed on heterogeneous hardware platforms, from resource-constrained embedded systems to powerful CPU- or GPU-equipped edge machines. The generality of *ChatCache* is discussed in Section VI in terms of the usage scenarios, input format, client scalability, hardware configuration. Besides, *ChatCache* is a promising solution to serve different intelligence services (*e.g.,* computer vision, virtual reality) where semantic redundancy can be leveraged. The modularized design makes it easy to replace the feature extraction model with other models designed with different target scenarios.

V. Implementation

A. ChatCache Framework

We implement *ChatCache* with Python 3, and the intelligent part works in PyTorch 1.4.0. The hierarchical cache system is implemented as an in-process cache. We design an LRU cache h_1 with an OrderedDict in Python, where the LRU eviction policy is supported by order of items. With each put/get operation, the order of items updates by relocating the most recent visited item to the front. When the cache is full of capacity, the item in the tail is evicted. We design an LFU cache h_2 with a dictionary to save the <key, value> pairs, and a set of OrderedDict to group items by their put/get frequency. Each put/get operation updates the visited item's frequency, and the least recent visited item in the least frequency group is evicted when the capacity is full. For data consistency, items evicted in h_1 will be marked as *delete* in S and h_2. Similarly, the command feature corresponding to the item evicted in h_2 will be evicted in h_1 and S as well. For data persistency, *ChatCache* serializes the searching space S and two HashMaps h_1, h_2 to save in the file system.

We use ZeroMQ to manage the request queue and balance the workload among *workers*. The communication socket interface of client, *frontend*, *backend*, and *worker* are asynchronous. Each client has a unique UUID that is prepended to the request for *frontend* to distinguish. The UUID will be stripped by the received *worker* for processing, and prepend back before sending the response back to the client. One *worker* takes one thread to process, and ZeroMQ proxy provides load balance to chime the multi-threading *workers*. Each *worker* maintains its own understanding model, while S, h_1, h_2 are global parameters sharing among *workers*.

B. Distilled SLU Model

In general, SLU models have the pre-trained part and a few CNN, RNN, FC layers to compose the end-to-end schema. The pre-trained part contains the phonemes and word information. It takes the audio input with ASR targets, to remove the speaker difference raised by dialect, gender, age, *etc.*, and catch the language pattern. In this work, we inherit from an SLU model [24] for knowledge distillation and *ChatCache* deployment in audio command scenarios.

The SLU model is quite compact and small. It has only one RNN layer after the pre-trained part. The pre-trained module has a few CNN and RNN layers. Thus, we take a two-step knowledge distillation. We first distill the pre-trained model by downsizing and layer cutting to generate the distilled pre-train

TABLE II: Knowledge distillation on SLU model.

Accuracy	Compress rate	Accelerate rate
1.26%	2.67	5.07

model, and downsize the RNN layer to finish the distillation. We tried temperatures of [1, 2, 5, **10**] (bold indicates the best value) and set the cross-entropy weight as 0.5, Table II indicates the performance of distilled model comparing to the original one. Assume the original model has the number of parameters A, with S processing time in inference, and distilled model has A^*, S^* respectively. The compress rate is A/A^*, accelerate rate is S/S^*. After distillation, the SLU model is more than 5X in processing time, nearly 3X in the memory cost, with a performance drop of less than 1.28%.

VI. System Evaluation

A. Environment Setup

Dataset Targeted on language understanding services, we evaluate *ChatCache* on the following two language-oriented datasets, including one text-based, and one audio-based datasets.

The *Quora Similar Question* (*QSQ*) is altered from Quora Question Pairs (*QQP*). *QQP* is one of the benchmark datasets provided in GLUE [32]. The original *QQP* datasets label *1* or *0* to identify if a pair of questions are semantically same or not. We design QSQ based on QQP by grouping similar questions as classes to form a class dataset. As an example, let $p(i, j)$ represents one pair item with is labeled with 0 or 1. Assume $p(a, b) = p(a, c) = 1, p(c, d) = 0$. In *QSQ*, a, b, c are grouped together as class C_1, and d is in class C_2. For $363,871$ question pairs in *QQP*, we generate *QSQ* with $60,460$ classes, number of items in each class ranging from 2 to 109.

The *FSC* dataset contains spoken English commands that happen when interacting with a smart home or voice assistant. The dataset is composed of 16kHz single-channel audio files. Each audio file records one single spoken English smart home command, such as home automation on various devices at multiple locations, and task management. There are 97 English speakers, with $30,043$ commands in 31 types with 248 different expressions in total. The different expressions are collected based on human knowledge, and are consistent with the observation that, the voice command usually changes from time to time in the real-world usage [12].

Zipfian distribution The Zipfian curve [33] has been observed in query logs of smart speaker [28], textual web search engines [34], image view website [10]. *FSC* and *QSQ* naturally demonstrate the Zipfian distribution in terms of the number of requests for different semantic classes within the dataset . Therefore, we sample requests randomly by class to generate the request log that for cache warmup and request. The number of requests increases as the number of selected classes increases. Requests within the same type are duplicated in semantic, while differ in speakers and expressions.

1) Hardware setup: We use heterogeneous hardware platforms (Table III), including Raspberry Pi (RPI), Intel Fog Reference Node (FRD), Jetson AGX Xavier (Xavier), and a GPU workstation Gorilla to deploy *ChatCache* for its generality evaluation. These platforms differ on GPU equipment, CPU architecture, and memory. Compared to the other two costly hardware platforms, RPI is an affordable and suitable edge device to be considered as a gateway for a home environment, which is one of the typical scenarios for *ChatCache*. FRD and Xavier are viewed as qualified edge servers to serve multiple clients. Gorilla is a powerful workstation with four first ray-tracing GPUs. The performance of *ChatCache* to serve a single client is evaluated on all these platforms, and the performance to serve multiple clients is evaluated on FRD.

B. Cache Efficiency

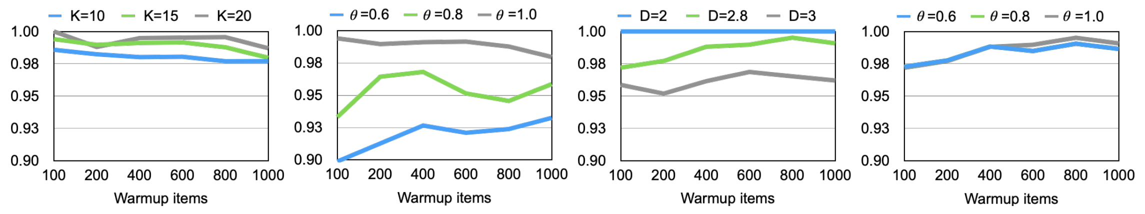
We define cache efficiency as the tradeoff between reusability and accuracy. The reusability is represented by recall, the ratio of the true cache hit to all hit items. It evaluates how many items correctly hit among all correctly distinguished items. Accuracy is represented by precision, the ratio of the true cache hit to expected hit items. It evaluates how many items correctly hit among all hit items. Depending on the cache configuration, there are two threshold policies to select nearest neighbors K'. One is the dynamic distance threshold; it selects K' based on the average distance D of K nearest neighbors. In other words, the distance distribution of K affects the composition of K'. Another is the strict distance threshold; it only selects distance shorter than a given value D to form K', D is not affected by distance distribution in K. Both dynamic and strict distance threshold policies select the dominant class with the cosine similarity θ. We discuss the impact of K, D, θ on the cache efficiency in different threshold designs. Figure 4 is the precision with the increasing number of warmup items in the cache, which is $2,000$ of the capacity. Figure 5 is the corresponding reuse rate in each circumstances. We randomly select ten classes from the *FSC* dataset to form the voice input. Items in each of the classes are semantically duplicated, which means they are not exactly the same; they differ from each other in either speakers or sentence expressions. We first warmup the cache system with a certain number of items, then use constant request items to evaluate. The warmup items increase during the evaluation, while the request items keep the same.

1) Dynamic distance threshold: Figure 4(a), 4(b) and Figure 5(a), 5(b) show the impact of K, θ on precision and recall in the dynamic distance threshold design respectively.

The average distance of all K nearest neighbors determines the dynamic distance threshold. Therefore, the increasing K contributes to increasing precision (*i.e.,* fewer items are classified mistakenly) and decreasing recall (*i.e.,* more items are considered as cache miss mistakenly) (θ fixed to 1.0 in Figure 4(a), 5(a)). When more candidates are involved in calculating the distance threshold, both the distance threshold and the number of items belonging to the true class are increasing. With more reference objects in classification, the search policy becomes more strict, with fewer items are determined as cache hit items. It affects both the correctly and mistakenly hit items. Correctly hit items are more affected as their number is hundreds of times than mistakenly hit items.

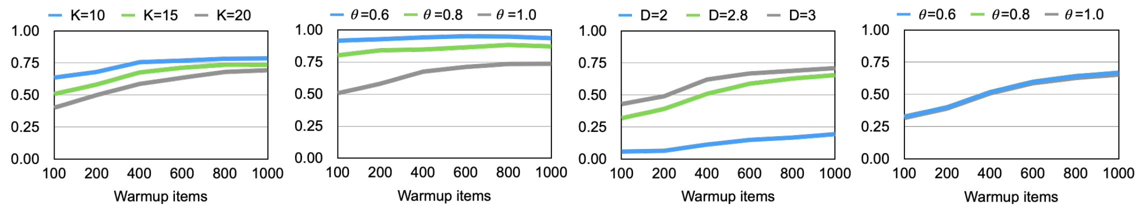
TABLE III: Hardware configuration.

Hardware	CPU	GPU	CPU Frequency	Cores	CPU Threads	Cost (USD)
Raspberry Pi mode 4B	ARMv7	N/A	1.5GHz	4	4	55.0
Intel Fog Reference Design	Intel Xeon E3-1275	N/A	3.6GHz	4	8	N/A
Jetson AGX Xavier (15W)	ARMv8	512-core Volta	2.3GHz	4	4	699.0
Jetson AGX Xavier (MAXN)	ARMv8	512-core Volta	2.3GHz	8	8	699.0
Gorilla	Intel Core i9-10940X	4 Quadro RTX 8000	3.30GHz	14	28	12,000.0



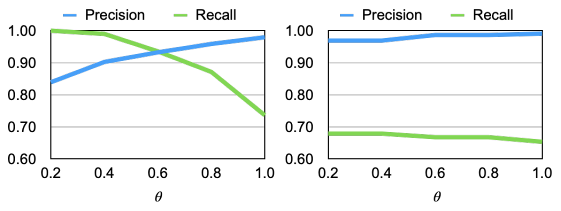
(a) K in dynamic distance. (b) θ in dynamic distance. (c) D in strict distance. (d) θ in strict distance.

Fig. 4: Precision with increasing warmup items. Request items and warmup items belongs to same group of classes.



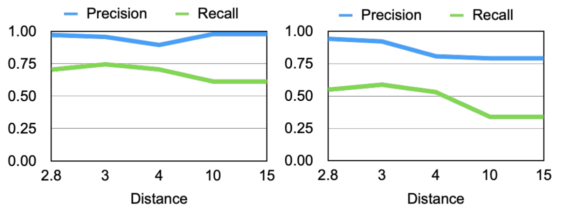
(a) K in dynamic distance. (b) θ in dynamic distance. (c) D in strict distance. (d) θ in strict distance.

Fig. 5: Recall with increasing warmup items.



(a) Dynamic threshold (b) Strict threshold

Fig. 6: Cache efficiency with different dominant factor θ.



(a) Voice requests (b) Text requests

Fig. 7: Cache efficiency with static distance D in different scenarios.

With K increasing, the decrease hit numbers and increase miss numbers demonstrate an increase in response accuracy and decrease in reuse rate. This impact on reuse rate is magnified with a small number of warmup items as fewer reference objects are obtained. The effect of K on precision recedes with K increasing, while on recall presents a constant performance drop. In the following experiences, if not otherwise specified, K is set to 15.

Comparing to the effect of K, response accuracy and reuse rate are more sensitive to the change of θ (Figure 4(b), 5(b)). The dominant factor θ represents the cosine similarity of the angle between the requested item and the dominant class. A higher value of θ denotes a closer angle with a higher probability that the requested item belongs to that class. As θ determines the sensitivity that a class is considered as dominator, increasing θ brings less cache hit rate, thus brings higher precision and lower recall rate (Figure 6(a)). In general, when θ is close to 1.0, the cache keeps high response accuracy and acceptable reuse rate. As one important design goal for a caching system is to keep a low false-positive rate, in the following experiences, if not otherwise specified, θ is 1.0.

2) Strict distance threshold: Different from the dynamic distance threshold where the number of nearest neighbors determines the candidate classes, from the preliminary experiment in Section II, we observe there existing some strict values that can distinguish inner and outer classes in different conversation scenarios.

Based on our observation from Figure 2(c), 2(d), we distinguish items from different semantic classes according to their Euclidean distance. The threshold value varies based on the different types of input, speakers, and statements. Figure 7 presents the performance when threshold distance varying from 2.8 to 15 in different conversational scenarios. We observe that when setting distance less than 4, cache efficiency benefits from a smaller threshold value in both cases. A closer observation with increasing warmup items in Figure 4(c), 5(c) shows that, too small distance ($D = 2$) results in extremely low reuse rate. For generality purposes, we set D as 2.8 in the rest of the experiments.

Given a strict distance threshold, the impact of θ on the result is minimal because the hard distance condition has filtered most of the objects belonging to different classes other than the requested item. More items in the warmup benefit the reuse rate, as more qualified nearest neighbors involving in the dominant class selection.

C. Conversational Scenarios

The evaluation of the cache efficiency discussed above is evaluated with *FSC* with the distilled SLU model. For the sake of generality, we discuss the conversational scenarios where the text is used as input as well. Text input is another primary interaction approach between the user and chat assistants, especially for mobile or web applications.

For *ChatCache* to understand text commands, we leverage the DistilBERT model provided by the Hugging Face community [35]. The modularized design of *ChatCache* makes it convenient to replace the SLU model with different semantic feature extraction models. The only thing need to be modified in *ChatCache* is the hyperparameter in search space S, as different understanding model generates features with different dimensions, and *ChatCache* needs to know the dimension in advance to create the searching space. The output of the pre-trained DistilBERT model is the semantic feature.

Figure 8 summarizes the cache performance for voice and text input with different configurations. The cache capacity and warmup items are fixed. Warmup items are randomly selected from 10 semantic classes for voice input and 20 classes for text input. We increase the number of semantic classes that form the requests to observe the response accuracy and reuse rate. Both dynamic (Figure 8(a), 8(c)) and strict distance threshold (Figure 8(b), 8(d)) methods are evaluated.

When the warmup items do not fully cover the semantic classes of requests, the response accuracy significantly drops when a dynamic distance threshold determines the dominant class. The strict distance threshold is a trade-off between reuse rate and response accuracy by maintaining a high precision with an acceptable recall rate. This pattern is general in both voice and text input scenarios. When the request classes are three times the warmup classes, the precision is maintained higher than 90%, with recall higher than 66% in audio and 50% in text. The efficiency of cache is related to the warmup's semantic classes. When the cache warmup items and requests are consistent in semantic classes, a dynamic distance threshold determined by the mean distance of NN can maintain a high value in precision and recall. When the warmup items are a subset of classes of request items, a strict distance threshold decreases the items that are considered as a hit, which effectively reduces the false positive rate while also increasing the false-negative rate. Therefore, for usage scenarios where the semantic classes (*i.e.,* user intents) are limited, such as chat assistant of customer service, *ChatCache* performs well when warmup with different types and deploy the dynamic distance threshold. On the contrary, for usage scenarios where the user intents are complicated, it is better to deploy *ChatCache* with a strict distance threshold to keep the high precision rate.

TABLE IV: Responsiveness of *ChatCache* on edge platforms.

Scenario	Platform	Response (ms)	Improve
Voice input	Cloud	1681.5	-
	RPI	663.1	60.6%
	Xavier (15W)	139.3	91.7%
	Xavier (MAXN)	85.8	94.9%
	FRD	30.0	98.0%
	Gorilla	36.7	97.8%
Text input	Cloud	206.0	-
	RPI	404.4	-
	Xavier (15W)	38.0	81.6%
	Xavier (MAXN)	29.3	85.8%
	FRD	26.3	87.2%
	Gorilla	18.1	91.2%

D. *ChatCache* Performance

1) Heterogenous edge: We first investigate *ChatCache* performance on heterogeneous edge platforms about the responsiveness and compare it with the cloud-based service (Table IV). We measure the end-to-end latency in milliseconds. For the cloud, responsiveness is the time consumed from a client sending a request to receiving the response. For edge platforms, only end-to-end latency of cache hit requests is measures to compute the overhead to deploy *ChatCache*. We measure the performance of the cloud with Google Dialogflow [1]. It is worth knowing that cloud service providers' responsiveness significantly varies from around 200 to 1,000 milliseconds, and Dialogflow is one of the services providing the fastest processing time.

Responsiveness and achieved improvements are evaluated on both *FSC* (SLU) and *QSQ* (NLU) datasets as two different conversation scenarios. RPI and FRD run the intelligent models on CPU, Xavier and Gorilla employ one GPU on each machine to run the models. Different edge platforms show promising performance for voice-based input by improving the responsiveness by $60.6\% - 98.0\%$. For text-based input, Xavier, FRD and Gorilla improves the responsiveness by $81.6\% - 91.2\%$. Given that cloud service responsiveness varies from 200 to $1,000$ ms roughly, these platforms potentially have higher improvement when integrated with different service providers. RPI presents its defect in processing text-based input, making it not suitable in this scenario. Even though, for voice-based input, RPI is an affordable solution to deploy *ChatCache* in a smart home environment.

Although the monetary cost of Gorilla surpasses Xavier and FRD, the performance of *ChatCache* on Gorilla does not proportionally improve. With significant responsiveness improvement on cache hit items and negligible overhead brought for cache miss items, all three platforms are capable of playing the role of edge server and deploy *ChatCache*.

2) System Scalability: In addition to the cache efficiency and responsiveness, we evaluate the scalability of *ChatCache*. Targeting on providing a scalable edge caching system for multiple clients, *ChatCache* is designed with an asynchronous network interface and multi-threading workers. Figure 9 compares *ChatCache*'s throughput with two mainstream cloud cognitive services: Google Dialogflow and Microsoft LUIS

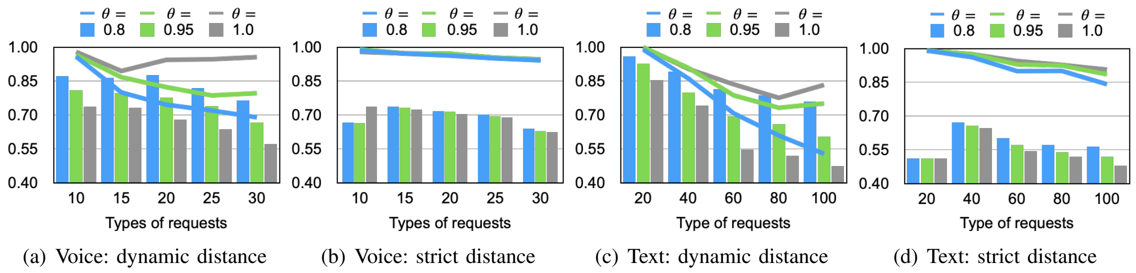


Fig. 8: Cache efficiency with an increasing number of uncached classes. The line is the precision rate, and the block is the recall rate. Warmup with 10 classes ((a)(b)), 20 classes ((c)(d)). The X-axis is the number of semantic classes that requests selected from.

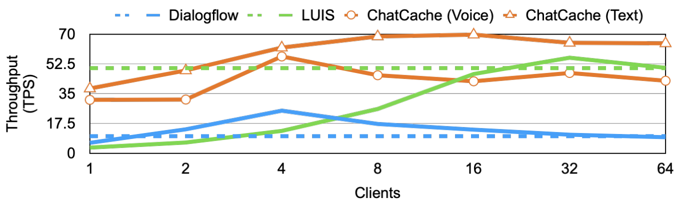


Fig. 9: The throughput of *ChatCache* comparing to the cloud cognitive service. Dash lines are the quota provided by each service.

[4]. Dialogflow and LUIS are cloud-based natural language understanding services. They provide APIs for developers to integrate with mobile or web applications, accept both audio and text input to understand the request's intent and execute pre-defined functions for fulfillments. The quota limitation on each service provided by their documents is presented in dash lines as a reference. *ChatCache* is deployed on Intel FRD, as its computation and storage resource is reasonable to be considered as an edge server for scalable deployment in practice. With the number of clients increasing, *ChatCache* achieves 42.6 transactions per second (tps) for voice input and 64.4 tps for text input. As the throughput and limits of service calls vary among cloud services, *ChatCache* is comparable to cloud services. With powerful resources (CPU or GPU) available, *ChatCache* can locate between the client and cloud to realize computation reuse on semantic redundancy items without imposing the bottleneck to the whole system.

The comparable throughput achieved by deploying *ChatCache* demonstrates that with a dedicated caching system designed for semantic redundancy and well-compressed intelligent models, *ChatCache* does not pose reduced performances on the critical path for conversational services.

VII. Related Work

With the increase of data volume, sending massive data from the client to the cloud causes unnecessary in-network transmission and heavy workload on the data center, making the service quality not well guaranteed. Caching at the edge significantly reduces the latency, in-network traffic, and data center workload. One typical edge caching application is content delivery networks (CDN) [36]. CDN publishes the content of the website to the geographic area closest to the user's network so that the user can obtain the desired content in the vicinity. Therefore, the user's access is not restricted by limited network bandwidth, extensive user access, and uneven distribution of application sites.

In cognitive services, semantic redundancy is commonly observed in requests, and a similarity searching can satisfy the request [37]. Depending on the similarity of request with the local cached item, the system determines if the request need to be forwarded to the remote server processing. Semantic redundancy has been leveraged for different application scenarios. For image recognition services, Drolia *et al.* propose Cachier [9] minimize latency by inserting a specified cache between the user and the cloud. In the virtual reality game field, Li *et al.* propose MUVR [11] as a framework to serve multiple virtual reality users at the same point of interest by maximizing the utilization of the edge cloud's computation and communication resources. Proactive caching strategy has been studied in the edge computing in terms of minimizing latency for popular computation results [38]. Zhang *et al.* [39] leverages a deep reinforcement learning algorithm to find the proactive caching policy for multi-view 3D videos. In the recommendation system area, Sermpezis *et al.* [16], [17] propose an idea called "Soft cache hits", which partially satisfy the user with related content when local cache miss happens. FoggyCache [10] addresses computation reuse for fuzzy redundancy in contextual recognition for across device collaboration. Falchi *et al.* [40] modifies LRU to work in metric spaces for content-based image retrieval systems. Clipper [15] is an online prediction serving system and maintains a prediction cache for the generic prediction function.

VIII. Conclusion

In this work, we observe and discuss fuzzy redundancy's insufficiency in addressing duplicate computation in conversational services. We propose *ChatCache* as a general semantic similarity approach, designing with a hierarchical cache, which functions wherever resides at the client, cloud, or edge within the network. *ChatCache* is a modularized and scalable system and supported with compressed intelligent models. The evaluation shows that *ChatCache* is a general and scalable solution as an edge server between the client and cloud to capture semantic redundancy and reuse computation results for conversational services with different input data types and usage scenarios. For most of the evaluated platforms, *ChatCache* reduces the user-perceived latency by at least 91.7% and 81.6% for audio, text requests. It also demonstrates comparable throughput performance as cloud cognitive service with 42.6 and 64.4 tps for audio and text requests, respectively.

References

[1] Google, "Dialogflow," 2020, https://dialogflow.com/.

[2] Amazon, "Amazon lex," 2020, https://aws.amazon.com/lex/.

[3] Facebook, "Wit.ai," 2020, https://wit.ai/.

[4] Microsoft, "Language understanding (luis)," 2019, https://www.luis.ai/.

[5] IBM, "Ibm watson products and solutions," https://www.ibm.com/watson/services/conversation/.

[6] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5mb model size." [Online]. Available: http://arxiv.org/abs/1602.07360

[7] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once-for-all: Train one network and specialize it for efficient deployment." [Online]. Available: http://arxiv.org/abs/1908.09791

[8] H. Wang, Z. Wu, Z. Liu, H. Cai, L. Zhu, C. Gan, and S. Han, "HAT: Hardware-aware transformers for efficient natural language processing." [Online]. Available: http://arxiv.org/abs/2005.14187

[9] U. Drolia, K. Guo, J. Tan, R. Gandhi, and P. Narasimhan, "Cachier: Edge-caching for recognition applications," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 276–286.

[10] P. Guo, R. Li, B. Hu, and W. Hu, "FoggyCache: Cross-Device Approximate Computation Reuse," *Living on the Edge*, p. 16, 2018.

[11] Y. Li and W. Gao, "Muvr: Supporting multi-user mobile virtual reality with resource constrained edge cloud," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2018, pp. 1–16.

[12] F. Bentley, C. Luvogt, M. Silverman, R. Wirasinghe, B. White, and D. Lottridge, "Understanding the Long-Term Use of Smart Speaker Assistants," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 2, no. 3, pp. 1–24, Sep. 2018. [Online]. Available: https://dl.acm.org/doi/10.1145/3264901

[13] J. Ramos *et al.*, "Using tf-idf to determine word relevance in document queries," in *Proceedings of the first instructional conference on machine learning*, vol. 242, no. 1. Citeseer, 2003, pp. 29–48.

[14] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs," *arXiv:1603.09320 [cs]*, Aug. 2018, arXiv: 1603.09320. [Online]. Available: http://arxiv.org/abs/1603.09320

[15] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 613–627.

[16] P. Sermpezis, T. Spyropoulos, L. Vigneri, and T. Giannakas, "Femto-caching with soft cache hits: Improving performance with related content recommendation," in *GLOBECOM 2017-2017 IEEE Global Communications Conference*. IEEE, 2017, pp. 1–7.

[17] P. Sermpezis, T. Giannakas, T. Spyropoulos, and L. Vigneri, "Soft cache hits: Improving performance through recommendation and delivery of related content," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 6, pp. 1300–1313, 2018.

[18] J. Weston, S. Chopra, and A. Bordes, "Memory Networks," *arXiv:1410.3916 [cs, stat]*, Nov. 2015, arXiv: 1410.3916. [Online]. Available: http://arxiv.org/abs/1410.3916

[19] P. Gupta, R. Ahmad, M. Shrivastava, P. Kumar, and M. K. Sinha, "Improve performance of machine translation service using memcached," in *2017 17th International Conference on Computational Science and Its Applications (ICCSA)*. IEEE, 2017, pp. 1–8.

[20] D. G. Lowe, "Object recognition from local scale-invariant features," in *Proceedings of the seventh IEEE international conference on computer vision*, vol. 2. Ieee, 1999, pp. 1150–1157.

[21] J. T. Foote, "Content-based retrieval of music and audio," in *Multimedia Storage and Archiving Systems II*, vol. 3229. International Society for Optics and Photonics, 1997, pp. 138–147.

[22] scikit learn, "sklearn, feature extraction, text, countvectorizer," 2020, https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html.

[23] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter." [Online]. Available: http://arxiv.org/abs/1910.01108

[24] L. Lugosch, M. Ravanelli, P. Ignoto, V. S. Tomar, and Y. Bengio, "Speech model pre-training for end-to-end spoken language understanding." [Online]. Available: http://arxiv.org/abs/1904.03670

[25] M. A. Gordon, K. Duh, and N. Andrews, "Compressing BERT: Studying the effects of weight pruning on transfer learning." [Online]. Available: http://arxiv.org/abs/2002.08307

[26] O. Zafrir, G. Boudoukh, P. Izsak, and M. Wasserblat, "Q8bert: Quantized 8bit BERT." [Online]. Available: http://arxiv.org/abs/1910.06188

[27] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network." [Online]. Available: http://arxiv.org/abs/1503.02531

[28] L. Xu, A. Iyengar, and W. Shi, "CHA: A caching framework for home-based voice assistant systems," in *2020 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2020, pp. 293–306.

[29] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A Survey of Model Compression and Acceleration for Deep Neural Networks," *arXiv:1710.09282 [cs]*, Jun. 2020, arXiv: 1710.09282. [Online]. Available: http://arxiv.org/abs/1710.09282

[30] M. Feng, B. Xiang, M. R. Glass, L. Wang, and B. Zhou, "Applying deep learning to answer selection: A study and an open task," in *2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*. IEEE, 2015, pp. 813–820.

[31] P. Hintjens, *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.", 2013.

[32] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding," 2018. [Online]. Available: http://arxiv.org/abs/1804.07461

[33] M. E. Newman, "Power laws, pareto distributions and zipf's law," *Contemporary physics*, vol. 46, no. 5, pp. 323–351, 2005.

[34] F. Lillo and S. Ruggieri, "Estimating the total volume of queries to google," in *The World Wide Web Conference*. ACM, 2019, pp. 1051–1060.

[35] H. Face, "The ai community building the future." 2021, https://huggingface.co/.

[36] A.-M. K. Pathan and R. Buyya, "A taxonomy and survey of content delivery networks," *Grid Computing and Distributed Systems Laboratory, University of Melbourne, Technical Report*, vol. 4, 2007.

[37] M. Garetto, E. Leonardi, and G. Neglia, "Similarity caching: Theory and algorithms," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 526–535.

[38] M. S. Elbamby, M. Bennis, and W. Saad, "Proactive edge computing in latency-constrained fog networks," in *2017 European conference on networks and communications (EuCNC)*. IEEE, 2017, pp. 1–6.

[39] Z. Zhang, Y. Yang, M. Hua, C. Li, Y. Huang, and L. Yang, "Proactive caching for vehicular multi-view 3d video streaming via deep reinforcement learning," *IEEE Transactions on Wireless Communications*, 2019.

[40] F. Falchi, C. Lucchese, S. Orlando, R. Perego, and F. Rabitti, "A metric cache for similarity search," in *Proceeding of the 2008 ACM workshop on Large-Scale distributed systems for information retrieval - LSDS-IR '08*. Napa Valley, California, USA: ACM Press, 2008, p. 43. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1458469.1458473