Google Summer of Code 2025 Proposal:
Generic High-Performance Caching Library for etcd

Student: Peter Chang (apullo777@github, MS CS student at Georgia Tech)
Email: peter.yaochen.chang@gmail.com

Abstract

I propose to develop a generic, high-performance caching library for etcd inspired by Kubernetes' watch cache. The goal is to provide etcd with a standardized in-memory cache layer that reduces load and latency by serving many read requests from cache rather than from the etcd store directly. This cache will maintain recent watch events and a cached state of the keyspace, enabling clients to efficiently retrieve data without repeatedly hitting disk or consensus. By the end of the project, we will have a well-tested etcd caching component that matches the Kubernetes watch cache features and can be integrated into etcd-based systems to improve scalability. This project will also be a valuable learning journey into etcd and Kubernetes internals, setting the stage for my long-term contributions to CNCF projects.

Benefits to the Community

Building scalable infrastructure directly on etcd can be challenging without caching. Kubernetes uses a watch cache internally to great effect – it caches recent state and events to serve API clients efficiently. However, this caching mechanism is currently tied to Kubernetes and not available for general etcd usage. Projects like Cilium and Calico's Typha (which use etcd as a datastore) have had to implement custom caches to fill this gap. A generic etcd cache library will benefit the broader community by providing a reusable, production-grade caching solution for any etcd-backed system.

Such a library will significantly reduce etcd load and improve response times for common read operations. For example, Kubernetes recently introduced consistent reads from its watch cache to avoid hitting etcd for every read, yielding a ~30% reduction in API server CPU usage. By bringing similar caching capabilities to etcd itself, we enable other applications to achieve comparable performance gains. This project lowers the barrier for building robust, scalable tools on etcd. It will also foster closer alignment between etcd and Kubernetes: the cache could eventually integrate with etcd-based projects for unified improvements across the cloud native ecosystem.

Deliverables

By the end of GSoC, I will deliver the following:

- **etcd Cache Library (go.etcd.io/cache):** A new Go package in the etcd repository providing core caching primitives analogous to Kubernetes' watch cache. This

includes:

    - *Watch event cache:* Maintains a history of etcd watch events in memory and demultiplexes watch requests (multiple consumers can share one etcd watch).

    - *B-tree state cache:* Maintains the latest state of the key-value store in an in-memory B-tree, updated via watch events. This will serve non-consistent range (list) queries quickly from memory.

    - *Snapshot support:* Ability to take point-in-time snapshots of the B-tree for serving exact stale reads (consistent view of past state).

    - *Consistency and fallback:* Support for consistent reads by ensuring the cache is up-to-date or falling back to etcd when required.

    - *Pluggable serialization:* Hooks for custom encoder/decoder so that applications can control how objects are marshaled in the cache.

    - *Custom indexing:* Extensible indexing mechanism to allow efficient lookups on user-defined keys or fields (similar to Kubernetes indexing by key or label).

- **Robustness and Performance Tests:** A comprehensive test suite, including unit tests, integration tests with an etcd instance, and end-to-end scenarios to ensure cache correctness under various conditions (network partitions, etcd restarts, high update volumes). I will also include robustness tests (fuzzing or fault injection) to validate cache consistency and error handling.

- **Metrics and Benchmarks:** Instrumentation for key metrics (cache hit/miss rates, cache size, update latency, etc.) and a set of benchmarks comparing read/write throughput with and without the cache. This will demonstrate performance improvements and help tune the cache's parameters.

- **Documentation & Examples:** Clear documentation on using the cache library, with examples of integrating it into an etcd client workflow. This includes usage guides, code examples, and recommended best practices for applications (possibly showing integration with a Kubernetes-like client or a Cilium use-case). Well-documented code will ease future maintenance and adoption.

- **Project Report/Blog (if applicable):** I will document my journey and technical decisions, potentially as blog posts or a final report. This will aid future contributors and show transparency in design choices.

All code will be developed in the etcd monorepo under the proposed `go.etcd.io/cache` package (and `.../cache/client` for client interfaces). Deliverables will be considered

complete when they meet the acceptance criteria defined with mentors (e.g. passing all tests, documented, and integrated in a demo scenario).

Detailed Project Timeline

I have structured the timeline to align with the project roadmap milestones (watch cache, B-tree cache, initialization handling, tests, metrics, etc.). My approach is iterative: deliver a minimal working cache early (by midterm) and then build advanced features and robustness in the second half. Below is a week-by-week plan:

- **Community Bonding:**
    - Become deeply familiar with etcd codebase and the Kubernetes watch cache implementation. Set up the development environment and build etcd from source.
    - Engage with the etcd community: introduce myself on mailing lists/Slack, discuss design ideas with mentors, and gather feedback on the proposed approach.
    - Study relevant designs (Kubernetes watch cache, clientv3 watch API, any existing etcd caching attempts) and draft a detailed design document for the cache library.
    - **Milestone:** Design doc reviewed by mentors; ready to start coding.
- **Week 1-2: Watch Cache Implementation – Phase 1**
  Goal: Implement the core watch event cache mechanism.
    - Begin coding the watch events cache component. This involves creating a cache structure that subscribes to etcd watch streams (using clientv3 Watch) and stores a history of recent events (likely in a ring buffer, indexed by revision).
    - Implement request demultiplexing: ensure multiple watch clients can attach to a single etcd watch stream and get the events from the cache. New watchers joining should be able to replay recent history from the buffer so they don't miss events.
    - Handle basic synchronization and memory management for the event buffer (configurable size). Ensure that if the cache overflows or etcd compaction makes the history incomplete, we detect it and fall back appropriately.
    - Write initial unit tests for the watch cache (simulate a stream of put/delete events and verify watchers receive them in order).

    - **Milestone (end of Week 2):** Basic watch cache working for a single key prefix, with tests passing for simple scenarios.

- **Week 3-4: B-Tree State Cache – Phase 2**
  Goal: Develop the in-memory state cache for range queries.

    - Implement a B-tree based cache to store the latest state of etcd (key-value pairs). I may use an existing Go B-tree library (like google/btree) for efficiency and ordered storage of keys.

    - Integrate this with the watch cache: initialize the B-tree by performing an initial Range request on etcd for a given key prefix or the whole keyspace. Then, ensure that each incoming watch event updates the B-tree (on PUT update the value or insert, on DELETE remove the key).

    - Support non-consistent *List* (range) requests by serving from the B-tree. This means when a client requests a range of keys, the library returns data from the B-tree (which reflects the latest known etcd state). We'll document that strictly consistent reads might still require checking etcd, which will be handled later.

    - Begin addressing cache initialization and re-initialization: e.g., if the cache is started fresh or etcd connection is lost, how do we refill the B-tree (possibly snapshotting etcd state again) and manage watch resumption. In this phase, implement a simple strategy for re-sync (like on failure, clear cache and do a fresh list).

    - Write tests for B-tree cache correctness: ensure that after a series of watch events, the B-tree state matches etcd state. Test range queries against direct etcd responses for consistency.

    - **Milestone (end of Week 4):** B-tree cache integrated with watch cache. Able to handle basic list and watch operations entirely from cache, passing tests for add/update/delete scenarios.

- **Week 5: Handling Edge Cases & Midterm Prep**
  Goal: Solidify the core caching functionality and prepare for midterm evaluation.

    - Complete support for requests during cache (re)initialization. For example, if a client issues a read while the cache is still populating or updating after a reconnect, implement logic to either wait for sync or serve a possibly stale result with a warning/flag. Ensure no requests hang indefinitely.

    - Add functionality for consistent reads: if a read request demands the latest etcd state (strong consistency), the library can compare etcd's latest revision to the cache's revision. If the cache is behind, either fetch the missing updates or bypass the cache for that request. (At this stage, a simple

approach might be to always fall back to etcd for explicit consistent reads.)

- Code cleanup and refactoring based on mentor feedback. Ensure code is well-commented and follows etcd project style guidelines.

- **Midterm Evaluation (around July 7):** By this point, core features (watch cache + B-tree cache with basic consistency handling) are expected to be functional. I will prepare a midterm report and possibly a demo for the mentors, showing the cache serving watch and list operations with reduced etcd calls.

- **Week 6-7: Advanced Features – Phase 3**
  Goal: Implement additional features to achieve parity with K8s watch cache and enhance usability.

  - Introduce custom encoder/decoder support in the cache API
    This means designing the cache such that it can store objects in a serialized form (or an application-specific struct) instead of raw etcd `KeyValue`. I will add interfaces or options for clients to provide marshal/unmarshal functions. Write tests with a dummy encoder (e.g., JSON encode values) to verify pluggability.

  - Add custom indexing capability
    I will design a way for clients to specify secondary indices on the cached data (for example, index by a portion of the key or a field in the value). Likely, this could reuse Kubernetes-style index functions. Implement one simple index (like prefix-based grouping) as a proof of concept and ensure the cache can retrieve data via the index efficiently.

  - Continue improving consistent read handling: potentially implement the *exact stale read* feature using B-tree snapshots
    For example, when a read with a specific revision is requested (or to serve stale reads up to a second old), the cache can take a snapshot of the B-tree at a given revision and serve from it. This may involve versioning entries or keeping old copies for a short window. This is an advanced goal; if complexity is high, I will prioritize core features first and treat this as stretch.

  - Start integrating metrics collection. Use Go's `metrics` or `prometheus` client to record cache performance stats (operations served from cache vs etcd, latency, memory usage, etc.). Expose these metrics in a way etcd can scrape or the client can consume.

  - **Milestone (end of Week 7):** Custom encoder/decoder and basic indexing feature implemented. Consistent read logic in place (at least fallback or basic snapshot support). Metrics hooks added. The cache now has feature parity with Kubernetes watch cache on paper, pending final testing.

- **Week 8-9: Testing, Optimization, and Metrics – Phase 4**
  Goal: Hardening the implementation with thorough tests and performance tuning.

    - Write end-to-end tests simulating realistic scenarios. For example, run a local etcd instance, perform high-volume writes, and ensure multiple clients using the cache see correct data. Test failure scenarios: restart etcd in the middle of watching, network delays, etc., to validate cache re-initialization logic.

    - Conduct performance benchmarks comparing throughput with and without cache. For watches: measure how many watch clients we can handle with one etcd stream versus without the cache. For reads: measure latency of serving from cache vs direct etcd under load. Identify bottlenecks (e.g., lock contention in the cache, B-tree performance) and optimize data structures or algorithms as needed.

    - Finalize the metrics collection and ensure they are documented (so users know what is tracked). Possibly create Grafana charts as examples for visualizing cache performance.

    - If any feature from earlier phases is incomplete or needs refinement (e.g., snapshot reads or indexing), address it now based on priority and mentor guidance.

    - **Milestone (end of Week 9):** All major features implemented and tested. Performance results available demonstrating efficiency. Ready to begin final documentation and polishing.

- **Week 10-11: Documentation, Integration & Final Polishing**
  Goal: Complete all documentation and prepare the project for handoff and future use.

    - Write comprehensive user documentation for the cache library: how to initialize it, configure it (buffer sizes, indexing, etc.), and integrate with an etcd client. Include example code and gotchas (e.g., using it within a Kubernetes controller or another etcd-based service).

    - Code polish: Fix any remaining bugs found in tests or during documentation examples. Improve code comments, ensure all public APIs are well-documented (Godoc style).

    - Work with mentors to possibly integrate the cache library into a real scenario for validation. For instance, if time permits, prototype integrating the library as a drop-in replacement in a Kubernetes API server or in a Cilium component to see it working in context. (This can be a simple demo, not full integration.)

    - Buffer some time for unexpected issues or final tweaks based on mentor feedback.

    - **Milestone:** Project ready for final evaluation. All features complete, with documentation and tests merged.

- **Final Week:**

    - Prepare slides or a project summary for the final GSoC submission. If required, publish a blog post about the experience and results.

    - Final mentor evaluations and cleanup of any remaining PRs.

    - **Final Outcome:** A caching library integrated (as an experimental package) in etcd's repository, not yet released until further stabilization per maintainers' processes, but ready for wider testing and future production hardening.

The above timeline is ambitious but achievable. I will maintain regular communication with my mentors, adapting the schedule if needed.

Technical Approach

Architecture: The etcd cache will be implemented as a client-side proxy layer that sits between etcd and etcd clients. It will likely run as part of the client library (or an etcd gateway) so that applications can opt-in to use the cache. The design takes heavy inspiration from Kubernetes' `watchCache` and related mechanisms, but adapted to be a standalone library. Key components of the design include:

- **Watch Event Cache:** I will create a component that wraps etcd's watch API. Instead of every client creating its own watch to etcd, a single watch stream (per key prefix or set of prefixes) is shared. Events from etcd (put, delete) are stored in a fixed-size circular buffer in memory (like Kubernetes' watch cache uses a ring buffer of recent events). Each event will carry its etcd revision. When a new watcher subscribes (e.g., a client wants to watch a prefix), if they request a start revision that is within the buffer's range, the cache can replay those events to the client from memory. If the start revision is older than what's in cache (or the cache is uninitialized), the code will fall back to an etcd *Range* request to get the current state, then resume watching from that point. This design ensures efficient fan-out (one etcd watch feeding many clients) and provides *historical event replay* for late joiners.

- **B-tree State Cache:** In addition to streaming events, the library maintains a snapshot of the latest state of etcd keys in memory using a B-tree structure. The B-tree naturally supports ordered keys and range queries efficiently. On cache initialization, we load the current state via an etcd Range call (which could retrieve thousands of keys) and build the B-tree. Subsequent etcd watch events are applied to this tree (inserts, updates, deletes). For any read (list) request, the cache can serve directly from the B-tree without hitting etcd, which significantly reduces latency and etcd CPU usage. This component essentially mirrors what etcd's state is, within

the constraints of how up-to-date the watch stream is. The B-tree will be keyed by the etcd keys (byte strings) and store values (and metadata like revision).

- **Consistency Management:** etcd provides linearizable reads (via quorum) and serializable reads. Our cache will primarily serve latest known state (which is eventually consistent, typically up-to-date within milliseconds). To support strong consistency when needed, the cache tracks the etcd revision it is at (from the latest applied watch event). If a client requests a linearizable read (or if we want to ensure no stale data), the library can compare etcd's current revision (perhaps by a lightweight etcd call or watch progress notifications) with the cache's revision. If the cache is lagging (for example, watch is temporarily behind), the read can either wait for the next event or perform a direct etcd query. An alternative approach (leveraging Kubernetes' beta feature) is to use etcd's internal mechanisms for *consistent reads from watch cache*, which rely on etcd's raft index – implementing this fully may be beyond GSoC scope, but I'll ensure our design can evolve to that. Additionally, for *exact stale reads*, the cache could keep occasional snapshots (or use copy-on-write on the B-tree) so that a read at an older revision can be satisfied from a snapshot if needed.

- **Customization Hooks:** To make the library broadly useful, I'll include hooks for custom processing:

    - *Encoders/Decoders:* etcd stores byte strings, but many applications (like Kubernetes) encode complex objects (Protobuf/JSON) in those values. I will allow the cache to accept an encoder/decoder so it can store objects in decoded form for direct use, or in encoded form for efficiency. For example, Kubernetes could plug in its object serializer so the cache stores real `Pod` objects, avoiding re-parsing on each read. This design will be optional and pluggable.

    - *Indexers:* The library will offer an interface for clients to define custom indices on the cached data. An index could be as simple as "group by prefix" or something content-based (like an object label). I will likely implement a basic indexing mechanism (e.g., maintain a map from index keys to lists of primary keys) to support quick lookups. This is similar to Kubernetes' `Indexer` in client-go. It will improve usability for high-level apps that need to query the cache in ways other than by primary key.

- **Handling Initialization and Failover:** A critical part of the approach is dealing with the period when the cache is starting up or if it falls out of sync (e.g., etcd connection loss). During initialization, I plan to buffer incoming read requests (or serve them from etcd directly with a note) until the cache has the initial state loaded. Upon etcd reconnect or watch failures (like compaction events where the watch missed too many events), the cache will automatically detect the gap (etcd watch returns a notify) and trigger a re-sync: reload state with a Range call and continue. Throughout this, correctness is paramount – the cache must not serve stale data that violates

etcd's consistency guarantees beyond what's documented.

- **Performance considerations:** The cache will be designed to handle high throughput. This means careful use of concurrency (etcd watch streams run in background goroutines, updates to the B-tree might be serialized through a channel or use fine-grained locks). I will evaluate thread-safety and possibly use read-write locks: many readers should access the B-tree concurrently, while writes (from watch events) are serialized to maintain order. Additionally, memory footprint will be monitored – large clusters might have millions of keys, so we'll make the cache size (or prefixes to cache) configurable. We will also implement size limits (like Kubernetes' watch cache has a fixed event window). By the end, the technical approach aims to produce a cache that is efficient, race-free, and doesn't compromise etcd's consistency guarantees.

This approach is ambitious but builds on known patterns from Kubernetes. By frequent consultation with mentors and referencing Kubernetes code as needed, I will tackle this systematically. The result will be a design that not only works for GSoC but is maintainable by the etcd project long-term.

Personal Background and Skills

I am a graduate student specializing in Computing Systems at Georgia Tech, and I have a passion for distributed systems and storage. Over the past few years, I've built a strong foundation through both academics and hands-on projects:

- **Distributed Filesystems & Caching:** I learned a lot by implementing an AFS-like distributed file system as a course project, where I developed weakly consistent client-side caching along with RPC-based synchronization. This project exposed me to cache coherence challenges and trade-offs between consistency and performance, and I became familiar with strategies for invalidation, versioning, and write-back caching in a distributed context—knowledge that is directly applicable to building an etcd watch cache.

- **Networking and Concurrency in C:** I have implemented several network servers in C, including a multi-threaded web proxy and a caching HTTP proxy server. In these projects, I dealt with high-concurrency, socket programming, and caching of web content. Through these, I've gained low-level insight into performance tuning (e.g., using non-blocking I/O, managing thread pools) and data structures for caches (such as LRU eviction). This experience will be valuable when optimizing the etcd cache for throughput and memory usage.

- **Go Experience:** While my deep systems experience is in C/C++, I have basic experience with Go and have written small projects in it. I am comfortable with Go's syntax, standard library, and concurrency model (goroutines and channels). I am eager to improve my Go skills and have been reading the etcd and Kubernetes Go

code to prepare for this project. My familiarity with C will help me understand Go's performance characteristics (memory management, pointer semantics) when implementing complex structures like a B-tree.

- **Academic Knowledge:** My coursework at Georgia Tech (and self-study) includes operating systems, distributed systems, and networks. I have a solid grasp of concepts like consensus (Raft, which etcd uses), consistency models, and data structure design. This theoretical background means I can quickly understand the nuances of etcd's architecture and the requirements for ensuring correctness in a caching layer on top of a strongly consistent store.

- **Tools and Workflow:** I am proficient with Git, GitHub, Linux command-line, and containerized environments. I can set up test clusters (for example, I have used Docker/Kubernetes locally) which will help in testing the cache. I also have experience writing documentation and diagrams for design docs, which I will use to communicate my approach clearly.

In summary, I offer a strong blend of theoretical knowledge and practical coding experience. While I'm new to the etcd codebase, I'm excited to leverage my background in caching and distributed protocols to contribute effectively and continue learning as I implement this project.

Commitment

I am fully committed to dedicating my time and effort to this project over the summer. Although I might need to take one course during this period, I am confident that I can still contribute 35+ hours per week to the project. I will treat GSoC as a full-time engagement, ensuring that I am available for regular mentor meetings, design discussions, and code reviews.

If selected, I plan to start ramping up immediately: during the community bonding period, I'll work closely with the mentors to refine the project scope and get up to speed. Throughout the coding period, I will maintain consistent communication with the etcd community by posting weekly updates, asking for feedback on designs, and promptly addressing review comments on my pull requests.

I understand that open-source projects span multiple time zones (mentors and contributors may be globally distributed), so I will adjust my work hours when needed to overlap with mentor availability (for instance, I'm currently in the US Eastern time zone but am willing to flex my hours for meetings in EU or IST time zones). Responsiveness and reliability will be my priorities: if blockers arise, I will communicate early and often so we can solve them collaboratively.

Finally, I am prepared to handle the challenges of a large codebase and will not hesitate to seek help or clarification. I've allocated additional buffer time in the timeline for unforeseen issues, and I am confident in my time management to meet all milestones. My university's

summer schedule aligns well with the GSoC period, and I have support from my academic advisors to focus on this project.

Post-GSoC Plans

My journey with etcd and CNCF is just beginning. Beyond GSoC 2025, I plan to continue refining the etcd cache library—addressing issues, adding enhancements, and incorporating feedback from the community. I also hope to make it easier for projects like Kubernetes, Cilium, and other etcd users to adopt this standardized caching solution by developing adapters and clear integration guides.

More importantly, I see this as a chance to become a long-term part of the etcd and CNCF community. I’m excited to learn, contribute, and grow alongside experienced contributors, whether it’s by reviewing PRs, participating in community discussions, or even mentoring future newcomers as my knowledge deepens.

In short, GSoC is my entry point into a community I truly care about, and I'm committed to staying involved long after the summer ends.