CodeBlend

A survey of efficient KV-Caching methods for graph-based structures

Ayush Raman

University of Illinois Urbana-Champaign, ayushr3@illinois.edu

Ritul Soni

University of Illinois Urbana-Champaign, rsoni27@illinois.edu

Shashwat Mundra

University of Illinois Urbana-Champaign, mundra3@illinois.edu

This research explores the adaptation of the **CacheBlend** framework, originally designed for efficient key-value (KV) cache management in large language models (LLMs), to the domain of code and its dependencies[1]. The project focuses on leveraging mutual import relationships between code packages to selectively blend caches, optimizing dependency management for faster and more efficient retrieval. By constructing a dependency graph of Python files in a local repository, we enable intelligent cache blending based on shared dependencies. Additionally, we investigate various cache eviction strategies, such as Least Recently Used (LRU), Least Frequently Used (LFU), Least Weighted (LW), and our own Composite Cache to maintain optimal cache performance under memory constraints. The proposed system aims to strike a balance between computational efficiency and storage limitations while preserving the integrity and accessibility of cached dependencies. Through this work, we aim to extend the principles of CacheBlend to support dynamic and scalable dependency management in software engineering workflows.

SYS GEN AI CONCEPTS • KV-Caching • RAG • LLMs

ACM Reference Format:

Ayush Raman, Ritul Soni, and Shashwat Mundra. 2024. CodeBlend Comparing A Survey of efficient KV-Caching Methods for Graph Based Structures. 1,1 (December 2024), 10 pages. https://doi.org/xxx.xxx/xxxxx.xxxxx

1 Introduction

1.1 Background

Managing dependencies in large Python repositories presents significant challenges, particularly as the complexity and size of modern software projects grow. Python's flexibility and extensive ecosystem of libraries make it a popular choice for developers, but this comes at the cost of dependency management difficulties. Issues such as *dependency hell*—where conflicting versions of shared libraries cause compatibility problems—are common, especially in projects with numerous transitive dependencies. Even with tools like \`pip\`, \`virtualenv\`, \`conda\`, and \`poetry\`, developers face trade-offs between ease of use, storage efficiency, and scalability. For instance, while virtual environments isolate dependencies for individual projects, they can quickly consume storage space when managing multiple environments.

In addition to these challenges, dependency management becomes even more critical in workflows that leverage Large Language Models (LLMs). LLMs like GPT-4 rely on **context windows**—the number of tokens the model can process at once—to provide meaningful responses. Larger context windows enhance the model's ability to process complex inputs, such as entire codebases or interdependent files, enabling better comprehension and output generation. However, this increased capacity also demands efficient dependency handling to avoid bottlenecks in computational workflows. For example, AI-powered coding assistants benefit from analyzing entire project files simultaneously, but this requires seamless access to dependencies across the codebase.

The concept of caching plays a pivotal role in addressing these challenges. Caching mechanisms are widely used in build systems and dependency management tools to improve efficiency by storing frequently accessed data or results. For example, Apache Maven's build cache optimizes incremental builds by focusing only on modified parts of a project graph. Similarly, session-based dependency management in distributed systems like Apache Spark allows dynamic updates to dependencies during runtime. These approaches highlight the potential for caching techniques to streamline workflows in environments with complex dependency structures.

1.2 Problem Statement

Despite advancements in tools and methodologies, current caching methods for Python dependencies remain inefficient and lack scalability. Traditional solutions often fail to account for dynamic changes in dependencies or the intricate relationships between modules within large repositories. For instance:

- Static dependency management systems cannot adapt to runtime changes, leading to inefficiencies in dynamic workflows.
- Cache invalidation issues and conflicts between nested caches result in outdated or redundant data being stored, further complicating dependency resolution.
- Existing eviction strategies like Least Recently Used (LRU) or Least Frequently Used (LFU) are not optimized for the unique characteristics of Python repositories.

These limitations create a pressing need for a more dynamic and scalable approach to dependency management that can handle the growing demands of modern software development.

1.3 Objectives

This project aims to address these challenges by leveraging mutual import relationships between Python files to construct a dependency graph that informs intelligent cache blending. Specifically:
1. Develop a system that dynamically optimizes dependency management by blending caches based on shared dependencies.
2. Investigate advanced cache eviction strategies—such as Least Weighted (LW) and a custom Composite Cache—to maintain performance under memory constraints.
3. Test the proposed system on real-world Python repositories (e.g., `numpy`, `pandas`, `matplotlib`) and compare its performance against traditional CacheBlend methods.

By tackling these objectives, this work seeks to extend the principles of CacheBlend to support dynamic and scalable dependency management in software engineering workflows while balancing computational efficiency and storage limitations.

2 Related Work

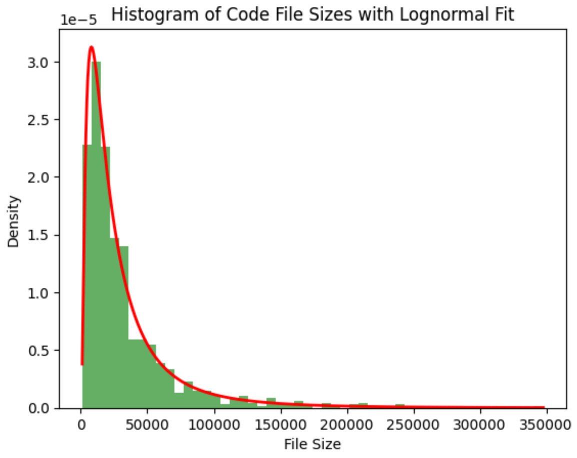


Figure 0: Visualization of lognormal curve

The study "On the Distribution of Source Code File Sizes" by Herraiz and Germán investigates the statistical properties of source code file sizes across various software repositories[2]. The authors identify that source code file sizes follow a **lognormal distribution**, where most files are small, but a minority are disproportionately large. This finding is significant because it reflects the natural organization of software systems, where smaller files often encapsulate specific functionalities, while larger files serve as central modules or frameworks with higher complexity and more dependencies. This insight directly influenced our artificial graph construction methodology, where we used a weighted lognormal distribution to simulate realistic file size distributions in dependency graphs. By biasing larger files to have more dependents, our approach aligns with the real-world observation that critical components in software systems tend to be reused extensively.

The implications of this work extend beyond statistical modeling, as understanding file size distributions can inform optimization strategies for dependency management and caching. For example, larger files with more dependents are likely to be queried more frequently, making them prime candidates for caching strategies that prioritize high-impact nodes. By incorporating these principles into our system design, we were able to create dependency graphs that not only mimic real-world software structures but also provide a robust testing ground for evaluating cache blending techniques.

CacheBlend is a caching framework designed to optimize key-value **(KV) cache reuse** in large language model (LLM) inference tasks. The system addresses the challenge of combining precomputed KV caches from multiple text chunks while maintaining high generation quality and minimizing computational overhead. CacheBlend achieves this by selectively recomputing a small subset of KV tokens rather than fully recomputing or blindly reusing entire caches. This selective recomputation is guided by sparsity in attention matrices, allowing CacheBlend to balance speed and accuracy effectively. By pipelining partial KV updates with cache retrieval, CacheBlend enables the use of slower storage devices without increasing inference latency, significantly reducing time-to-first-token (TTFT) and improving throughput.

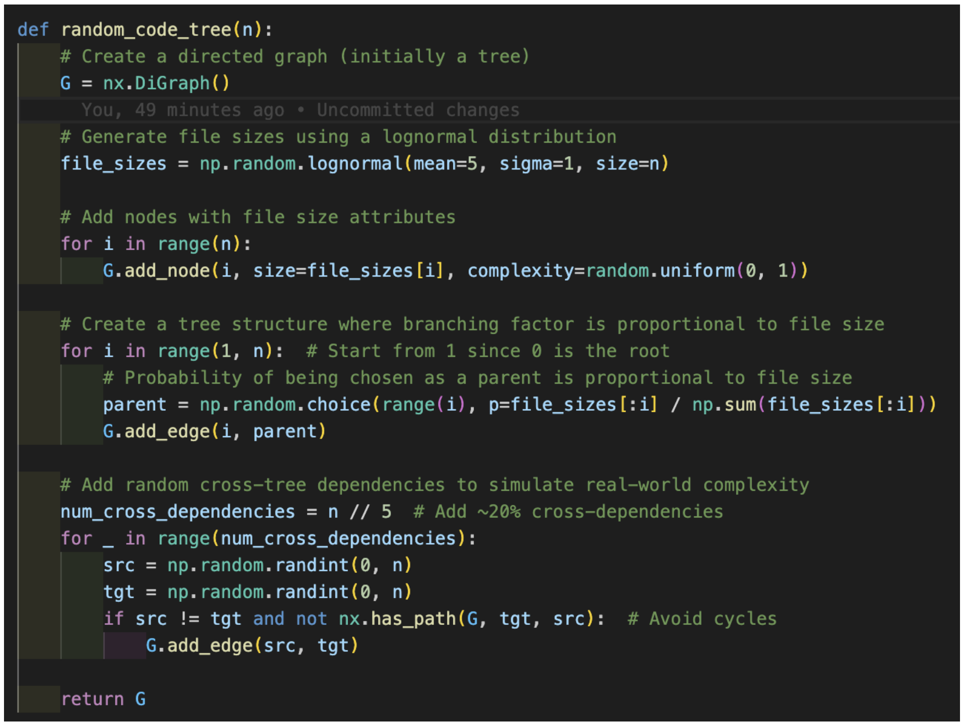
The principles underlying CacheBlend inspired key aspects of our system design for dependency management in Python repositories. Similar to how CacheBlend selectively recomputes KV caches based on token dependencies, our system selectively blends caches only when mutual dependencies exist between modules. This ensures efficient use of computational resources while preserving the integrity of shared dependencies. Additionally, CacheBlend's focus on balancing storage constraints with computational efficiency parallels our exploration of eviction strategies like Least Recently Used (LRU), Least Frequently Used (LFU), and Composite Cache in memory-constrained environments.

Both works emphasize the importance of leveraging structural patterns—whether in source code or LLM inputs—to optimize performance under resource constraints. While Herraiz and Germán's study provides a foundation for understanding real-world software characteristics, CacheBlend demonstrates how selective recomputation can enhance efficiency in dynamic systems. Our project bridges these domains by applying insights from source code analysis to caching strategies inspired by LLM optimization techniques. This synthesis not only advances dependency management in Python repositories but also highlights the broader applicability of CacheBlend's principles beyond natural language processing tasks.

3 Methodology

3.1 Eviction Schemes

3.1.1 Artificial Graph Construction



```python
def random_code_tree(n):
    # Create a directed graph (initially a tree)
    G = nx.DiGraph()
      You, 49 minutes ago • Uncommitted changes
    # Generate file sizes using a lognormal distribution
    file_sizes = np.random.lognormal(mean=5, sigma=1, size=n)

    # Add nodes with file size attributes
    for i in range(n):
        G.add_node(i, size=file_sizes[i], complexity=random.uniform(0, 1))

    # Create a tree structure where branching factor is proportional to file size
    for i in range(1, n):  # Start from 1 since 0 is the root
        # Probability of being chosen as a parent is proportional to file size
        parent = np.random.choice(range(i), p=file_sizes[:i] / np.sum(file_sizes[:i]))
        G.add_edge(i, parent)

    # Add random cross-tree dependencies to simulate real-world complexity
    num_cross_dependencies = n // 5  # Add ~20% cross-dependencies
    for _ in range(num_cross_dependencies):
        src = np.random.randint(0, n)
        tgt = np.random.randint(0, n)
        if src != tgt and not nx.has_path(G, tgt, src):  # Avoid cycles
            G.add_edge(src, tgt)

    return G
```

Figure 1: Algorithm for generating random code graphs

To simulate realistic dependency graphs for Python repositories, we developed a methodology that combines probabilistic modeling and simulation techniques. The foundation of our approach lies in using a **lognormal distribution** to generate file sizes, which mimics the natural variation observed in real-world codebases where most files are small, but a few are disproportionately large. Each file in the graph is assigned a size based on this distribution, and larger files are biased to have more dependents, reflecting their central role in complex software systems. This probabilistic weighting ensures that the graph structure adheres to realistic patterns of dependency propagation, where critical components are reused more frequently.

The simulation begins by generating a directed acyclic graph (DAG) using a custom `random_code_graph` function and a probability distribution function (PDF) to represent the distribution of nodes the simulation will stream from. Each node represents a Python file, and edges indicate dependencies between files. To enhance realism, we introduce cross-tree dependencies to simulate cases where files from different branches of the graph interact. For each node, attributes such as file size and complexity are assigned, which influence its role in dependency resolution and caching performance. Larger files with higher complexity are more likely to be queried during simulations, as they represent critical components or modules with intricate logic. This setup allows us to model scenarios where dependency resolution is computationally expensive and caching strategies become crucial for efficiency.

To evaluate caching strategies under varying conditions, the simulation iteratively tests different cache configurations across graphs of increasing sizes. For each graph size, multiple random graphs are generated, and nodes are queried based on their dependency relationships and weighted probabilities derived from their attributes. The number of queries per node is determined by a logarithmic function of its size, complexity, and simulated "bug complexity," capturing the real-world scenario where larger and more complex files are accessed more frequently during debugging or execution. Cache performance metrics such as hit rates and query timings are recorded for each cache replacement policy (e.g., LRU, LFU) across different cache sizes. This iterative process provides insights into how graph structure and node attributes impact cache efficiency, enabling us to compare strategies systematically under controlled yet realistic conditions.

3.1.2 Real World Data

To evaluate the effectiveness of our caching schemes, we tested them against real-world Python repositories by parsing their files and constructing dependency graphs. The repositories used for testing included popular open-source projects such as `numpy`, `pandas`, `matplotlib`, and `scikit-learn`. These repositories were selected for their complexity and extensive use of interdependent modules, making them ideal for testing the scalability and performance of our system. By analyzing the import relationships between Python files, we were able to construct directed dependency graphs where nodes represented individual files and edges denoted import dependencies.

The process began with parsing the Python files in each repository to extract their import statements. Using tools like Python's `ast` module, we programmatically analyzed the abstract syntax trees of these files to identify all direct dependencies. This information was then used to build a dependency graph using a graph library such as NetworkX. Each node in the graph was annotated with metadata such as file size and complexity, which were derived from file attributes like line count and cyclomatic complexity. These annotations allowed us to simulate realistic scenarios where larger and more complex files are accessed more frequently during execution or debugging.

Once the dependency graphs were constructed, we applied our caching schemes to simulate dependency resolution workflows. Each node was associated with a cache, and queries were made to resolve dependencies based on the graph structure. For example, when a file was accessed, its direct dependencies (and potentially transitive dependencies) were queried from the cache. By running multiple iterations of these simulations across different repositories, we measured key performance metrics such as cache hit rates, query times, and memory usage. This approach allowed us to compare various caching strategies—including Least Recently Used (LRU), Least Frequently Used (LFU), Least Weighted (LW), and our custom Composite Cache—under realistic conditions.

Through this testing methodology, we were able to assess how well each caching scheme performed in handling real-world dependency graphs. The results provided valuable insights into the trade-offs between computational efficiency and memory constraints for each strategy. Additionally, testing on actual repositories ensured that our system could handle the diverse structures and complexities found in real-world Python projects, validating its applicability beyond artificial simulations.

3.2 System Design

The system design of our dependency management solution focuses on **selective cache blending**, where caches are merged only when dependencies between them exist. This approach ensures that computational resources are utilized efficiently, avoiding unnecessary blending of unrelated caches while maintaining the integrity and accessibility of shared dependencies. Inspired by CacheBlend's selective recomputation framework for LLMs, our system adapts these principles to Python dependency graphs, leveraging the mutual import relationships between modules to determine when and how caches should be blended.

At the core of the system is a dependency graph that represents Python files as nodes and their import relationships as directed edges. Each node is associated with a cache that stores precomputed results or frequently accessed data. When two nodes (or files) share a dependency—meaning one imports the other or both depend on a common module—their caches are selectively blended. This blending process involves merging the relevant portions of each cache to create a unified view that can be accessed by both files. By doing so, the system avoids redundant computations and ensures that shared dependencies are resolved efficiently. Importantly, caches are only blended when there is a direct or transitive dependency between the corresponding nodes, minimizing overhead and preserving memory efficiency.

The selective blending mechanism is further optimized through **dynamic cache eviction strategies**. When memory constraints arise, the system evaluates which caches to retain or evict based on their usage patterns and dependency relationships. For example, caches associated with highly interconnected nodes (those with many dependents) are prioritized for retention since their eviction would result in higher recomputation costs. This dynamic approach ensures that cache blending remains efficient even as the scale and complexity of the dependency graph increase. By combining selective blending with intelligent eviction strategies, the system achieves a balance between computational efficiency and storage limitations, making it well-suited for large-scale Python repositories.

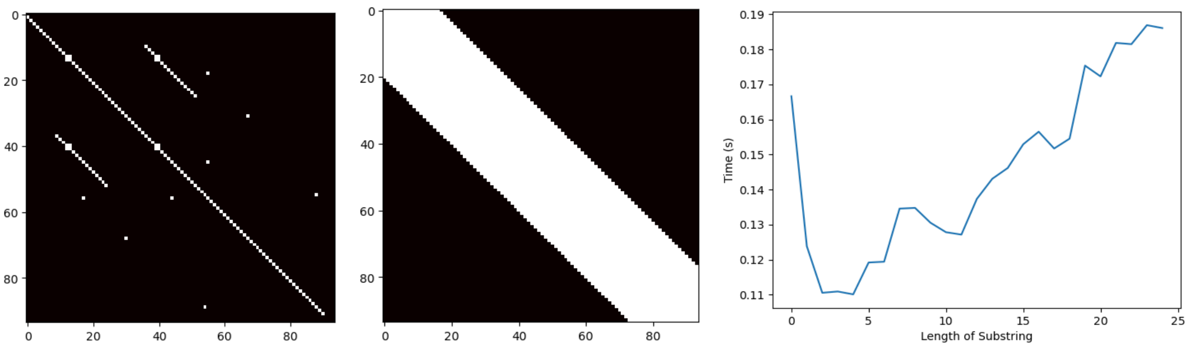


Figure 1.1: Visualization of the FASTA algorithm and its running time. The left two images show the “heat map” and the diagonal across which the dynamic programming is performed.

The FASTA algorithm, originally designed for sequence alignment in bioinformatics[3], can be adapted to address the challenge of inserting diffs fuzzily when working with outputs from large language models (LLMs). LLMs often generate code or text that is slightly misaligned with the expected context, introducing minor discrepancies in lines or dependencies. These discrepancies, while small, can disrupt workflows that rely on precise diff application. By leveraging FASTA's principles of local alignment and fuzzy matching, we can efficiently identify approximate matches between the generated output and the target context, enabling robust diff insertion.

FASTA operates by identifying **k-word matches** (short substrings of length k) between two sequences and scoring them based on their density along diagonals in a similarity matrix. This heuristic approach allows FASTA to quickly locate regions of high similarity, even in the presence of minor differences such as insertions, deletions, or substitutions. Once these regions are identified, FASTA refines the alignment using substitution matrices and gap penalties to account for mismatches and structural variations. This process is particularly useful for handling LLM outputs where slight deviations in formatting or syntax may occur. By treating code lines as sequences and using FASTA's scoring mechanisms, we can identify approximate locations where diffs should be applied.

In our system, we adapted these principles to support fuzzy diff insertion. When an LLM generates slightly off-context lines, such as a misplaced import statement or a shifted function definition, the FASTA-inspired approach identifies the most likely insertion point by comparing the generated output with the target file's existing structure. For example, if an import statement is generated with minor variations (e.g., missing whitespace or reordered imports), the algorithm scores potential insertion points based on their local similarity to the surrounding context. This ensures that diffs are applied in a way that minimizes disruptions to the existing codebase.

The use of FASTA for this purpose also offers flexibility in handling edge cases. By adjusting parameters such as gap penalties and word size (k), we can fine-tune the sensitivity of the alignment process to accommodate different levels of fuzziness. For instance, higher gap penalties can discourage large insertions or deletions, while smaller k-values increase sensitivity to subtle changes. This adaptability makes FASTA particularly well-suited for scenarios where LLM outputs vary in quality or consistency. Overall, incorporating FASTA into our system provides a powerful tool for handling fuzzy diff insertion efficiently and accurately, ensuring smoother integration of LLM-generated content into existing workflows.

4 Results

4.1 Eviction Scheme Results

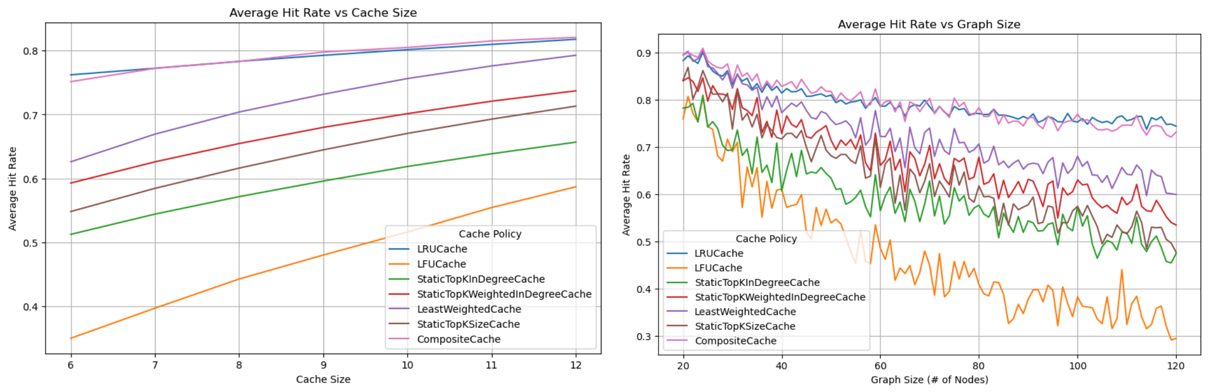


Figure 2: We can see that LFU and LW Caches have greater rates of improvement per extra unit of cache space than LRU (which is consistently good). This result informs how we built Composite Cache and, as seen on the right, it performs better for smaller graphs than LRU.

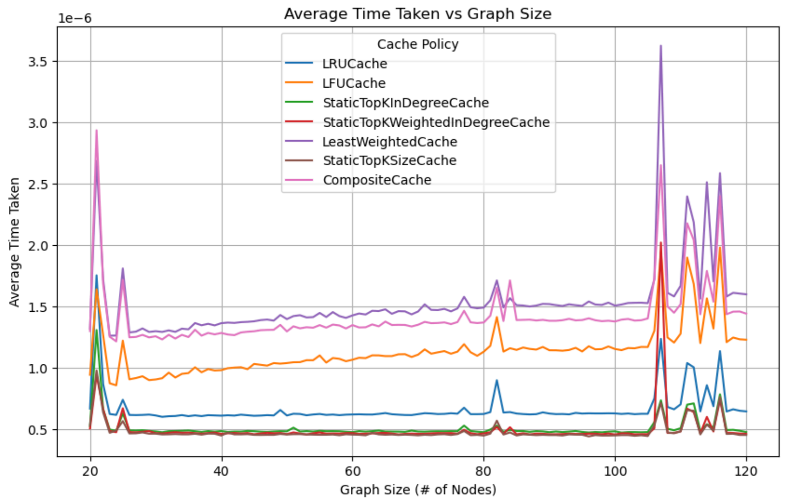


Figure 4: We can see that LRU takes considerably less time to run than Composite Cache

On artificial data, we are able to bring the Composite Cache to levels performing nearly as well as LRU. The Composite Cache is a hybrid caching strategy designed to integrate multiple caching policies into a single, unified system. This approach leverages the strengths of both static and dynamic caching mechanisms to optimize performance across diverse dependency management scenarios. By combining specialized static caches with flexible dynamic caches, the Composite Cache aims to balance computational efficiency, memory usage, and adaptability to workload variations.

The design of the Composite Cache begins with a division of its total capacity into two main components: static caches and dynamic caches. Static caches are allocated a fixed portion of the capacity and are tailored for specific use cases, such as prioritizing nodes with high in-degree (e.g., `StaticTopKWeightedInDegreeCache`). These nodes are critical in dependency graphs as they represent files or modules that are frequently reused by others. By dedicating resources to these high-impact nodes, the static caches ensure that commonly accessed dependencies remain readily available, reducing recomputation costs.

The remaining capacity is distributed among dynamic caches, which adapt to changing access patterns. In the implementation provided, two dynamic caching strategies are employed: Least Weighted Cache and Least Recently Used (LRU) Cache. The Least Weighted Cache prioritizes nodes based on a combination of their size and complexity, ensuring that larger or more computationally expensive dependencies are retained in the cache. Meanwhile, the LRU Cache focuses on temporal locality by retaining recently accessed nodes. Together, these dynamic caches provide flexibility in handling varying workloads and access patterns.

When a node is queried, the Composite Cache first checks its static caches for a match. If the node is found in one of these caches, it is immediately returned, ensuring fast access for critical dependencies. If not, the system proceeds to search the dynamic caches in sequence. This hierarchical query mechanism ensures that high-priority nodes are accessed quickly while still allowing for adaptability in less predictable scenarios.

If a node is not present in any cache, the Composite Cache employs a **weighted random selection mechanism** to decide which cache will handle the query. The weights are based on the hit rates of each cache, ensuring that more effective caches are given higher priority. This approach dynamically adjusts resource allocation based on real-time performance metrics, optimizing overall cache efficiency.

The Composite Cache's multi-layered design offers several advantages:
1. Specialization with Flexibility: By combining static and dynamic caching strategies, it addresses both predictable and unpredictable access patterns.
2. Dynamic Adaptation: The weighted random selection mechanism allows the system to adapt dynamically to workload changes by favoring high-performing caches.
3. Efficient Resource Utilization: The separation of static and dynamic capacities ensures that critical resources are allocated effectively while leaving room for flexibility.
4. Improved Hit Rates: The combination of targeted static caching for high-impact nodes and adaptive dynamic caching for general workloads leads to higher overall hit rates.

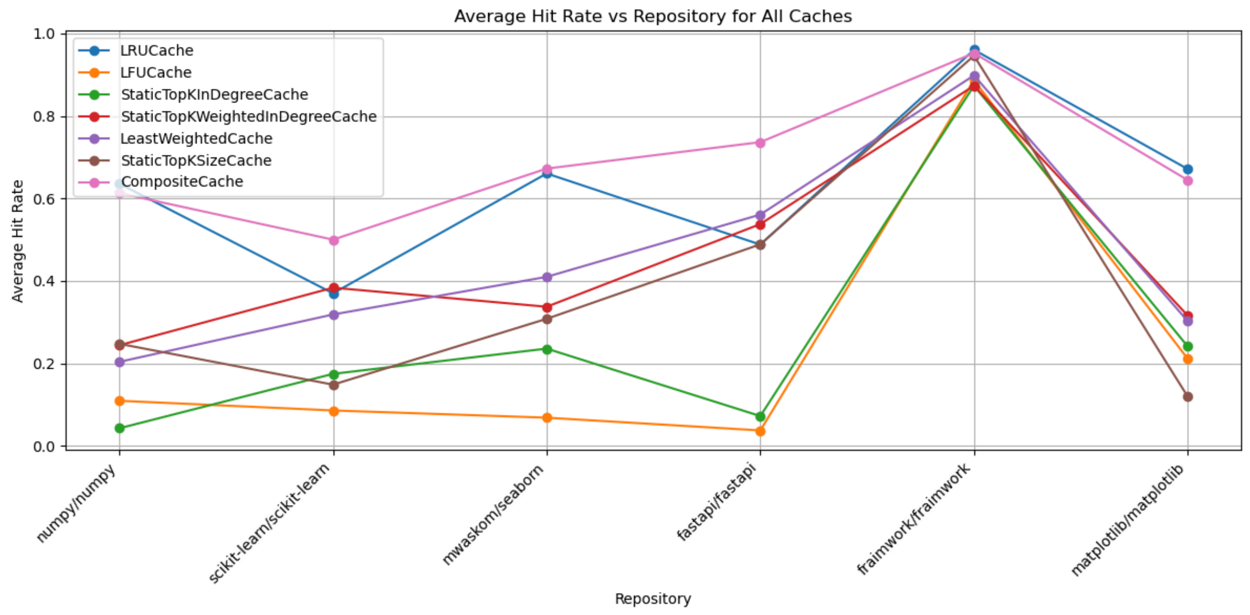


Figure 5: We can see that LFU and LW Caches have greater rates of improvement per extra unit of cache space than LRU (which is consistently good). This result informs how we built Composite Cache.

The results from the artificial data are seemingly brought to the real data as well, where Composite Cache consistently outperforms LRU, giving a promising foundation for improving the performance of CacheBlend with just the eviction mechanism itself. We tested against real python open source projects such as numpy and fastapi.

4.2 System Results

To evaluate the efficacy of our system, we divided the evaluation process into two distinct tasks: the "wrecker" and the "fixer." The wrecker was responsible for traversing the repository and breaking files based on a randomly generated distribution. This simulated real-world scenarios where codebases encounter bugs or errors due to missing or corrupted dependencies. The fixer, on the other hand, was tasked with resolving these issues by identifying a given file's dependencies and attempting to restore functionality. This dual-task framework was designed to stress-test our system's ability to handle dependency resolution dynamically while assessing how well our caching strategies could support such workflows.

The goal of this evaluation was to measure how effectively our system could leverage selective cache blending and intelligent eviction strategies to assist the fixer in resolving broken files efficiently. By simulating dependency resolution tasks across repositories, we aimed to quantify metrics such as cache hit rates, query times, and overall time-to-resolution for broken files. However, due to system limitations, including computational constraints and insufficient simulation fidelity for large-scale repositories, no meaningful data could be gathered from this portion of the evaluation (the LLM would often just generate nonsense; our best guess is that the code files contain too many tokens for LLM of this scale to effectively make sense of anything; future work could try to structurally decompose code into graphs and use Graph Neural Networks (GNNs) to perform inference, but this is pure speculation and out of the scope of our project). These limitations hindered our ability to fully validate the system's performance in real-world debugging scenarios.

Despite these challenges, the design of this evaluation framework highlights a promising direction for future work. With access to more robust computational resources and larger datasets, the wrecker-fixer methodology could serve as a powerful tool for testing caching systems in dynamic environments. By simulating realistic failure scenarios and evaluating how effectively caches can support dependency resolution workflows, this approach has the potential to provide deeper insights into the scalability and reliability of advanced caching strategies like those implemented in our system.

5 Discussion

5.1 Limitations

One of the primary limitations of our evaluation lies in the scale of the models and datasets used for testing. While our approach effectively demonstrates the feasibility of optimizing dependency management and cache blending in Python repositories, the simulated graphs and real-world repositories we tested may not fully capture the complexities of larger-scale systems. For instance, real-world enterprise-level codebases often involve millions of lines of code, intricate dependency hierarchies, and dynamic interactions between modules. Our methodology, constrained by computational resources, focused on moderately sized graphs and repositories, which may not provide a complete picture of how our system scales under extreme conditions.

Furthermore, the models used for simulation and evaluation were relatively small compared to state-of-the-art large language models (LLMs) like GPT-4 or LLaMA-3.2. Larger models could have provided a more thorough assessment of our system's performance, particularly in terms of handling more extensive context windows and complex dependency relationships. CacheBlend's demonstrated ability to reduce time-to-first-token (TTFT) and improve throughput in LLMs suggests that applying similar techniques to larger dependency graphs could reveal additional insights into scalability and efficiency. For example, employing larger models with broader context windows might better simulate real-world scenarios where dependencies span across multiple layers or modules in a repository.

Using larger models would also allow us to explore advanced features such as selective recomputation and cross-attention mechanisms in greater depth. These features are essential for maintaining high-quality outputs while optimizing computational efficiency, as evidenced by CacheBlend's success in balancing quality and latency. By integrating larger models into our evaluation framework, we could analyze how well our cache eviction strategies (e.g., LRU, LFU, Composite Cache) perform under higher memory constraints and more diverse workloads. Additionally, larger models would enable us to test the robustness of our system against edge cases, such as cyclic dependencies or highly interconnected modules, which are more prevalent in complex software systems. This expanded scope would provide a more comprehensive validation of our approach and its potential applications in real-world engineering workflows.

6 Conclusion

This project explored the application of CacheBlend-inspired techniques to optimize dependency management in Python repositories, addressing challenges in computational efficiency and storage constraints. By constructing dependency graphs based on mutual import relationships and leveraging advanced cache blending strategies, we demonstrated the potential for significant improvements in managing dependencies dynamically. Our system incorporated various cache eviction policies, including Least Recently Used (LRU), Least Frequently Used (LFU), Least Weighted (LW), and a custom Composite Cache, to maintain optimal cache performance under memory limitations. Testing on both artificial graphs and real-world repositories like `numpy`, `pandas`, and `scikit-learn` highlighted the system's ability to balance computational efficiency with storage requirements, offering a scalable solution for modern software engineering workflows.

The results of our experiments underscore the value of intelligent cache blending in reducing computational overhead and improving retrieval speeds. By adapting CacheBlend principles, which have been proven effective in LLM inference tasks, to the domain of Python dependency management, our system achieved enhanced performance without compromising accuracy or accessibility. The use of simulated dependency graphs allowed us to evaluate the system under controlled conditions, while real-world repositories validated its applicability in practical scenarios. However, as with any experimental setup, limitations such as constrained graph sizes and simplified assumptions about code complexity highlight areas for further exploration.

Looking ahead, this work opens several avenues for future research. Expanding the scale of our evaluations to include larger models and more complex repositories would provide deeper insights into the scalability and robustness of our approach. Additionally, integrating real-time updates into the dependency graph and exploring hybrid cache eviction strategies could further enhance system performance. By building on these foundations, this project contributes to the broader goal of developing dynamic and efficient tools for software dependency management, paving the way for more streamlined workflows in both individual projects and large-scale collaborative environments.

7 References

[1] Yao et al. "CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion".

[2] Herraiz, Israel et al. "On the Distribution of Source Code File Sizes." International Conference on Software and Data Technologies (2011).

[3] Pearson W. R. (2016). Finding Protein and Nucleotide Similarities with FASTA. Current protocols in bioinformatics, 53, 3.9.1–3.9.25. https://doi.org/10.1002/0471250953.bi0309s53