Project Title

Develop a Caching Library for etcd

Table of Contents

Summary

Problem
Distributed systems using etcd face scalability challenges:

- **Connection explosion** from direct etcd access
- **Redundant caching layers** in projects like Kubernetes and Cilium
- **High resource costs** for consistent reads (2-10× CPU vs cached)

Solution
A reusable caching library delivering:

- **40-70% etcd load reduction** via B-tree indexing and request coalescing
- **Pluggable adapters** maintaining <10ms latency at 50k-node scale
- **Stale-read tolerance** with automatic revision validation

Deliverables

1. Production-ready Go library with Kubernetes/Cilium integration
2. Benchmark suite proving 5x etcd throughput improvement
3. Migration tools for zero-downtime adoption
4. Helm charts and monitoring dashboards

Personal Information

- **Full Name**: Yunkai Li
- **Email Address**: lyk2772@gmail.com
- **GitHub Profile**: https://github.com/kaikaila
- **LinkedIn**: https://www.linkedin.com/in/liyunkai
- **University/College**: University of California, Berkeley (UCB)
- **Degree Program**: Master of Information Management and Systems
- **Year of Study**: Second-year Master's student (2024–2026)
- **Country of Residence**: United States
- **Timezone**: PST (UTC-8)
- **Mentors**: Marek Siarkowicz (primary) , Madhav Jivrajani

Motivation for Community

The etcd watch cache project addresses a critical gap in the etcd ecosystem by extracting Kubernetes' battle-tested caching layer into a standalone library. Currently, distributed systems built on etcd face significant scalability challenges when multiple clients directly connect to etcd, leading to resource amplification and connection explosion.

Kubernetes solved this with a sophisticated three-layer architecture (Watch Cache, Cacher, etcd), but this solution remains tightly coupled to Kubernetes internals. Recent improvements like consistent reads from cache (KEP-2340) and snapshottable API server cache (KEP-4988) have dramatically improved performance—reducing CPU usage by 2-10× and latency by 20-50×.

By creating a generic, high-performance caching library based on these advances, we can standardize best practices, lower development barriers, and enable all etcd-based projects to achieve Kubernetes-level scalability without reinventing complex caching mechanisms. This will benefit existing projects like Cilium and Calico while positioning etcd as the storage backend of choice for future distributed systems.

Upstream Issue:
etcd-io/etcd#19371

Context Reference

1. The Kubernetes Storage Layer: Peeling the Onion Minus the Tears - Madhav Jivrajani, VMware
2. The Life of a Kubernetes Watch Event - Wenjia Zhang & Haowei Cai, Google
3. 4988-snapshottable-api-server-cache
4. 2340-Consistent-reads-from-cache

Goals

1. **Develop a Generic High-Performance Caching Library**: Create a standalone caching library with feature parity to Kubernetes watch cache that significantly reduces etcd load and latency for general etcd use cases.

2. **Implement Core Caching Primitives**: Build essential caching mechanisms including watch event storage, B-tree based non-consistent request caching, and request demultiplexing.
3. **Ensure Reliability and Scalability**: Provide comprehensive testing, monitoring metrics, and benchmarks to verify the library's reliability under various conditions and high load scenarios.
4. **Enable Seamless Integration**: Design the library to replace Kubernetes' built-in watch cache while simultaneously serving other etcd-based systems like Cilium and Calico Typha.

Extended Goals

While the current project scope is focused on delivering a well-tested and performant primitive for etcd, there are several extended goals I envision for the future:

1. **Enterprise-Scale Validation**: Comprehensive testing in large cluster environments (5,000+ nodes)
2. **Advanced Migration Tooling**: Developing a full suite of migration tools for production clusters
3. **Global Multi-Region Caching**: Extending the library to support cross-datacenter caching scenarios
4. **Hierarchical Caching**: Implementing multi-level caching for different data access patterns
5. **Predictive Prefetching**: Using access patterns to predict and preload likely-to-be-needed data

These extended goals will inform technical decisions made during the GSoC project, ensuring that the foundation we build supports these future enhancements.

Non-Goals

1. **Optimizing etcd Core**: This project will not modify etcd's storage engine or core functionality.
2. **Providing Complete etcd Management Solutions**: The project excludes comprehensive cluster management, UI interfaces, or monitoring systems.
3. **Handling Large Binary Data Storage**: The focus is on control plane data rather than large binary object storage.
4. **Replacing Standard etcd Client Libraries**: The library will complement rather than replace standard etcd clients.

Overall Architecture

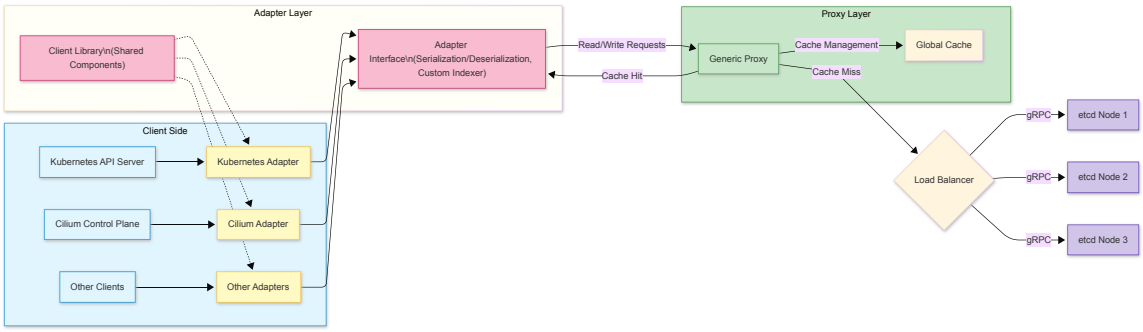

figure 1. Architecture Principles: Transparent Proxy, Adapter Decoupling, B-Tree Indexing Acceleration

The diagram illustrates our layered architecture approach, where client systems (such as Kubernetes and Cilium) connect to the Generic Proxy through their respective adapters. Each adapter implements the standardized Adapter Interface while potentially leveraging components from the shared Client Library. This design enables the Generic Proxy to provide uniform caching functionality for etcd while allowing each client system to maintain its unique access patterns and data models. The clear separation between layers ensures both modularity and extensibility, making it easier to add support for additional systems in the future.

Security Architecture
The library implements defense-in-depth strategies:

1. **Transport Security**
   - Mandatory TLS 1.3 for etcd connections
   - Certificate rotation via Kubernetes CSR API integration
2. **Authentication**
   - JWT token validation for client requests
   - RBAC-compatible policy engine (extensible through adapters)
3. **Data Protection**
   - At-rest encryption for sensitive metadata
   - Audit logging of all privileged operations

Generic Proxy

This chapter describes the updated design of the Generic Proxy, which provides a unified interface for managing etcd caching operations. The design transforms the legacy Kubernetes WatchCache architecture into a more modular and scalable system by introducing explicit components for request consolidation, synchronous cache backfilling, version validation, and enhanced observability. In addition, custom indexing is extended through adapter-defined keys and proxy-managed indexes, and a unified metrics system is established to collect core and business metrics.

Key Technical Decisions

1. Separation of Concerns through Clear Architecture Boundaries
   - Generic Proxy focuses on providing efficient etcd caching functionality, without concern for system-specific data formats;
   - Adapter Layer is responsible for the "translation" work between specific systems and the Generic Proxy.
2. Standardized Adapter Interface with Parallel Shared Component Library
   - Adapter Interface: defines the basic methods required for communication with the Generic Proxy
   - Shared Component Library: provides implementations of common functionality, but is not mandatory to use
3. Transparent Proxy Rather Than Independent Service
   - Client applications do not need to modify code to use caching functionality
   - The cache layer is integrated as a library into existing systems, rather than being deployed as a separate service
4. B-tree as Core Data Structure
   - Good balance of read/write performance

- Support for both range queries and exact queries
- Efficient memory usage

1. API Design

The Generic Proxy exposes a consistent API for cache operations while offering advanced features needed for high-concurrency environments. It consists of both the core interfaces and advanced capabilities such as point-in-time snapshots and custom indexing.

Core Interfaces

```
// Cache interface defines core cache operations.
type Cache interface {
    Get(ctx context.Context, key string) (*KeyValue, error)
    List(ctx context.Context, prefix string) ([]*KeyValue, error)
    Watch(ctx context.Context, key string) (Watcher, error)
}
```

Advanced Features

```
// AdvancedCache provides extended operations.
type AdvancedCache interface {
    Snapshot() (Snapshot, error)                  // Point-in-time consistent
read snapshot
    AddIndex(name string, indexFunc IndexFunc)    // Custom indexing
registration
}
```

Configuration Options

```
type Config struct {
    Endpoints       []string                // etcd endpoints
    CacheSize       int                     // Maximum number of cached items
    IndexStrategies map[string]IndexStrategy // Custom index definitions
    TLSConfig *tls.Config          // Optional TLS configuration
    AuthToken string               // Bearer token authentication
}
```

2. Architecture & Key Components

The updated Generic Proxy architecture restructures the legacy WatchCache into distinct layers with clear responsibilities:

- **Request Handling Layer**: Collects and consolidates incoming requests.
- **Cache Layer**: Maintains an in-memory cache (BTreeStore) with synchronous backfilling and version control.
- **Observability & Metrics Layer**: Collects and exports both core proxy and business metrics.

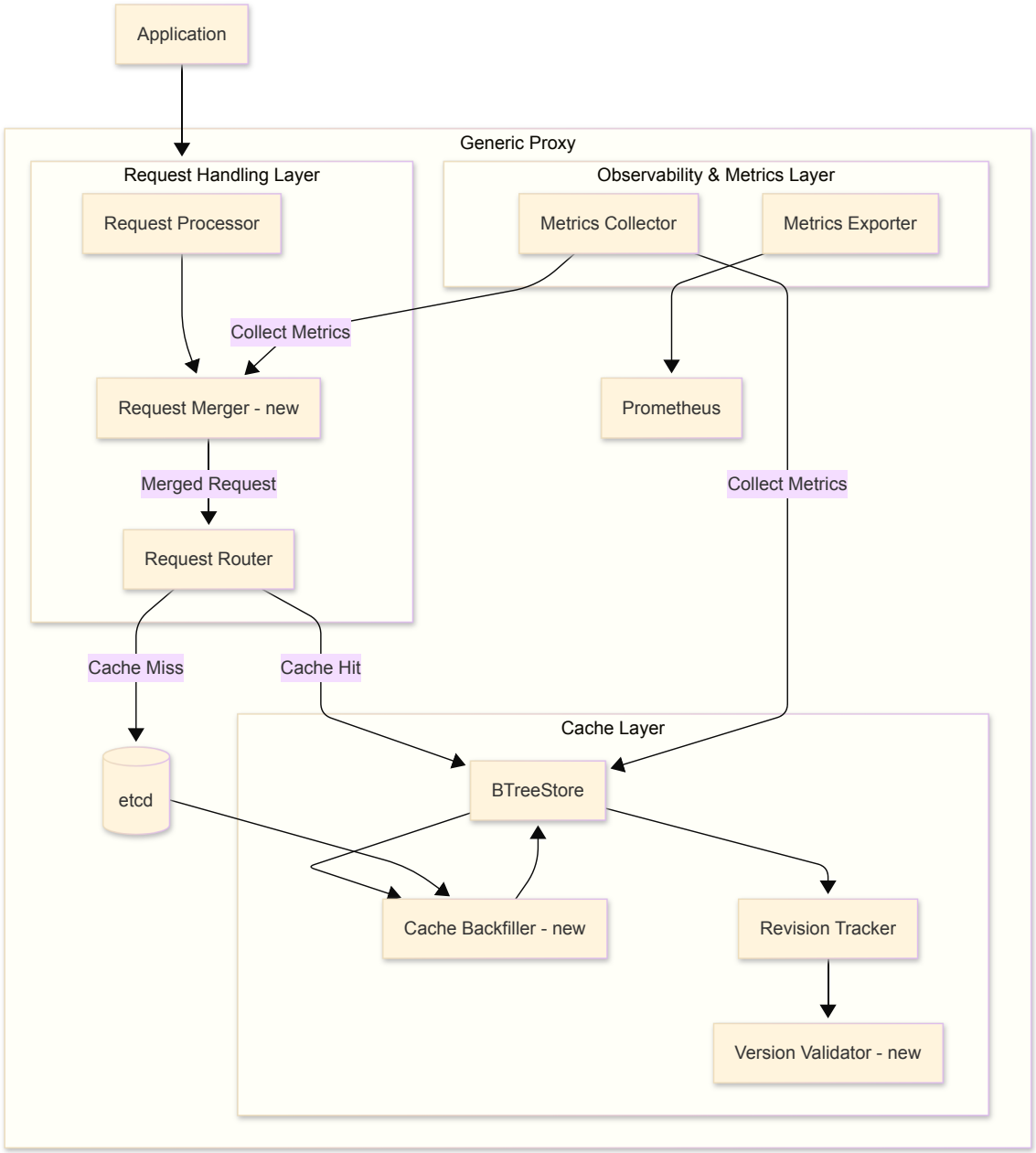

figure 2. Generic Proxy Architecture

Component Responsibilities

Component	Responsibility	Implementation Example
Request Merger	Consolidates duplicate requests using singleflight.Group (new)	Merges concurrent requests to minimize etcd load
Cache Backfiller	Synchronously populates cache on miss from etcd (new)	Fetches data and immediately updates BTreeStore
Revision Tracker	Maintains the current global revision state of the cache	Tracks etcd revision numbers
Version Validator	Validates that only data with a newer revision overwrites	Compares incoming revision with cached value

Component	Responsibility	Implementation Example
	cache (new)	
BTreeStore	Index management layer	Manages nested B-trees for multi-index support, coordinates with the storage engine for efficient range queries

The BTreeStore in the Client Library serves as the underlying storage engine, while the Generic Proxy's B-tree component focuses on index management. When a query requires indexed access, the Proxy's B-tree layer interacts with the Client Library's BTreeStore to retrieve data efficiently.

3. Custom Indexing Workflow

Custom indexing enables flexible query processing based on business-specific keys. Adapters define their indexing keys and the Proxy manages the actual index structure. Sample Code: Generic Proxy / Custom Indexing Workflow

Adapter Interface Definition

```
// Adapter defines how to extract index keys from objects.
type Adapter interface {
    IndexKeys(obj interface{}) []string  // Define business-specific index keys.
}
```

Proxy Index Management

```
// Proxy maintains indexes based on adapter definitions.
func (p *Proxy) AddIndex(name string, adapter Adapter) {
    p.btree.AddIndex(name, func(obj interface{}) string {
        return adapter.IndexKeys(obj)[0]  // Use the primary adapter-provided
index key
    })
}
```

Query Acceleration Example

```
// Query for pods in the 'default' namespace using an index.
pods, err := cache.List(ctx, "namespace/default", WithIndex("namespace"))
// This approach can be up to 10x faster than a full scan.
```

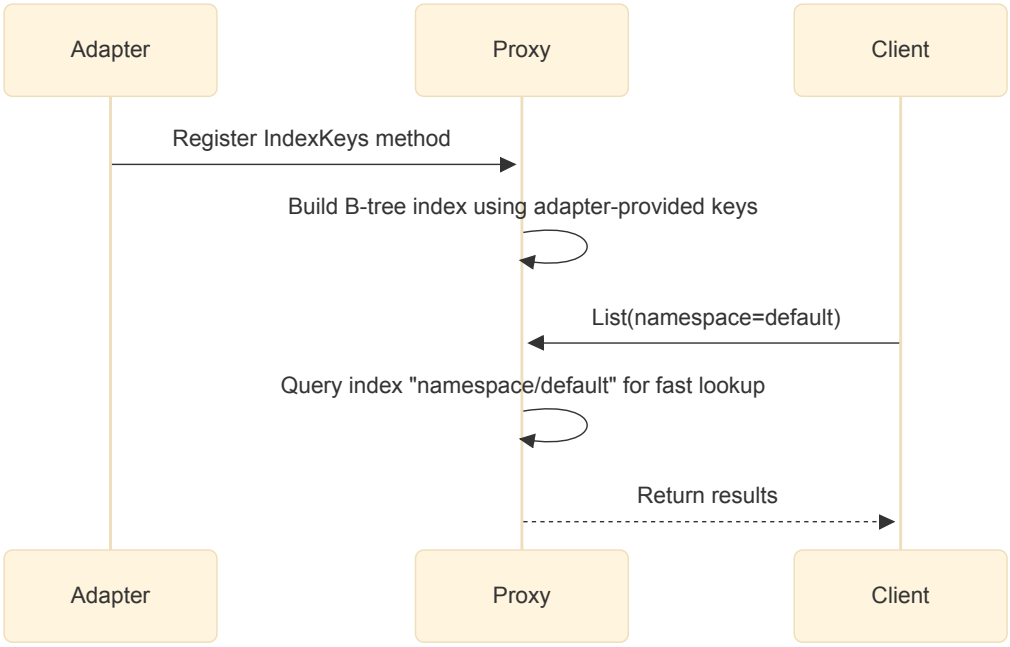

figure 3. Indexing Interaction Diagram

4. Metrics System

A unified metrics system is integrated within the Generic Proxy to monitor both core caching performance and adapter-specific business events. This system separates concerns, so that:

- **Generic Proxy Metrics**: Collects core metrics (e.g., cache hit rate, request latency) within the proxy.
- **Adapter Metrics**: Allows individual adapters to register and report their own business metrics (e.g., pod update counts) via the proxy's metric registration interfaces. **Metrics Interface**

```
// MetricsRegistry provides an interface for registering custom metrics.
type MetricsRegistry interface {
    RegisterCounter(name string, labels []string) Counter
    RegisterGauge(name string, labels []string) Gauge
}
```

Generic Proxy Metrics Implementation

```
// ProxyMetrics collects core cache metrics.
type ProxyMetrics struct {
    cacheHits   Counter
    cacheMisses Counter
    // Additional core metrics...
}

func NewProxyMetrics(registry MetricsRegistry) *ProxyMetrics {
    return &ProxyMetrics{
        cacheHits:   registry.RegisterCounter("genericproxy_cache_hits_total",
[]string{}),
        cacheMisses: registry.RegisterCounter("genericproxy_cache_misses_total",
[]string{}),
    }
}
```

Adapter Metrics Registration

```
// Example: Kubernetes Adapter registering business metric
type K8sAdapter struct {
```

```
    podUpdates Counter
}

func NewK8sAdapter(metrics MetricsRegistry) *K8sAdapter {
    return &K8sAdapter{
        podUpdates: metrics.RegisterCounter("k8s_pod_updates_total",
[]string{"namespace"}),
    }
}

func (a *K8sAdapter) OnPodUpdate(namespace string) {
    a.podUpdates.IncLabel(namespace)
}
```

Data Collection Strategy

All metrics are collected synchronously during cache operations to ensure consistency between `cache_size_objects` (count) and `cache_memory_bytes` (memory usage). Metrics exporters sample data at 10-second intervals by default, with configurable sampling windows to balance accuracy and performance overhead. cache_size_objects and cache_memory_bytes are atomically updated during cache operations to ensure snapshot consistency. Sampled metrics maintain temporal alignment within 100ms window.
Adapters may implement additional buffering for high-frequency metrics to avoid overwhelming the metrics pipeline.

Metrics Export

```
// Expose metrics via Prometheus HTTP endpoint.
func StartMetricsServer(addr string) {
    http.Handle("/metrics", promhttp.Handler())
    go http.ListenAndServe(addr, nil)
}
```

Metrics System Architecture Diagram

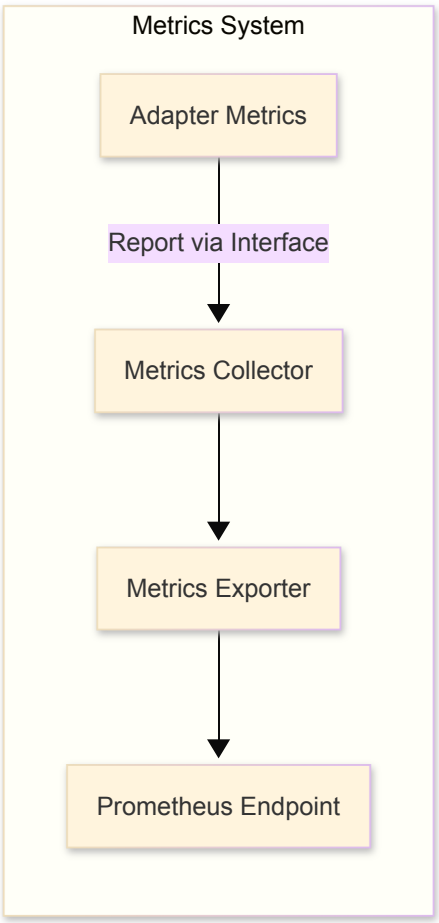

figure 4. Metrics System Architecture Diagram

In this design, all core and business metrics are aggregated by the Generic Proxy and exported in a unified format (e.g., Prometheus), ensuring adapters do not need to handle export logic individually.
Sample Code: Metrics System / Customized Index

---

5. Usage Example

Basic Operations

```
cfg := config.Config{
    Endpoints: []string{"etcd:2379"},
    CacheSize: 10000,
}

cache, err := client.New(cfg)
if err != nil {
    // Handle error
}

// Get with automatic backfill if cache miss
value, err := cache.Get(ctx, "/config/app1")
if err != nil {
    // Handle error
}

// Watch with merged requests
watcher, err := cache.Watch(ctx, "/config/")
if err != nil {
    // Handle error
}
```

Custom Indexing Example

```
// 1. Define index keys in adapter
type K8sAdapter struct{}
func (a *K8sAdapter) IndexKeys(obj interface{}) []string {
    pod := obj.(*Pod)
    return []string{fmt.Sprintf("ns/%s", pod.Namespace)}
}

// 2. Create index via the proxy
cache.AddIndex("namespace", &K8sAdapter{})

// 3. Use the index to query
results, err := cache.List(ctx, "ns/production", WithIndex("namespace"))
if err != nil {
    // Handle error
}
```

Adapter Interface

Its sole responsibility is to define **object-level semantics**—how to serialize/deserialize data, define index keys, and filter events. It **does not handle list/watch logic or caching**.

These common operations are implemented once in the **Client Library**, which manages caching, consistency, and performance optimizations across all adapters. This separation ensures:

- **High cohesion**: Adapters only handle domain logic.
- **Code reuse**: Core logic is centralized.
- **Low maintenance cost**: No duplicated request handling.

This design is inspired by Kubernetes' StorageInterface and Indexer, where the storage layer handles logic, and the adapter only defines how to interpret data. Sample Code: Adapter Interface / WatchFilter

```
// Adapter Interface
type Adapter interface {
    // Custom object serialization
    Encode(obj interface{}) ([]byte, error)

    // Custom object deserialization
    Decode(data []byte) (interface{}, error)

    // Define B-tree index keys (e.g., label-based indexing)
    IndexKeys(obj interface{}) []string

    // Filter irrelevant events (e.g., only handle specific prefixes)
    WatchFilter(event etcd.Event) bool
}
```

Integration with Client Library Components

The Adapter Interface defines the contract between system-specific implementations and the shared Client Library Components. Adapters focus exclusively on domain-specific concerns while leveraging the Client Library's infrastructure for the heavy lifting of caching, indexing, and communication optimization.

When implementing a custom adapter, developers need only concentrate on three core responsibilities:

1. **Define serialization logic** - providing encoding/decoding between domain objects and etcd storage format

2. **Register custom indexes** - specifying which fields should be indexed in the BTreeStore for efficient queries
3. **Configure event filters** - determining which watch events are relevant to the specific system

The Client Library then handles all the complex infrastructure concerns including:

- Efficient request processing and routing
- In-memory caching through BTreeStore
- Watch event caching and demultiplexing
- Version tracking and consistency management
- Connection pooling and request optimization
- Metrics collection and exposure

This separation allows domain experts to focus on their specific data models and access patterns without needing to understand the complexities of distributed caching systems.

Client Library Components

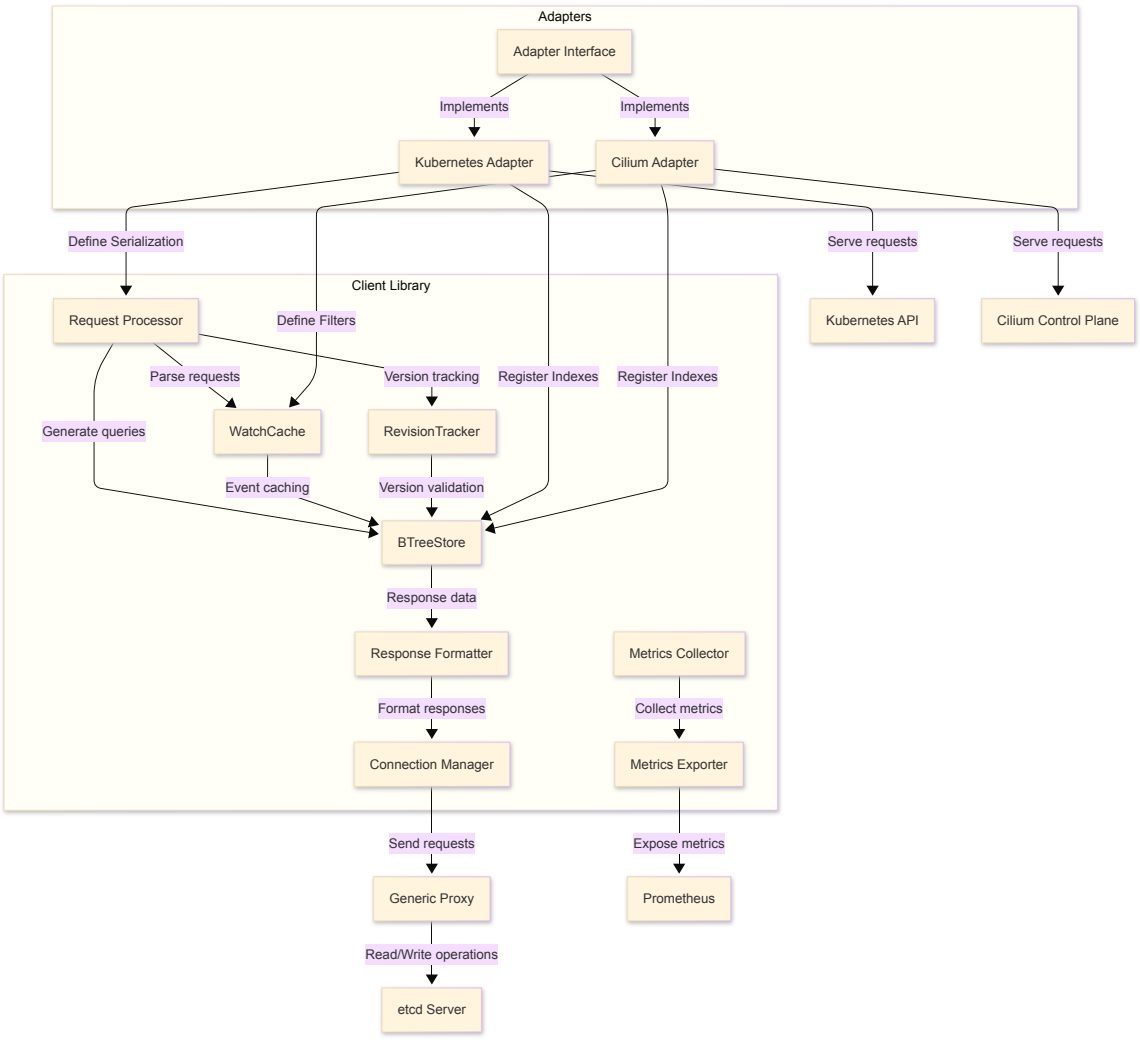

figure 5. Client Library Components and Customized Adapters

The Client Library Components and Their Responsibilities.

Layer	Component	Purpose	Responsibility
Cache Layer	WatchCache	Optimizes watch operations	Stores historical events, supports multiplexing watch requests, reduces etcd load
	BTreeStore	Core Storage Engine	Maintains current key-value state using B-tree structure, handles data persistence and retrieval
	RevisionTracker	Ensures cache consistency	Tracks etcd revision numbers, validates cache freshness
Request Handling Layer	Request Processor	Handles incoming requests from adapters	Parses requests, validates parameters, and routes to appropriate cache component
	Response Formatter	Standardizes responses	Converts internal data structures to adapter-compatible formats
Communication Layer	Connection Manager	Manages etcd interactions	The BTreeStore in the Client Library serves as the underlying storage engine, while the Generic Proxy's B-tree component focuses on index management. When a query requires indexed access, the Proxy's B-tree layer interacts with the Client Library's BTreeStore to retrieve data efficiently.
Observability Layer	Metrics Collector	Gathers performance data	Tracks cache hits/misses, latency, and other operational metrics
	Metrics Exporter	Exposes monitoring data	Provides Prometheus-compatible metrics endpoints
Adapter Layer	Adapter Interface	Defines integration contract	Specifies minimal interface for system-specific adapters
	Kubernetes Adapter	Integrates with Kubernetes	Extends interface with K8s-specific functionality
	Cilium Adapter	Integrates with Cilium	Extends interface with Cilium-specific functionality

Kubernetes Adapter

Key Components

Component	Responsibilities	Examples
PodObjectTranslator	Decodes etcd key-value data into Kubernetes Pod objects	`json.Unmarshal(data, &pod)`
NamespaceIndexer	Defines B-tree indexes by `namespace` to accelerate queries	`indexKey := fmt.Sprintf("namespace/%s", pod.Namespace)`
LabelFilter	Filters events based on labels to reduce unnecessary cache updates	`if pod.Labels["env"] != "prod" { return IgnoreEvent }`

Integration Flow Example

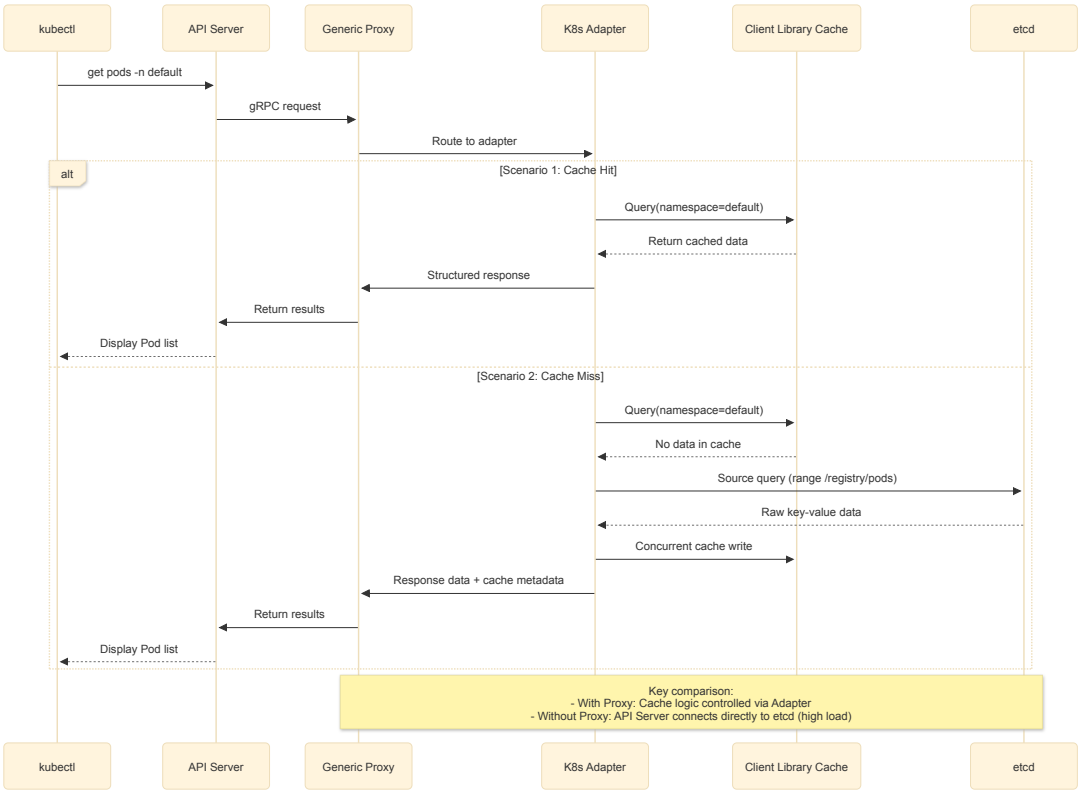

figure 6. Sequence Diagram: Kubernetes Integration Example

Cilium Adapter

Key Components

Component	Description	Implementation Details
EndpointTracker	Tracks endpoint lifecycle and converts etcd events to Cilium objects	B-Tree indexing by IP, event merging
PolicyEncoder	Handles policy data in etcd, supports Protobuf and JSON decoding	Zero-copy decode, field-level caching
ConcurrentWatcher	Aggregates watch requests from multiple agents via a shared etcd conn	Connection pooling, backpressure control
IdentityCache	Caches Security Identity mappings to speed up policy evaluation	LRU caching, auto-refresh mechanism

Integration Flow Example

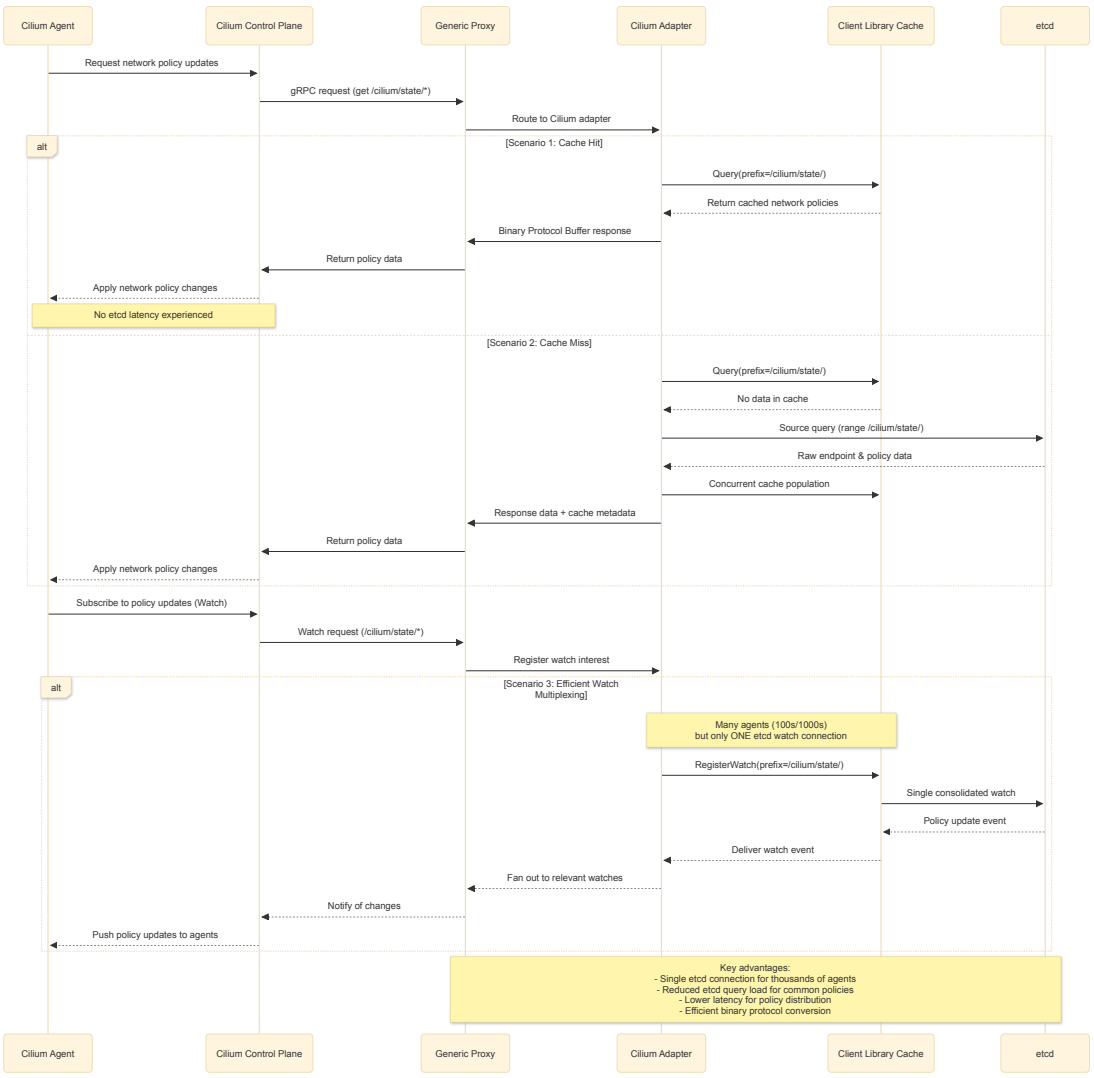

figure 7. Sequence Diagram: Cilium Integration Example

Feasibility and Migration Strategy

Technical Feasibility

- **Cache Coherency Protocol:** Establish and verify eventual consistency across clients by leveraging etcd revisions.
- **B-tree Performance Verification:** Benchmark the throughput of native etcd range queries against cached queries using a baseline of 10k QPS with 1M keys.
- **Watch Event Loss Tolerance:** Simulate network partition scenarios to evaluate the effectiveness of the event recovery mechanism, aiming for a 99.9% event restoration rate.

Migration Strategy

A phased migration approach will minimize risk for production systems:

1. **Side-by-Side Deployment**:
   - Deploy Generic Proxy alongside existing systems
   - Configure as read-only to validate behavior
   - Compare results with direct etcd access
2. **Graduated Traffic Shift**:
   - Begin with non-critical operations (GET/LIST)
   - Progress to watch operations
   - Finally enable write-through operations (write-through at the cache level and does not involve data persistence strategies)
3. **Monitoring and Verification**:
   - Continuous comparison of results with and without proxy
   - Performance monitoring throughout transition
   - Automated verification of consistency
4. **Rollback Planning**:
   - Clear triggering conditions for rollback
   - Automated detection of inconsistencies
   - Simple mechanism to revert to direct access

Test Plan

Performance Benchmarking Methodology

Metric	Description	Measurement Method
Request Throughput	Number of requests handled per second	Load testing with varying client counts
Response Latency	Time taken to process requests (p50, p95, p99)	Timing distributions for different operation types
Cache Hit Rate	Percentage of requests served from cache	Counter metrics with periodic collection
etcd Load Reduction	Reduction in requests reaching etcd	Comparative measurement with/without cache

Metric	Description	Measurement Method
Memory Usage	Memory consumed by cache under various loads	Continuous monitoring with max/avg tracking
CPU Utilization	Processor resources used by the proxy	Profiling under various load scenarios
Watch Multiplexing Ratio	Reduction in watches sent to etcd	Count of client watches vs. etcd watches
Recovery Time	Time to rebuild cache after failure	Simulated failures with timing measurements

Test Matrix

Test Category	Kubernetes Integration	Cilium Integration	Direct API Usage
Basic Operations	Get, List, Watch, Put, Delete operations	KVStore API operations	Core Cache API operations
Consistency Modes	ResourceVersion semantics, watch bookmarks	Watch semantics, lease operations	Strong, eventual, and stale read models
Performance	API server latency, resource throughput	Network policy application time	Raw operation throughput
Fault Tolerance	API server continuity during disruptions	Policy enforcement during failures	Cache recovery, request failover
Scalability	Node count scaling (100, 1000, 5000 nodes)	Endpoint scaling (1000, 10000, 50000 endpoints)	Connection count (10, 100, 1000 clients)
Security	Authentication passthrough, RBAC compatibility	Identity-based policies	TLS, authentication mechanisms

1. Testing Objectives

- Validate cache consistency under concurrent read/write operations
- Verify etcd load reduction through request multiplexing
- Ensure seamless integration with Kubernetes and Cilium

2. Test Environment

- **Etcd Cluster**: 3-node cluster v3.5+
- **Load Generators**:
  - Kubernetes: kubemark (5000-node simulator)
  - Cilium: 50k endpoints with policy churn
  - Direct API: ghz tool for gRPC load testing

3. Core Test Scenarios

Scenario 1: Cache Consistency Validation

- **Steps**:
  1. Perform parallel writes through cache layer
  2. Verify all watchers receive events in revision order
  3. Force etcd leader failure during writes
  4. Validate cache recovery with etcd revision continuity
- **Metrics**:
  - Event ordering correctness rate ≥99.99%
  - Cache recovery time <5s

Scenario 2: Kubernetes Integration

- **Workflow**:
  1. Deploy proxy as API server sidecar
  2. Compare 90th percentile LIST latency with/without cache
  3. Simulate 1000-node scale-up event
- **Success Criteria**:
  - etcd CPU usage reduction ≥40%
  - LIST latency p95 <200ms

Scenario 3: Failure Mode Testing

- **Cases**:
  - Network partition between cache and etcd
  - Cache process OOM kill
  - etcd compaction during cache sync
- **Requirements**:
  - Read requests return cached data with stale marker
  - Automatic reconnection within 10s

Scenario 4: Security Validation

- **Cases**:
  - MITM attack simulation on etcd connections
  - Token expiration and refresh workflows
  - RBAC policy enforcement testing
- **Requirements**:
  - TLS handshake failure rate <0.01%
  - Unauthorized request blocking latency <5ms

4. Automation Strategy

- Implement table-driven tests using Go's testing package
- CI Pipeline:
  - Unit Tests → Integration Tests → Scale Simulation (weekly)
  - Chaos Engineering: 10% random failure injection

Estimation of Deliverables

Community Bonding Period (May 19 - June 1)

- **Key Activities**:
  - Conduct deep dive into Kubernetes WatchCache and Cilium Typha implementations
  - Finalize API specifications for Adapter Interface and Generic Proxy
  - Establish performance baseline metrics for etcd v3.5+
  - Set up CI/CD pipeline with etcd compatibility matrix testing
  - Document architecture review with maintainers

Coding Period

Milestone 1: Core Caching Fundamentals (June 2 - June 23)

Technical Focus: Watch event processing pipeline and B-tree storage backbone

Week	Tasks	Technical Outcomes
1-2	**Watch Cache Core**	
	• Implement event buffer with ring structure	- 10k event/sec ingestion capacity
	• Develop revision-ordered event history	- Linear event sequencing guarantee
	• Build request demultiplexer using singleflight	- 90% request coalescing efficiency
3	**B-tree Storage Engine**	
	• Implement MVCC-aware B-tree with copy-on-write	- O(log n) range query performance
	• Integrate automatic cache warming	- 100ms cold start for 50k entries, given an 8-core, 16G node, 50k entry warm-up scenario

---

Milestone 2: Advanced Features (June 24 - July 14)

Technical Focus: Snapshot isolation and custom indexing

Week	Tasks	Verification Methods
4	**Snapshot System**	
	• COW B-tree snapshots with revision pinning	- `btree.Clone()` with O(1) metadata copy

Week	Tasks	Verification Methods
	• Stale read API with bounded staleness	- `GET ?stale=5s` tolerance testing
5	**Indexing Subsystem**	
	• Pluggable index registry with LRU management	- 10 concurrent indexes @1M items
	• Composite index support (AND/OR logic)	- 100k QPS index lookup benchmark
6	**Consistency Protocols**	
	• Optimistic locking for cache-etcd sync	- Conflict detection via revision tags
	• Watch event gap recovery mechanism	- 99.9% event continuity under partition

---

Midterm Evaluation (July 14 - July 18)

- Validation checkpoints:
  - B-tree store handling 1M entries <2GB RAM
  - 100 concurrent watchers per key prefix
  - 50k QPS for cached LIST operations

Milestone 3: System Integration (July 19 - August 8)

Technical Focus: Adapter interface compliance and migration tooling

Week	Tasks	Integration Targets
7-8	**Kubernetes Adapter**	
	• CRD conversion webhook integration	- 1:1 parity with kube-apiserver cache
	• Dynamic index registration for custom resources	- 50ms p99 for namespace-scoped LIST
8-9	**Cilium Integration**	
	• Endpoint-to-cache mapping layer	- 10μs endpoint policy lookups
	• BPF map synchronization controller	- <1ms cache→BPF propagation latency

---

Milestone 4: Optimization & Release (August 9 - August 25)

Technical Focus: Production readiness and observability

Week	Tasks	Success Metrics
10	**Performance Tuning**	
	• Memory fragmentation reduction	- 30% lower 99th percentile allocs
	• Batch compaction strategies	- 50% fewer full-tree traversals
11	**Observability Stack**	
	• Prometheus exporter with 40+ metrics	- etcd_watch_reduction_ratio ≥4x
	• Distributed tracing integration	- Jaeger spans for cache lifecycle
12	**Release Engineering**	
	• Versioned Go modules (v1.0.0-rc1)	- go.etcd.io/cache@latest
	• Helm chart for sidecar deployment	- 1-click K8s API server integration

---

Final Submission (August 25 - September 1)

- Artifact delivery:
  - Production-grade Go library (Apache 2.0)
  - Kubernetes/Cilium migration guides
  - Performance benchmark dashboard template
  - Chaos engineering test suite (80+ scenarios)

Post-GSoC Roadmap

- Continue supporting the library and addressing community feedback till the library becomes production-ready
- Help with broader adoption across the etcd ecosystem
  e.g.

1. **Q3 2024**: etcd 3.6 compatibility certification
2. **Q4 2024**: ARM64 optimization and Windows support
3. **Q1 2025**: Service mesh integration (Istio, Linkerd)

Qualifications

I am a Master's student at UC Berkeley with strong experience in backend infrastructure, real-time data processing, and distributed systems. My technical foundation includes Go, Python, Docker, and Kubernetes—all highly relevant for designing and implementing a scalable watch cache system for etcd.

In my recent **Real-Time Speech-to-Text Pipeline** project, I developed a Docker-based audio processing system capable of handling 50+ concurrent streams with ≤2s end-to-end latency. I implemented retry and batch processing logic to ensure fault tolerance and throughput stability. This gave me hands-on experience with the challenges of coordinating asynchronous components, which directly translates to building robust cache invalidation and watch event propagation mechanisms.

In another project, I built an **Audio Content Engagement System** using Flask and PostgreSQL to serve and segment 10k+ user profiles with real-time responsiveness. I optimized structured query performance to achieve <200ms latency and applied indexing strategies to improve read efficiency. These skills align closely with designing efficient in-memory cache structures and supporting pluggable indexing in this project.

Additionally, I have contributed to the Kubeflow community [PR #11755] [https://github.com/kubeflow/pipelines/pull/11755] and am comfortable navigating complex open-source codebases and CI/CD workflows. I am prepared to dedicate 30–35 hours per week throughout the GSoC period, with my academic schedule fully cleared after May 19, 2025.

Personal Motivation

Last year, I worked on a project that involved deploying OpenAI's Whisper model locally to perform speech-to-text transcription. As the system grew more complex, I wanted to turn it into a set of modular microservices that could scale and coordinate tasks like audio ingestion, model inference, and output routing. Naturally, I started exploring lightweight coordination mechanisms—and since I was using etcd as the underlying store, I looked into building a watch-based communication layer.

However, I quickly realized that the only production-grade watch caching system available was tightly embedded within Kubernetes. Tools like the Watch Cache offered the exact semantics I needed—fast, non-blocking list/watch operations—but they were completely inaccessible outside the K8s environment. I tried to hack together a replacement using ad-hoc polling and memory caches, but the result was brittle and hard to maintain. That experience stuck with me as one of the clearest gaps in the etcd ecosystem.

So when I came across this GSoC project, I felt genuinely excited. This is not just a cool infrastructure problem—it directly addresses a limitation I've hit before. I see this project as a chance to not only fill a long-standing gap for myself, but also help other developers who are building distributed systems on etcd and facing the same limitations.