The artifact I chose for this enhancement is my Go-based reverse proxy, a project I originally built in early 2025. At its core, the proxy forwards client requests to the correct backend service, but over time it has grown into a much more capable system. It now includes routing logic through the registry, retry logic with backoff, rate limiting, health checks, and response caching. This milestone focused specifically on strengthening the data structures and algorithms within the caching layer, which lives primarily in cache.go and integrates tightly with the main application logic in app.go and the request flow in handlers.go.

The original cache was an intentionally simple hash map with a TTL on each entry. It worked, but it had clear limitations that made it unsuitable for real-world load. It had no size constraints, no eviction policy, and no awareness of what data was regularly accessed. The cleanup function scanned the entire map on an interval, which meant eviction was essentially linear. There was no way to prevent the cache from consuming too much memory, especially when handling large responses. This made it obvious that the next step for demonstrating mastery in algorithms and data structures was the implementation of a real caching strategy. That's where the LRUCache came in.

The enhanced version replaces the old TTL-only map with a true Least Recently Used (LRU) cache built using the classic combination of a hash map and a doubly linked list. This was implemented directly inside cache.go, and integrating it required revisiting how requests flow through HandleGetRequest in handlers.go. The new design supports O(1) lookups, inserts, and deletions, which is exactly what an efficient cache should provide. The hash map gives constant-time access to nodes, while the doubly linked list maintains usage order by keeping the most recently accessed items at the head and the least recently used at the tail. When the cache reaches its configured byte capacity, the tail node is evicted immediately, also in constant time.

Implementing the LRU cache also meant revisiting how expiration works. Instead of relying only on TTL, the new version supports dual eviction criteria: entries expire either when they surpass their TTL or when memory constraints force an eviction. This created a more realistic caching model; one that behaves closer to what you'd see in production systems like CDN edge caches or web accelerators. The design also tracks the memory footprint of each response so the cache never grows unbounded. This required careful work to ensure that adding, removing, or updating nodes consistently updates the byte counters.

I chose this artifact because it showcases exactly the kind of algorithmic problem-solving expected from the Data Structures and Algorithms course outcomes. Building an LRU cache from scratch, integrating it into a live proxy, ensuring thread safety, and maintaining O(1) operations required a deeper level of design work than anything in the earlier version. The improvement is visible not only in performance but in the architectural quality of the proxy itself. Someone reading the code can now see a clear understanding of linked lists, hashing, time complexity, memory management, and concurrency control.

For the planned course outcomes, the enhancement demonstrates strong use of algorithmic principles by choosing the right data structures to meet performance requirements. It also reflects deliberate design choices, such as how to structure the linked list operations, how to prevent race conditions with proper locking, and how to handle TTL and size-based eviction together. This also strengthened my communication skills, because designing and documenting the cache required explaining why certain structures were chosen and how they interact with the rest of the proxy.

The enhancement process was a real learning experience. Implementing a doubly linked list by hand is always tricky, especially when you also have to maintain head and tail pointers, handle edge cases, and update links without breaking the chain. I had to carefully think through scenarios like removing a node from the middle, moving a node to the head after a cache hit, and evicting from the tail when the cache is full. Ensuring that these operations stayed thread-safe was another challenge, because concurrent reads and writes in a proxy environment can easily lead to subtle race conditions.

The other major challenge was integrating the new cache without breaking the existing request flow. handlers.go uses the cache inside the GET handler, so I had to ensure that I didn't introduce any breaking API changes while completely reworking the underlying structure. Keeping the Get and Store method signatures intact allowed the rest of the system to behave exactly the same from the outside, even though internally the logic was far more sophisticated.

This enhancement strengthened the proxy in a meaningful, production-oriented way. It was a practical application of data structure design inside a real system that handles dynamic routing, retries, and full request lifecycles. This milestone showed the system-level thinking, core algorithm knowledge, and careful reasoning required to meet the Data Structures and Algorithms outcomes in the capstone.