Blue Origin: A Foundational CDN for Learning Distributed Systems

Overview

This project builds a foundational Content Delivery Network (CDN) that accelerates web content delivery by caching copies at geographically distributed 'edge' servers. The key architectural challenge is designing a multi-tiered, consistent, and resilient caching system that dramatically reduces latency and origin load while handling real-world complexities like invalidation and request storms.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

Context and Problem Statement

> **Milestone(s):** This section provides the foundational understanding and motivation for the entire CDN project. It underpins the challenges addressed by all five milestones.

Delivering web content at a global scale presents a fundamental engineering challenge. When a user in Tokyo clicks a link hosted on a server in New York, the request and the data must travel across thousands of miles of network infrastructure. This journey is fraught with latency, bottlenecks, and points of failure. A **Content Delivery Network (CDN)** is the distributed systems solution to this problem, transforming the internet's architecture from a hub-and-spoke model into a mesh of local delivery points. Building a CDN is a classic, complex distributed systems problem because it requires elegantly balancing consistency, performance, resilience, and scale across many independent nodes. Before we delve into *how* to build one, we must deeply understand the *why*—the core problem it solves and the architectural constraints it operates within.

Mental Model: The Global Library System

Imagine a single, massive central library (the **Origin Server**) that holds every book (web resource) for a worldwide audience. A reader (the **Client**) in a distant city wants a popular novel. Under the naive model, the reader must request the book from the central library, wait for it to be packaged and shipped across continents, and then receive it. This is slow, expensive in shipping costs (bandwidth), and overwhelms the central library's staff (CPU) and shipping docks (network interface) whenever a book becomes popular.

A CDN is analogous to building a network of local branch libraries (**Edge Caches**) in every major city. When the first reader in Tokyo requests the novel, the local branch fetches a copy from the central library and stores it on its own shelves. When the next reader in Tokyo requests the same book, the branch library can serve it immediately from its local stock—a **Cache Hit**. The experience is dramatically faster, cheaper, and relieves the central library.

This simple analogy breaks down at scale, revealing the true complexity:

- **Shelf Space Management:** Branch libraries have limited shelf space (memory/disk). They must decide which books to keep (**Cache Eviction Policy**) when new ones arrive.
- **Updated Editions:** When the central library releases a revised edition of the novel (the origin content changes), all branch copies become outdated. The system needs a way to notify branches to discard their old copies (**Cache Invalidation**).
- **Rush Hours:** If a famous author announces a new book at noon, thousands of readers might arrive at their local branch at the same moment before any copy is on the shelf. If every branch then simultaneously calls the central library, it will be overwhelmed (**Cache Miss Storm**). A smart system might have regional distribution centers (**Origin Shields**) that coordinate a single fetch and then supply all local branches.
- **Finding Your Nearest Branch:** A reader must be directed to their closest, operational branch (**Geo-Routing** and **Health Checking**), not just the geographically nearest one that might be closed for maintenance.

This project, Blue Origin, is about designing and building the systems and protocols that make this "global library network" intelligent, efficient, and robust.

The Core Delivery Problem

Serving content from a single origin server to a global user base creates three critical, interconnected problems:

1. **Latency (The Speed of Light Problem):** Network latency is governed by physics. A request/response round-trip between Tokyo and New York cannot be faster than the time light takes to travel that distance through fiber optics, which is roughly 100-200 milliseconds for a single trip. When a webpage requires dozens of resources (HTML, CSS, JavaScript, images), this delay compounds, leading to slow page loads. The **propagation delay** is fixed; the only way to reduce it is to shorten the physical distance between the user and the content.
2. **Origin Load & Scalability (The Stampede Problem):** An origin server has finite capacity—CPU, memory, network bandwidth, and I/O. A sudden surge in traffic (a "flash crowd" from a popular news article or a product launch) can easily exceed this capacity, causing the server to slow to a crawl or crash. This is economically inefficient; serving a static image file does not require the origin's expensive application logic, yet it consumes critical resources. The goal is to **offload** repetitive, simple work from the origin.

3. **Availability & Resilience (The Single Point of Failure Problem):** A single origin server (or even a single data center) represents a **single point of failure**. A network partition, a power outage, or a hardware failure in that location can make the service entirely unavailable to all users, everywhere. A distributed system must be designed to **degrade gracefully**, continuing to serve users even when components fail.

The CDN directly attacks these three problems:

- **Against Latency:** By placing **edge nodes** in hundreds of locations (**Points of Presence**, or PoPs) close to users, the physical distance for most requests is reduced to tens of miles, cutting latency from hundreds of milliseconds to single-digit milliseconds.
- **Against Origin Load:** By **caching** (storing) copies of content at the edge, repeated requests for the same resource are served locally without touching the origin. This can reduce origin traffic by 90% or more for cacheable content.
- **Against Availability Issues:** With many distributed edge nodes, the failure of one node affects only a subset of users. Furthermore, cached content can often be served even if the origin is temporarily unavailable (**Serve Stale**), providing a buffer for origin recovery.

The architectural challenge lies in making this distribution appear seamless, consistent, and automatic to both the end user and the content provider.

Existing Approaches & Vocabulary

To ground our design, it's essential to understand the landscape of existing solutions and establish a precise vocabulary.

Approach	Description	Key Characteristics	Relation to Our Project
Commercial CDNs (Cloudflare, Akamai, Fastly)	Full-service, global networks offering caching, security, load balancing, and optimization as a service.	Proprietary, massively scaled software and hardware. Use advanced routing (Anycast), extensive PoPs, and offer rich control APIs.	Our inspiration and conceptual blueprint. We are building a foundational, educational version of their core caching layer.
Open-Source Caching Proxies (Varnish Cache, NGINX, Apache Traffic Server)	Software that can be deployed on a server to act as a reverse proxy with caching capabilities.	Single-node or manually configured clusters. Powerful configuration languages (VCL for Varnish). Often form the building blocks of DIY CDNs.	Our edge cache component will implement logic similar to Varnish's core HTTP caching. We use their design (e.g., surrogate keys) as a specification.
Cloud Provider CDN Services (AWS CloudFront, Google Cloud CDN, Azure CDN)	Managed CDN services integrated with the provider's other cloud services (storage, compute).	Tightly coupled with the provider's ecosystem. Often simpler to set up but less customizable than commercial CDNs.	Illustrates the integration of a CDN with origin services (like object storage).

Core Vocabulary Understanding these terms is crucial for the rest of the document:

- **Origin / Origin Server:** The ultimate source of truth for content. This is the application server (e.g., `api.example.com`) or storage bucket (e.g., `assets.example.com`) that the CDN is protecting and accelerating.
- **Edge Node / Edge Server / PoP (Point of Presence):** A geographically distributed server that caches content and serves requests directly to users. It is the "front line" of the CDN.
- **Cache Hit:** When a user's request can be served entirely from the edge node's local cache. This is the ideal outcome, providing the lowest latency and zero origin load.
- **Cache Miss:** When the requested resource is not in the edge node's cache (or is stale and invalid). The edge node must fetch a fresh copy from upstream (either an origin shield or the origin itself).
- **Origin Shield / Mid-Tier Cache:** An optional caching layer between the edge nodes and the origin. It acts as a "super-edge" or regional aggregator to further reduce requests to the origin and provide request collapsing.
- **Control Plane:** The management and coordination layer of the CDN. It handles configuration distribution, analytics aggregation, and invalidation propagation. It is distinct from the **Data Plane** (the edge nodes) that handles user traffic.
- **TTL (Time-To-Live):** The duration, in seconds, that a cached resource is considered fresh. After this time, it becomes stale and should be revalidated or refetched.
- **Invalidation / Purging:** The active removal of content from the cache before its TTL expires, typically because the origin content has changed.
- **Geo-Routing:** Directing a user's request to the geographically closest (or otherwise optimal) edge node, often based on the user's IP address.

This project synthesizes principles from all these approaches. We take the robust HTTP caching semantics from Varnish and RFC 9111, the distributed architecture concepts from commercial CDNs, and build them into a cohesive, understandable system. The following sections detail the design of each component, starting with the heart of the system: the edge cache.

Goals and Non-Goals

> **Milestone(s):** All milestones (1 through 5)

This section precisely defines the scope of the "Blue Origin" CDN project. For an educational project of this complexity, clear boundaries are essential to maintain focus on the core distributed systems concepts while acknowledging that a production-grade CDN encompasses many additional features. The goals are directly

mapped to the five project milestones, while the non-goals explicitly exclude features that, while important in commercial CDNs, would distract from the foundational learning objectives.

Goals (What We Must Build)

These goals represent the complete set of functional requirements that the "Blue Origin" CDN must implement. Each goal is aligned with one of the five project milestones, ensuring progressive implementation of increasingly sophisticated CDN capabilities.

Milestone 1: Edge Cache Implementation

- **HTTP Caching with RFC 9111 Compliance:** Implement an edge server that acts as a compliant HTTP cache, intercepting client requests and serving cached responses when possible. The cache must respect standard HTTP caching headers (`Cache-Control` , `Expires` , `ETag` , `Last-Modified`).
- **Sophisticated Cache Key Generation:** Construct cache keys that uniquely identify resources by combining the request URL with the values of headers listed in the `Vary` response header (e.g., `Accept-Encoding` , `Accept-Language`). Correctly handle the special case of `Vary: *` , which indicates the resource should never be cached.
- **TTL Hierarchy and Freshness Management:** Implement a multi-layered Time-To-Live (TTL) logic that prioritizes cache directives in the correct order: `s-maxage` (for shared caches like CDNs), `max-age` , and finally the `Expires` header. Maintain metadata for each cache entry indicating its freshness lifetime and current state (fresh, stale).
- **Cache Validation with Conditional Requests:** Support `If-None-Match` (with `ETag`) and `If-Modified-Since` (with `Last-Modified`) conditional requests from clients. When a cached resource is stale, the edge should issue a conditional request to the upstream (shield or origin) and serve a `304 Not Modified` response if the content is unchanged, saving bandwidth.
- **Stale Content Serving Strategies:** Implement `stale-while-revalidate` and `stale-if-error` behaviors, allowing the cache to serve stale content while asynchronously revalidating it with the origin or when the origin is unavailable, improving availability and perceived performance.
- **Memory Management with Eviction Policies:** Implement a bounded cache storage (e.g., based on entry count or memory usage) with a configurable eviction policy—either **LRU (Least Recently Used)** or **LFU (Least Frequently Used)**—to remove the least valuable entries when capacity is exceeded.

Milestone 2: Cache Invalidation

- **Programmatic Purge API:** Expose an administrative HTTP API (e.g., `PURGE /path/to/resource`) that immediately removes a specific cached resource from the edge cache. The next request for that resource must trigger a fresh fetch from the origin.
- **Tag-Based Invalidation with Surrogate Keys:** Support the `Surrogate-Key` response header, allowing the origin to assign one or more tags (keys) to a cached resource. Implement a purge-by-tag API (e.g., `PURGE /purge/tag/KEY`) that invalidates all resources tagged with that key in a single operation, enabling efficient bulk invalidation of related content (e.g., all product pages when a category changes).
- **Pattern-Based Invalidation (Bans):** Implement "ban" functionality that invalidates cache entries matching a pattern (e.g., a URL path prefix like `/api/products/*` or a regular expression). Bans are evaluated lazily when a request matches the pattern, rather than scanning the entire cache upfront.
- **Soft Purge (Graceful Invalidation):** Support a "soft purge" operation that marks content as stale but continues to serve it to clients while asynchronously revalidating it in the background. This provides a smoother user experience compared to a hard purge, which might cause a temporary slowdown as all clients trigger simultaneous cache misses.
- **Distributed Invalidation Propagation:** Design a mechanism for propagating invalidation commands (purges, bans, tag purges) from one edge node to all other edge nodes in the CDN, ensuring eventual consistency of the cache state across the network. This propagation should complete within a configurable time window.

Milestone 3: Origin Shield & Request Collapsing

- **Mid-Tier Caching Layer:** Implement an **origin shield** server that sits between the edge nodes and the origin server. It acts as a shared, secondary cache tier, absorbing repeated requests from multiple edge nodes for the same resource.
- **Request Collapsing (Coalescing):** When the shield receives multiple concurrent requests for the same uncached resource (identical cache key), it should collapse them into a single upstream request to the origin. The resulting response is then broadcast to all waiting requestors, dramatically reducing load on the origin during "cache miss storms" or "thundering herd" scenarios.
- **Request Queuing and Timeout Management:** For collapsed requests, implement a fair queuing mechanism where subsequent requests for the same key wait for the ongoing fetch to complete. This queue must have a timeout shorter than the client's timeout to prevent clients from waiting indefinitely if the upstream fetch hangs.
- **Negative Caching:** Cache error responses (like `404 Not Found` or `5xx` origin errors) with a short, configurable TTL. This prevents a flood of requests to the origin for non-existent or temporarily failing resources.
- **Origin Load Protection:** Implement circuit breakers or concurrency limits at the shield to prevent overwhelming the origin server with too many simultaneous requests, even during massive traffic spikes.

Milestone 4: Edge Node Distribution & Routing

- **Multi-Node Edge Network:** Architect the CDN to consist of multiple, geographically distributed **edge nodes** (Points of Presence). Each node runs the edge caching software independently but coordinates for invalidation and analytics.
- **Geo-Aware Client Routing:** Implement a routing mechanism that directs client requests to the **nearest** healthy edge node based on the client's IP address. This reduces latency by minimizing network distance.

- **Health Checking and Failover:** Implement active health checks between a **control plane** (or routing layer) and edge nodes. Unhealthy nodes should be automatically removed from the routing table, and traffic should be redirected to the next-nearest healthy node, ideally within 5 seconds of failure detection.
- **Consistent Hashing for Cache Distribution:** Use consistent hashing to distribute content keys across the pool of edge nodes. This ensures that requests for a given resource are generally routed to the same edge node, improving cache efficiency. More importantly, it minimizes cache reshuffling (the "thundering herd" effect on origin) when nodes are added or removed, as only a fraction of keys need to be remapped.
- **Control Plane for Coordination:** Implement a simple control plane that edge nodes register with, report health to, and receive configuration/invalidation commands from. This decouples the data plane (serving user requests) from the management plane.

Milestone 5: CDN Analytics & Performance Optimization

- **Cache Performance Analytics:** Track and expose real-time metrics for each edge node and the shield, including cache hit ratio (hits/requests), bandwidth saved, and breakdown by content type or URL pattern.
- **On-the-Fly Content Compression:** Implement response body compression at the edge for text-based content (HTML, CSS, JS, JSON). Support both `gzip` (widely compatible) and `Brotli` (better compression) algorithms, selecting the best based on the client's `Accept-Encoding` header and configurable quality/CPU trade-offs.
- **HTTP Range Request Support:** Fully support the `Range` and `If-Range` request headers for partial content delivery. This is critical for large files (videos, software downloads) and adaptive bitrate streaming. The cache must be able to store, serve, and revalidate partial responses.
- **Advanced Stale-While-Revalidate:** Extend the Milestone 1 implementation with robust background revalidation logic, ensuring that stale content served under `stale-while-revalidate` is eventually updated without impacting client response times.

Non-Goals (What We Explicitly Won't Build)

The following features are commonly found in commercial CDN offerings but are explicitly out of scope for this educational project. This focus allows us to delve deep into core caching, distribution, and invalidation patterns without being overwhelmed by peripheral concerns.

Non-Goal	Reason for Exclusion
Full TLS/SSL Termination & Certificate Management	While HTTPS is mandatory for the modern web, implementing TLS (key exchange, certificate validation, SNI, OCSP stapling) is a complex domain in itself. We assume TLS is handled by a reverse proxy (like NGINX or Caddy) in front of our edge node. Our CDN will work with plain HTTP or behind a TLS terminator.
Distributed Denial of Service (DDoS) Protection	Mitigating large-scale attack traffic requires specialized infrastructure (scrubbing centers, anycast routing, rate limiting at line speed) and threat intelligence. It's a critical production feature but orthogonal to learning caching architecture.
Web Application Firewall (WAF)	Scanning HTTP traffic for SQL injection, XSS, and other OWASP Top 10 vulnerabilities involves complex rule engines and security expertise. It's a valuable add-on layer but not part of the core CDN data plane.
Video Transcoding & Adaptive Bitrate Streaming	Transforming video formats (e.g., to HLS or DASH manifests) requires significant CPU/GPU resources and media expertise. Our CDN will **deliver** video files efficiently (with range request support) but not **transform** them.
Dynamic Content Acceleration (DSA) & API Optimization	Techniques for caching personalized or non-cacheable content (e.g., via Edge Side Includes, sophisticated request/response transformation) involve complex business logic integration. We focus on static and cacheable dynamic content.
Real User Monitoring (RUM) & Synthetic Monitoring	Collecting fine-grained performance data from end-user browsers and running synthetic tests from various locations is a vast analytics domain. Our analytics are server-side focused (hit ratios, bandwidth).
Object Storage Integration as Origin	While many CDNs pull from cloud storage (S3, GCS), implementing the authentication (AWS SigV4) and semantics of various storage backends adds complexity. Our origin is assumed to be a standard HTTP web server.
Multi-CDN Failover & Intelligent Traffic Steering	Switching traffic between different CDN providers based on performance is a strategic layer above a single CDN's operation. We build one CDN, not an orchestrator of many.
Edge Computing (Serverless Functions at Edge)	Executing customer-provided JavaScript/Wasm code at the edge (like Cloudflare Workers) is a paradigm shift from caching. Our edge nodes are pure caching/proxy servers.
Detailed Billing & Usage Reporting	Metering traffic per customer and generating invoices is a business support system, not a core distributed systems challenge.

> **Design Insight:** The choice of non-goals reflects a fundamental learning strategy: depth over breadth. By implementing the data plane of a CDN—caching, distribution, invalidation, shielding—you grapple with the essential problems of state replication, consistency, and fault tolerance in distributed systems. These concepts transfer to countless other domains, whereas the non-goals listed are more specialized verticals (security, media, finance).

Why These Boundaries Matter

Establishing clear non-goals prevents **scope creep**, which is the enemy of a successful educational project. For example:

- **TLS Termination:** Adding TLS would require diving into cryptography libraries, PKI, and potentially performance optimizations like TLS session resumption. This could easily consume more time than the caching logic itself.

- **DDoS Protection:** Building even basic rate limiting is valuable, but advanced DDoS mitigation involves network-level filtering and global threat intelligence feeds, which are impractical to simulate in a learning environment.
- **WAF:** Writing a secure WAF rule engine is a project unto itself and requires deep security knowledge to avoid creating bypass vulnerabilities.

By stating these exclusions upfront, we channel all effort into the **core value proposition of a CDN**: reducing latency and origin load through intelligent caching and geographic distribution. The architecture we design, however, remains compatible with these additional features—they could be added as layers in front of or behind our caching logic, following the Unix philosophy of composable components.

> **Milestone Checkpoint 0 - Scoping Validation:** Before writing any code, verify your understanding of the project's scope. Write a brief document answering: 1) What are the five core capabilities my CDN must have? 2) For each non-goal, what is an alternative (e.g., using an existing tool) to achieve that functionality in a production deployment? This ensures you won't accidentally start implementing features that are out of scope.

Implementation Guidance

This section provides practical starting points for organizing your codebase and implementing the scoped features. Since Python is the primary language, guidance will focus on Pythonic patterns and libraries.

A. Technology Recommendations Table

Component	Simple Option (Getting Started)	Advanced Option (For Exploration)
HTTP Server (Edge/Shield)	`http.server` (standard library) - Simple, built-in.	`aiohttp` (async) - High performance, supports HTTP/1.1 fully.
HTTP Client (to upstream)	`urllib.request` (standard library) - Simple.	`httpx` (sync or async) - Modern, supports HTTP/2, connection pooling.
Cache Storage	`functools.lru_cache` or `dict` with custom eviction - In-memory, simple.	`redis` (via `redis-py`) - External, distributed, persistent.
Data Serialization	`pickle` for caching objects (careful with security).	`msgpack` or `orjson` - Faster, more compact than JSON.
Concurrency Control	`threading.Lock` / `queue.Queue`	`asyncio` with locks and queues - More scalable for I/O.
Health Checks	Simple TCP/HTTP ping in a background thread.	`aiohttp` client with timeout and retry logic.
GeoIP Lookup	`geoip2` library with free MaxMind GeoLite2 database.	Integration with a paid, more accurate GeoIP service API.
Metrics & Analytics	`prometheus_client` - Exposes metrics for Prometheus.	Custom time-series storage (e.g., InfluxDB) or logging to ELK stack.

B. Recommended File/Module Structure

Organize your project from the start to separate concerns and make milestones map to clear code modules.

```
blue_origin_cdn/
├── README.md
├── pyproject.toml (or requirements.txt)
├── src/
│   ├── blue_origin/
│   │   ├── __init__.py
│   │   ├── main.py                    # Entry point, CLI parsing
│   │   ├── config.py                  # Configuration loading (YAML/TOML)
│   │   ├── metrics.py                 # Prometheus metrics setup (Milestone 5)
│   │   ├── control_plane/             # Milestone 4
│   │   │   ├── __init__.py
│   │   │   ├── server.py              # Control plane HTTP API
│   │   │   ├── registry.py            # Edge node registration
│   │   │   └── health_checker.py      # Active health checks
│   │   ├── edge/                      # Milestone 1 & 5
│   │   │   ├── __init__.py
│   │   │   ├── server.py              # Main HTTP request handler
│   │   │   ├── cache/                 # Caching logic core
│   │   │   │   ├── __init__.py
│   │   │   │   ├── storage.py         # CacheStorage class (LRU, etc.)
│   │   │   │   ├── key_generator.py   # Cache key creation
│   │   │   │   ├── ttl_manager.py     # Freshness/expiry logic
│   │   │   │   └── validator.py       # Conditional request handling (304)
│   │   │   ├── compression.py         # Brotli/gzip middleware (Milestone 5)
│   │   │   ├── range_handler.py       # HTTP Range support (Milestone 5)
│   │   │   └── analytics.py           # Hit/miss tracking (Milestone 5)
│   │   ├── invalidation/              # Milestone 2
│   │   │   ├── __init__.py
│   │   │   ├── api.py                 # PURGE/ban HTTP API endpoints
│   │   │   ├── purge.py               # Purge by URL/tag logic
│   │   │   ├── ban.py                 # Ban pattern matching & storage
│   │   │   ├── tags.py                # Surrogate key index management
│   │   │   └── broadcaster.py         # Propagate invalidations (pub/sub)
│   │   ├── shield/                    # Milestone 3
│   │   │   ├── __init__.py
│   │   │   ├── server.py              # Shield HTTP handler (similar to edge)
│   │   │   ├── request_coalescer.py   # Request collapsing & queuing logic
│   │   │   ├── negative_cache.py      # Cache for error responses
│   │   │   └── circuit_breaker.py     # Origin load protection
│   │   ├── routing/                   # Milestone 4
│   │   │   ├── __init__.py
│   │   │   ├── geo_router.py          # Map client IP to nearest edge
│   │   │   ├── consistent_hashing.py  # Consistent hashing ring
│   │   │   └── health.py              # Health check logic (used by edge/shield)
│   │   └── utils/
│   │       ├── __init__.py
│   │       ├── http_utils.py          # Header parsing, date formatting
│   │       └── logging.py             # Structured logging setup
└── tests/                             # Comprehensive test suite
    ├── unit/
    └── integration/
```

C. Infrastructure Starter Code

Here is a complete, reusable utility for parsing HTTP caching headers—a prerequisite for Milestone 1.

PYTHON

```python
# src/blue_origin/utils/http_utils.py

import time

import email.utils

from datetime import datetime, timezone

from typing import Optional, Dict, Tuple, List

from dataclasses import dataclass


@dataclass

class CacheDirectives:

    """Parsed Cache-Control directives relevant to a CDN."""

    s_maxage: Optional[int] = None

    max_age: Optional[int] = None

    no_cache: bool = False

    no_store: bool = False

    must_revalidate: bool = False

    proxy_revalidate: bool = False

    public: bool = False

    private: bool = False

    stale_while_revalidate: Optional[int] = None

    stale_if_error: Optional[int] = None


    @classmethod

    def from_header(cls, cache_control_header: Optional[str]) -> 'CacheDirectives':

        """Parse a Cache-Control header string into a structured object."""

        directives = cls()

        if not cache_control_header:

            return directives


        tokens = [token.strip().lower() for token in cache_control_header.split(',')]

        for token in tokens:

            if '=' in token:

                key, value = token.split('=', 1)

                key = key.strip()

                try:

                    value_int = int(value.strip())

                except ValueError:

                    continue  # Ignore malformed integer values


                if key == 's-maxage':

                    directives.s_maxage = value_int

                elif key == 'max-age':

                    directives.max_age = value_int
```

```
                elif key == 'stale-while-revalidate':
                    directives.stale_while_revalidate = value_int
                elif key == 'stale-if-error':
                    directives.stale_if_error = value_int
            else:
                if token == 'no-cache':
                    directives.no_cache = True
                elif token == 'no-store':
                    directives.no_store = True
                elif token == 'must-revalidate':
                    directives.must_revalidate = True
                elif token == 'proxy-revalidate':
                    directives.proxy_revalidate = True
                elif token == 'public':
                    directives.public = True
                elif token == 'private':
                    directives.private = True
        return directives

def parse_http_date(date_str: Optional[str]) -> Optional[float]:
    """Parse an RFC 1123/822 date string into a Unix timestamp.
    Returns None if the string is invalid or empty.
    """
    if not date_str:
        return None
    try:
        dt = email.utils.parsedate_to_datetime(date_str)
        if dt.tzinfo is None:
            dt = dt.replace(tzinfo=timezone.utc)
        return dt.timestamp()
    except (TypeError, ValueError):
        return None

def is_response_cacheable(
    status_code: int,
    method: str,
    cache_control: CacheDirectives,
    vary_header: Optional[str]
) -> bool:
    """Determine if a response can be stored in the cache per RFC 9111.
```

```
    Args:
        status_code: HTTP status code.
        method: Request method (only GET and HEAD are cacheable by default).
        cache_control: Parsed Cache-Control directives.
        vary_header: Value of the Vary header.

    Returns:
        True if the response is cacheable, False otherwise.
    """
    # Section 3: https://httpwg.org/specs/rfc9111.html#caching.overview
    if method not in ('GET', 'HEAD'):
        return False
    if cache_control.no_store:
        return False
    if cache_control.private:
        return False  # Private responses are not cacheable by shared caches (CDN)
    if vary_header == '*':
        return False  # Vary: * means never cache
    # Generally, only successful responses are cached (200, 203, 206, 300, 301, 308, 404, 405, 410, 414, 501)
    # For simplicity, we'll cache 200, 206, 301, 302, 304 (304 is for validation, not stored), 404, etc.
    cacheable_statuses = {200, 203, 204, 206, 300, 301, 302, 303, 307, 308}
    # Note: 304 Not Modified is a response to a conditional request and does not contain a body to cache.
    if status_code not in cacheable_statuses:
        return False
    return True
```

D. Core Logic Skeleton Code

For the central caching logic (Milestone 1), here is a skeleton for the main request handler method on the edge node.

PYTHON

```python
# src/blue_origin/edge/server.py

from typing import Dict, Optional, Tuple

from http.client import HTTPResponse

from .cache.storage import CacheStorage

from .cache.key_generator import generate_cache_key

from .cache.ttl_manager import TTLManager

from .cache.validator import validate_conditional_request

from ..utils.http_utils import CacheDirectives, parse_http_date, is_response_cacheable


class EdgeRequestHandler:

    def __init__(self, cache_storage: CacheStorage, upstream_url: str):

        self.cache = cache_storage

        self.upstream = upstream_url  # Could be shield or origin URL



    def handle_request(self, request_headers: Dict[str, str], request_body: bytes = b"") -> Tuple[int, Dict[str, str], bytes]:

        """

        Main request handling algorithm for the edge node.



        Args:

            request_headers: Dictionary of HTTP request headers.

            request_body: Raw request body.



        Returns:

            Tuple of (status_code, response_headers, response_body).

        """

        method = request_headers.get(':method', 'GET')

        url = request_headers.get(':path', '/')



        # TODO 1: Generate cache key from request URL and Vary header dimensions.

        #   - Extract headers listed in any existing Vary header from the cache.

        #   - For initial lookup, use a provisional key (just URL). After fetching

        #     from upstream, you'll know the full Vary headers and can store correctly.

        cache_key = generate_cache_key(url, request_headers)



        # TODO 2: Look up the cache key in the cache storage.

        cache_entry = self.cache.get(cache_key)



        if cache_entry:

            # TODO 3: Check if the cached entry is fresh using TTLManager.

            #   - Calculate age from Date header and current time.

            #   - Compare against s-maxage, max-age, Expires.
```

```
        #   - Determine if it's fresh, stale, or needs revalidation.
        freshness_state = TTLManager.check_freshness(cache_entry)

        # TODO 4: If fresh, serve directly from cache (cache hit).
        if freshness_state == 'fresh':
            # TODO 5: Apply any request collapsing for Range headers if present.
            return cache_entry.status, cache_entry.headers, cache_entry.body

        # TODO 6: If stale but allowed by stale-while-revalidate, serve stale
        #   and trigger asynchronous revalidation.
        if freshness_state == 'stale':
            if TTLManager.can_serve_stale(cache_entry):
                # Fire-and-forget background revalidation
                self._revalidate_in_background(cache_key, cache_entry, request_headers)
                return cache_entry.status, cache_entry.headers, cache_entry.body

        # TODO 7: If cache entry exists but is stale and needs validation,
        #   build a conditional request (If-None-Match, If-Modified-Since)
        #   using the cached entry's ETag and Last-Modified.
        conditional_headers = validate_conditional_request(cache_entry, request_headers)
        # Forward to upstream with conditional headers...

    # TODO 8: Cache miss or needs revalidation: forward request to upstream.
    #   - Merge conditional headers if this is a revalidation.
    #   - Use a connection pool for efficiency.
    upstream_response = self._fetch_from_upstream(request_headers, request_body)

    # TODO 9: Parse upstream response headers and check if cacheable.
    cache_control = CacheDirectives.from_header(upstream_response.headers.get('Cache-Control'))
    if is_response_cacheable(upstream_response.status, method, cache_control, upstream_response.headers.get('Vary')):
        # TODO 10: If cacheable, compute final cache key with actual Vary header values.
        final_cache_key = generate_cache_key(url, request_headers, upstream_response.headers.get('Vary'))
        # TODO 11: Store the response in cache with computed TTL.
        self.cache.set(final_cache_key, upstream_response, cache_control)

    # TODO 12: Return the response to the client.
    return upstream_response.status, upstream_response.headers, upstream_response.body

def _revalidate_in_background(self, cache_key: str, cache_entry, request_headers: Dict[str, str]):
    """Asynchronously revalidate a stale cache entry."""
```

```
        # TODO: Implement background thread or async task to revalidate.
        pass

    def _fetch_from_upstream(self, headers: Dict[str, str], body: bytes):
        """Forward request to upstream (shield or origin)."""
        # TODO: Implement HTTP client with connection pooling.
        pass
```

E. Language-Specific Hints (Python)

- **Concurrency:** Use `threading.Lock` for simple mutexes around the cache storage. For request collapsing at the shield, `threading.Condition` is ideal for waiting/notifying threads.
- **Data Structures:** Use `collections.OrderedDict` to implement an LRU cache easily. For LFU, you'll need a more complex structure (e.g., a dict plus a min-heap or `SortedDict`).
- **HTTP Dates:** Always work with UTC timestamps internally. Use `email.utils.formatdate` to generate RFC 1123 dates for responses.
- **Performance:** For header parsing, avoid repeated string splits. Consider caching parsed `CacheDirectives` objects alongside the cached response.
- **Testing:** Use `unittest.mock` to simulate upstream origin responses and test cache behavior without a network.

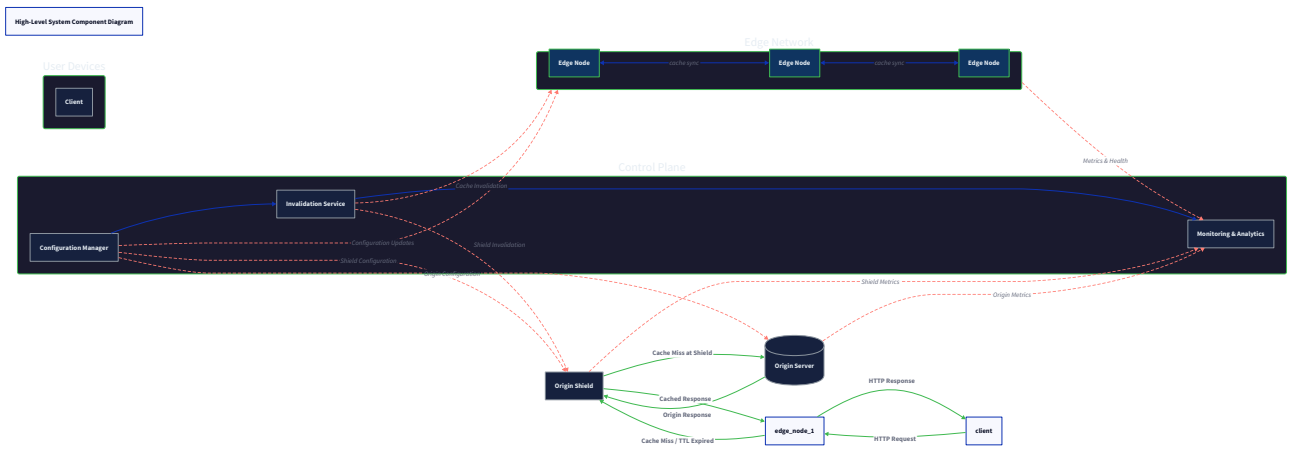
F. Milestone Checkpoint for Goals

After reading this section, you should be able to clearly articulate what you will and won't build. A good checkpoint is to create a simple markdown table in your project notes mapping each milestone to the 2-3 most critical deliverables. For example:

Milestone	Critical Deliverable	Verification Method
1	Cache serves cached response for same URL	`curl` twice, second response has `X-Cache: HIT` header
2	`PURGE /api/purge` removes cached item	`curl` after purge shows `X-Cache: MISS`
3	Two simultaneous requests for same uncached resource result in one origin fetch	Logs show only one upstream request
4	Client IP from US routes to US edge node IP	`dig` or `curl` shows different IPs for different geo-locations
5	Text response is served with `Content-Encoding: gzip`	Response headers show compression

High-Level Architecture

> **Milestone(s):** All milestones (1 through 5)
> This section presents the architectural blueprint for the entire CDN system. Understanding this bird's-eye view is critical before diving into individual component designs, as it establishes the relationships and responsibilities that define how the system works as a whole.



Component Overview & Responsibilities

Think of our CDN as a **global postal system** with specialized facilities at different distances from the end recipient:

Component	Postal System Analogy	Technical Responsibility	Key Data Held
Client	Letter sender/recipient	Initiates HTTP requests for web content; expects fast responses	None (external to CDN)
Edge Node	Local post office	Serves cached content to nearby users; first point of contact	Cached responses, local hit/miss counters
Origin Shield	Regional sorting facility	Aggregates requests from multiple edge nodes; provides secondary cache layer	Shield-level cache, request coalescing state
Origin Server	Central mail repository	Ultimate source of truth for all content; serves uncached resources	Original content, database, application logic
Control Plane	Postal management HQ	Coordinates the network: routing, invalidation, health monitoring, analytics	Node registry, routing rules, invalidation queue

Each component has a clearly defined responsibility that minimizes overlap and creates clean separation of concerns. The **data plane** (Edge Nodes and Origin Shield) handles user traffic, while the **control plane** manages configuration and coordination.

Detailed Component Specifications

Edge Node (PoP - Point of Presence) The edge node is the CDN's frontline — geographically distributed servers that intercept client requests before they travel long distances to the origin. Each edge node operates autonomously but follows centralized policies.

> **Design Principle:** Edge nodes should be stateless in terms of configuration but stateful in terms of cache content. They can operate independently during network partitions, serving cached content even if the control plane is unavailable.

Responsibility	Details	Critical Interfaces
Request Handling	Accepts HTTP/HTTPS connections, parses requests, generates cache keys	`EdgeRequestHandler.handle_request()`
Cache Lookup & Serving	Checks local cache storage, serves fresh content, validates stale entries	Cache storage interface
Cache Population	Fetches missing content from upstream (shield or origin), stores with metadata	`_fetch_from_upstream()`
TTL Enforcement	Respects cache-control directives, marks entries stale when expired	`CacheDirectives` processing
Local Invalidation	Removes entries when purge commands arrive from control plane	Purge API endpoint
Health Reporting	Periodically sends heartbeat and metrics to control plane	Health check endpoint

Origin Shield (Mid-Tier Cache) The origin shield sits between edge nodes and the origin, acting as a **request aggregator** and **secondary cache**. Its primary purpose is to protect the origin from being overwhelmed by cache miss storms—when many edge nodes simultaneously request the same uncached resource.

> **Key Insight:** The shield introduces a small additional latency (typically 10-50ms) but dramatically reduces origin load. For high-traffic sites, this trade-off is almost always worthwhile.

Responsibility	Details	Key Mechanism
Request Collapsing	Deduplicates concurrent identical requests from multiple edge nodes	Request coalescing map with waiters
Secondary Caching	Maintains its own cache (typically larger than edge caches)	Shield-level cache storage
Negative Caching	Briefly caches error responses (404, 503) to prevent origin storms	Short TTL for error status codes
Load Shedding	Queues or rejects requests when origin is overloaded	Adaptive queuing with circuit breakers
Health Monitoring	Probes origin health and bypasses cache when origin is unhealthy	Active health checks

Control Plane The control plane is the CDN's "brain"—it doesn't handle user traffic but ensures the data plane operates correctly and efficiently. It maintains a global view of the network.

Responsibility	Details	Data Structures
Geo-Routing Configuration	Maps client IP prefixes to optimal edge nodes	GeoIP database, routing tables
Health Monitoring	Collects heartbeats from edge nodes and shield	Node registry with health status
Invalidation Propagation	Distributes purge commands to all relevant nodes	Message queue/pub-sub system
Analytics Aggregation	Collects metrics from all nodes for reporting	Time-series database of hits/misses
Consistent Hashing Management	Maintains the distribution ring for cache sharding	Consistent hashing ring state

Origin Server The origin is the authoritative source for content. While not part of the CDN proper, its behavior profoundly affects CDN design.

Characteristic	Implication for CDN Design
Cache-Control Headers	Determines TTLs and cacheability of responses
ETag/Last-Modified Headers	Enables conditional revalidation
Vary Headers	Indicates response varies by certain request headers
Surrogate-Key Headers	Allows tag-based invalidation (custom header)

Recommended File/Module Structure

A well-organized codebase from the start prevents architectural drift and makes the five milestones easier to implement incrementally. Below is the recommended Python package structure, organized by component and responsibility.

```
blue_origin_cdn/                    # Project root
├── pyproject.toml                  # Python project configuration
├── README.md
├── requirements.txt
│
├── src/                            # Source code (installable package)
│   └── blue_origin/
│       ├── __init__.py
│       │
│       ├── data_models/            # Core data structures (Milestone 1)
│       │   ├── __init__.py
│       │   ├── cache_entry.py      # CacheEntry class and related structures
│       │   ├── cache_directives.py # CacheDirectives parsing class
│       │   └── invalidation.py     # PurgeRequest, BanRule, SurrogateKeyIndex
│       │
│       ├── edge/                   # Edge node implementation (Milestones 1, 5)
│       │   ├── __init__.py
│       │   ├── handler.py          # EdgeRequestHandler class
│       │   ├── cache_storage.py    # CacheStorage interface and LRU implementation
│       │   ├── cache_key.py        # Cache key generation logic
│       │   ├── compression.py      # Gzip/Brotli compression (Milestone 5)
│       │   ├── range_requests.py   # HTTP Range support (Milestone 5)
│       │   └── analytics.py        # Local hit/miss tracking
│       │
│       ├── shield/                 # Origin shield (Milestone 3)
│       │   ├── __init__.py
│       │   ├── shield_handler.py   # Shield's request handler
│       │   ├── request_coalescing.py # Request collapsing logic
│       │   └── negative_cache.py   # Negative response caching
│       │
│       ├── control_plane/          # Control plane (Milestones 2, 4)
│       │   ├── __init__.py
│       │   ├── node_registry.py    # Tracks edge nodes and their health
│       │   ├── geo_routing.py      # GeoIP lookup and routing logic
│       │   ├── invalidation_bus.py # Distributes purge commands
│       │   └── hashing_ring.py     # Consistent hashing implementation
│       │
│       ├── http_utils/             # Shared HTTP utilities
│       │   ├── __init__.py
│       │   ├── headers.py          # Header parsing utilities
│       │   ├── cache_validation.py # is_response_cacheable() and helpers
│       │   └── date_parsing.py     # parse_http_date() function
│       │
│       ├── protocols/              # Communication protocols
│       │   ├── __init__.py
│       │   ├── health_check.py     # Health check protocol messages
│       │   └── invalidation_pb.py  # Invalidation protocol (could be protobuf)
│       │
│       └── cli/                    # Command-line interfaces
│           ├── __init__.py
│           ├── edge_node.py        # Starts an edge node server
│           ├── shield_node.py      # Starts a shield node
│           ├── control_plane.py    # Starts the control plane
│           └── admin.py            # Admin commands (purge, ban, stats)
│
├── tests/                          # Comprehensive test suite
│   ├── __init__.py
│   ├── test_edge/
│   ├── test_shield/
│   ├── test_control_plane/
│   └── integration/
│
├── configs/                        # Configuration files
│   ├── edge_config.yaml.example
│   ├── shield_config.yaml.example
│   └── control_plane_config.yaml.example
│
└── scripts/                        # Deployment and utility scripts
    ├── deploy_edge.sh
    ├── load_test.py
    └── geoip_update.py
```

Module Dependencies and Import Structure

The dependency flow follows the logical data flow: edge nodes depend on shared utilities and data models; the control plane is relatively independent; and the shield sits between them.

```
edge/ → http_utils/ → data_models/
shield/ → http_utils/ → data_models/
control_plane/ → protocols/ → data_models/
```

> **Design Decision:** We place the `CacheDirectives` class in `data_models/` rather than `http_utils/` because it's a core data structure used throughout the system, not just an HTTP utility. This makes it available to all components without creating circular dependencies.

Configuration Management

Each component should be configurable via YAML files and environment variables:

PYTHON

```python
# Example configuration pattern for edge node

@dataclass

class EdgeConfig:

    upstream_url: str  # URL of shield or origin

    cache_capacity_mb: int = 1024

    listen_port: int = 8080

    control_plane_url: Optional[str] = None

    geoip_database_path: str = "./data/GeoIP2-City.mmdb"



    @classmethod

    def from_yaml(cls, path: str) -> "EdgeConfig":

        # Load and validate configuration

        pass
```

This structure supports all five milestones:

- **Milestone 1 & 5:** `edge/` and `data_models/` directories
- **Milestone 2:** `control_plane/invalidation_bus.py` and edge invalidation handling
- **Milestone 3:** `shield/` directory
- **Milestone 4:** `control_plane/geo_routing.py` and `hashing_ring.py`
- Shared utilities in `http_utils/` support all milestones

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option (Getting Started)	Advanced Option (Production-Ready)
HTTP Server	Python's `http.server` with threading	`asyncio` + `aiohttp` for async I/O
Cache Storage	`dict` with LRU eviction using `collections.OrderedDict`	`redis` as external cache store
Configuration	Python `dataclasses` with YAML loading	`pydantic` with validation and hot reload
Inter-Component Comm	HTTP REST endpoints	gRPC with Protocol Buffers for lower latency
GeoIP Lookup	Offline MaxMind DB with `geoip2` library	Commercial GeoIP service with SLA
Metrics Collection	In-memory counters exposed via `/metrics` endpoint	`prometheus` client library + Grafana
Message Propagation	Redis Pub/Sub for invalidation events	Apache Kafka for ordered, durable events

B. Complete HTTP Utilities Starter Code

Since HTTP header parsing is a prerequisite for all components but not the core learning goal, here's complete, working code for the foundational utilities:

PYTHON

```python
# src/blue_origin/http_utils/date_parsing.py

"""RFC 1123/822 date parsing utilities for HTTP caching."""

import email.utils

from datetime import datetime

from typing import Optional

def parse_http_date(date_str: str) -> Optional[float]:

    """

    Parse an RFC 1123/822 date string into a Unix timestamp.

    Args:

        date_str: Date string like "Tue, 15 Nov 1994 08:12:31 GMT"

    Returns:

        Unix timestamp (seconds since epoch) or None if parsing fails

    """

    if not date_str:

        return None

    try:

        # email.utils.parsedate_to_datetime handles RFC 1123, 850, and asctime formats

        dt = email.utils.parsedate_to_datetime(date_str)

        return dt.timestamp()

    except (ValueError, TypeError):

        # Log warning in production

        return None

def format_http_date(timestamp: float) -> str:

    """

    Format a Unix timestamp to RFC 1123 string.

    Args:

        timestamp: Unix timestamp in seconds

    Returns:

        RFC 1123 formatted date string

    """

    dt = datetime.utcfromtimestamp(timestamp)

    return dt.strftime("%a, %d %b %Y %H:%M:%S GMT")

# src/blue_origin/http_utils/headers.py
```

```
"""HTTP header parsing and manipulation utilities."""

import re
from typing import Dict, List, Optional, Set

def parse_cache_control_header(header_value: str) -> Dict[str, Optional[str]]:
    """
    Parse Cache-Control header into a dictionary of directives.

    Args:
        header_value: e.g., "public, max-age=3600, stale-while-revalidate=60"

    Returns:
        Dictionary mapping directive names to values (or None for boolean directives)
    """
    directives = {}
    if not header_value:
        return directives

    # Split by comma, trim whitespace
    for part in header_value.split(","):
        part = part.strip()
        if "=" in part:
            key, value = part.split("=", 1)
            directives[key.strip()] = value.strip()
        else:
            directives[part] = None  # Boolean directive like "no-cache"

    return directives

def get_header_values(headers: Dict[str, str], header_name: str) -> List[str]:
    """
    Get all values for a header (handling comma-separated and multiple headers).

    HTTP allows both comma-separated values and multiple headers with the same name.

    Args:
        headers: Dictionary of header name -> value
        header_name: Header name (case-insensitive)

    Returns:
        List of individual header values
```

```
    """
    normalized_name = header_name.lower()
    values = []

    for name, value in headers.items():
        if name.lower() == normalized_name:
            # Split by comma, but be careful of quoted strings
            # Simple implementation for most cases
            values.extend([v.strip() for v in value.split(",")])

    return values

# src/blue_origin/http_utils/cache_validation.py
"""Cache validation logic per RFC 9111."""

from typing import Dict, Optional
from ..data_models.cache_directives import CacheDirectives

def is_response_cacheable(
    status_code: int,
    method: str,
    cache_control: Optional[CacheDirectives],
    vary_header: Optional[str]
) -> bool:
    """
    Determine if a response can be stored in the cache per RFC 9111.

    Args:
        status_code: HTTP status code
        method: HTTP method of request
        cache_control: Parsed CacheDirectives object
        vary_header: Value of Vary header (or None)

    Returns:
        True if the response can be cached
    """
    # RFC 9111 Section 3: Only successful responses to GET are cacheable by default
    if method.upper() != "GET":
        return False

    # Status codes that are cacheable by default (per RFC 9111)
    cacheable_by_default = {200, 203, 204, 206, 300, 301, 308, 404, 405, 410, 414, 501}
```

```
    # Check for explicit no-store directive
    if cache_control and (cache_control.no_store or cache_control.private):
        return False

    # Vary: * means never cache (RFC 9111 Section 4.1)
    if vary_header and vary_header.strip() == "*":
        return False

    # Check if status code is cacheable
    if status_code in cacheable_by_default:
        return True

    # Some status codes are cacheable only with explicit directives
    if status_code in {302, 307}:
        # These are cacheable if explicit freshness info is provided
        if cache_control and (cache_control.max_age is not None or
                              cache_control.s_maxage is not None or
                              cache_control.public):
            return True

    return False
```

C. Core Data Model Skeleton

```python
# src/blue_origin/data_models/cache_directives.py

"""Structured representation of Cache-Control directives."""


from dataclasses import dataclass

from typing import Optional

from ..http_utils.headers import parse_cache_control_header


@dataclass

class CacheDirectives:

    """Parsed Cache-Control directives from HTTP headers."""

    s_maxage: Optional[int] = None  # For shared caches (CDN should use this)

    max_age: Optional[int] = None   # For both shared and private caches

    no_cache: bool = False          # Must revalidate with origin

    no_store: bool = False          # Don't store in cache at all

    must_revalidate: bool = False   # Must revalidate when stale

    proxy_revalidate: bool = False  # Like must-revalidate but only for shared caches

    public: bool = False            # Explicitly cacheable by shared caches

    private: bool = False           # Not cacheable by shared caches

    stale_while_revalidate: Optional[int] = None  # Can serve stale while revalidating

    stale_if_error: Optional[int] = None          # Can serve stale if origin error


    @classmethod

    def from_header(cls, cache_control_header: Optional[str]) -> "CacheDirectives":

        """

        Parse a Cache-Control header string into a structured object.


        Args:

            cache_control_header: Raw Cache-Control header value


        Returns:

            CacheDirectives object with parsed values


        TODO 1: Call parse_cache_control_header() to get dictionary of directives

        TODO 2: Convert string values to appropriate types (ints for numeric directives)

        TODO 3: Set boolean fields based on presence of boolean directives

        TODO 4: Handle edge cases: multiple values, invalid formats, unknown directives

        TODO 5: Ensure s-maxage takes precedence over max-age for CDN purposes

        TODO 6: Handle contradictory directives (e.g., both public and private)

        """

        directives = cls()
```

```
        if not cache_control_header:
            return directives

        # TODO: Implement parsing logic
        parsed = parse_cache_control_header(cache_control_header)

        # TODO: Map parsed directives to fields
        # Example:
        # if "s-maxage" in parsed:
        #     directives.s_maxage = int(parsed["s-maxage"])

        return directives

    def is_cacheable_by_cdn(self) -> bool:
        """
        Determine if this response can be cached by a CDN (shared cache).

        Returns:
            True if CDN can cache this response
        """
        # TODO 1: Check no_store - if True, return False
        # TODO 2: Check private - if True, return False (private means no shared cache)
        # TODO 3: If public is True, return True (explicitly cacheable)
        # TODO 4: If no explicit directives, default is cacheable for GET requests
        # TODO 5: Consider other directives that might affect cacheability
        pass
```

```
# src/blue_origin/data_models/cache_entry.py
"""Core cache entry data structure."""

from dataclasses import dataclass
from typing import Dict, List, Optional
import time

@dataclass
class CacheEntry:
    """Represents a cached HTTP response with metadata."""
    # Primary key components
    key: str                        # Generated cache key
    url: str                        # Original request URL
    vary_headers: Dict[str, str]    # Values for headers in Vary header
```

```
    # Response data
    status_code: int
    headers: Dict[str, str]         # Original response headers
    body: bytes                     # Response body

    # Cache metadata
    fetched_at: float               # Unix timestamp when fetched
    expires_at: float               # Unix timestamp when entry expires
    last_used_at: float             # For LRU eviction
    use_count: int                  # For LFU eviction

    # Invalidation support
    surrogate_keys: List[str]       # Tags for group invalidation

    # Validation support
    etag: Optional[str] = None
    last_modified: Optional[str] = None

    def is_fresh(self, current_time: Optional[float] = None) -> bool:
        """
        Check if cache entry is still fresh.

        Args:
            current_time: Unix timestamp (defaults to time.time())

        Returns:
            True if entry hasn't expired
        """
        if current_time is None:
            current_time = time.time()
        return current_time < self.expires_at

    def is_stale_but_revalidatable(self, current_time: Optional[float] = None) -> bool:
        """
        Check if stale entry can be served during revalidation.

        Args:
            current_time: Unix timestamp
```

```
        Returns:
            True if stale-while-revalidate period hasn't expired
        """
        # TODO 1: Compare current_time to expires_at + stale_while_revalidate
        # TODO 2: Need access to CacheDirectives to know stale_while_revalidate duration
        # TODO 3: Return True if within grace period
        pass
```

D. Edge Request Handler Skeleton

```python
# src/blue_origin/edge/handler.py

"""Main request handling logic for edge nodes."""

import time
from typing import Dict, Tuple, Optional
from ..data_models.cache_directives import CacheDirectives
from ..data_models.cache_entry import CacheEntry

class EdgeRequestHandler:
    """Handles HTTP requests at the edge node."""

    def __init__(self, cache, upstream: str):
        """
        Initialize handler with cache storage and upstream URL.

        Args:
            cache: CacheStorage implementation
            upstream: URL of shield or origin server
        """
        self.cache = cache
        self.upstream = upstream

    def handle_request(self, request_headers: Dict[str, str],
                       request_body: bytes) -> Tuple[int, Dict[str, str], bytes]:
        """
        Main request handling algorithm for the edge node.

        Args:
            request_headers: Dictionary of HTTP request headers
            request_body: Raw request body

        Returns:
            Tuple of (status_code, response_headers, response_body)

        TODO 1: Extract request method and URL from headers
        TODO 2: Generate cache key using URL and Vary header dimensions
        TODO 3: Look up cache entry by key
        TODO 4: If cache hit and fresh: return cached response
        TODO 5: If cache hit but stale: handle revalidation logic
        TODO 6: If cache miss: fetch from upstream using _fetch_from_upstream()
```

```
        TODO 7: If response is cacheable: store in cache with metadata
        TODO 8: Apply compression if requested and appropriate (Milestone 5)
        TODO 9: Update analytics counters (hit/miss)
        TODO 10: Return final response
        """
        # TODO: Implement the full algorithm
        pass

    def _fetch_from_upstream(self, headers: Dict[str, str],
                             body: bytes) -> Tuple[int, Dict[str, str], bytes]:
        """
        Forward request to upstream (shield or origin).

        Args:
            headers: Request headers (may be modified)
            body: Request body

        Returns:
            Tuple of (status_code, response_headers, response_body)

        TODO 1: Add CDN-specific headers (e.g., X-Forwarded-For)
        TODO 2: Make HTTP request to upstream URL
        TODO 3: Handle timeouts and connection errors
        TODO 4: Parse response headers and body
        TODO 5: Extract cache directives and validation headers
        TODO 6: Return response tuple
        """
        pass

    def _revalidate_in_background(self, cache_key: str,
                                  cache_entry: CacheEntry,
                                  request_headers: Dict[str, str]):
        """
        Asynchronously revalidate a stale cache entry.

        Args:
            cache_key: Key of stale cache entry
            cache_entry: The stale cache entry
            request_headers: Original request headers
```

```
        TODO 1: Create conditional request with If-None-Match/If-Modified-Since
        TODO 2: Send request to upstream
        TODO 3: If 304 Not Modified: update entry's expiry time
        TODO 4: If 200 OK: replace entry with new response
        TODO 5: Handle errors (keep stale entry if stale-if-error allows)
        TODO 6: Implement proper async/threading to not block main request
        """
        pass
```

E. Python-Specific Implementation Hints

1. **Concurrency Model:** Use threading for simplicity (`concurrent.futures.ThreadPoolExecutor`) or asyncio for high performance. For educational purposes, threading is recommended initially.
2. **Cache Storage:** Implement `CacheStorage` as an abstract base class with concrete implementations:

```python
from abc import ABC, abstractmethod
from typing import Optional


class CacheStorage(ABC):
    @abstractmethod
    def get(self, key: str) -> Optional[CacheEntry]:
        pass

    @abstractmethod
    def set(self, key: str, entry: CacheEntry) -> None:
        pass

    @abstractmethod
    def delete(self, key: str) -> bool:
        pass


class LRUCacheStorage(CacheStorage):
    def __init__(self, max_size_mb: int):
        self.max_size = max_size_mb * 1024 * 1024
        self.current_size = 0
        self.storage = OrderedDict()  # Maintains insertion order
```

3. **HTTP Date Handling:** Always use UTC for HTTP dates. Python's `email.utils` module handles the complex parsing of various date formats.
4. **Header Case Sensitivity:** HTTP headers are case-insensitive for names but case-sensitive for values. Normalize header names to lowercase when storing but preserve original casing when forwarding to origin.
5. **Memory Management:** For the cache, estimate entry size as `len(body) + sum(len(k) + len(v) for k, v in headers.items())` . Use `sys.getsizeof()` for rough estimates but be aware it doesn't account for referenced objects.
6. **Testing:** Use `unittest.mock` to simulate upstream servers and `pytest` for comprehensive testing. The `httpx` library is excellent for both client and server testing.

Data Model

> **Milestone(s):** Milestone 1 (Edge Cache Implementation), Milestone 2 (Cache Invalidation), Milestone 5 (CDN Analytics & Performance Optimization)

The data model defines the foundational structures that allow the Content Delivery Network (CDN) to store, manage, and track cached content. These structures represent the **"state of the world"** for the caching system—what content is stored, where, for how long, and how it can be invalidated or analyzed. Without a well-defined data model, the system would lack the consistency needed for reliable caching operations, efficient invalidation, and meaningful performance insights. This section details three core aspects: the cache entry itself, structures for invalidation, and structures for analytics.

Core: The Cache Entry

Mental Model: The Cargo Container Think of a `CacheEntry` as a standardized shipping container. Each container has:

- **A unique manifest (key)** that exactly describes its contents and destination.
- **The actual goods (body)** – the response data being delivered.
- **Shipping labels (headers)** that specify handling instructions (like compression, encoding).
- **A customs stamp (metadata)** including loading time, expiration date, and tracking of how often it's been accessed.
- **Cargo tags (surrogate keys)** that group this container with others from the same shipment for batch operations.

Just as a port's efficiency depends on how quickly it can locate and retrieve the right container using its manifest, the **edge cache**'s performance hinges on efficiently storing and retrieving `CacheEntry` objects using their `key`.

The `CacheEntry` is the central data structure stored within the `CacheStorage` of each **edge node** and **origin shield**. It encapsulates a complete HTTP response along with the metadata required for HTTP caching semantics, eviction decisions, and invalidation. Its design directly implements the caching logic specified in RFC 9111.

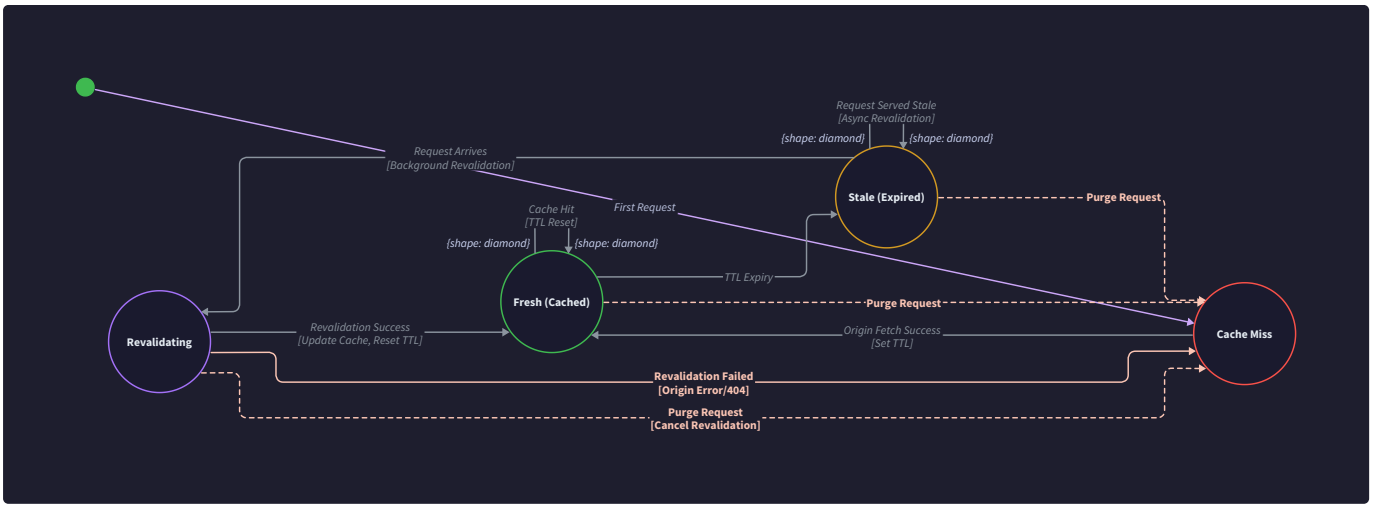
The following table details every field in the `CacheEntry`. These fields collectively enable the four key cache behaviors: freshness validation, revalidation, efficient lookup, and lifecycle management.

Field Name	Type	Description
`key`	`str`	The **primary lookup key** for the cache storage. This is a deterministic string generated from the request URL and the values of headers specified in the `Vary` response header (e.g., `Accept-Encoding`). It uniquely identifies a cached representation of a resource.
`url`	`str`	The original, normalized request URL (including scheme, host, path, and query string). Stored separately from the `key` for debugging, analytics, and potential invalidation by URL pattern.
`vary_headers`	`Dict[str, str]`	A dictionary mapping header names (from the `Vary` header) to their specific values from the *request* that produced this cached response. For a `Vary: Accept-Encoding, User-Agent` response, this dict would contain the `accept-encoding` and `user-agent` values from the request that resulted in this cache entry.
`status_code`	`int`	The HTTP status code of the cached response (e.g., `200`, `404`, `302`). Non-`200` responses (like `301`, `404`, `503`) can also be cached, often with shorter TTLs (**negative caching**).
`headers`	`Dict[str, str]`	The HTTP response headers as received from the upstream (**origin server** or **origin shield**). Critical headers like `Cache-Control`, `ETag`, `Last-Modified`, `Content-Type`, and `Content-Encoding` are stored here verbatim to be sent back to clients.
`body`	`bytes`	The raw, uncompressed response body. Storing the uncompressed form allows the edge to re-encode on-the-fly for clients that support different compression algorithms (e.g., Brotli vs. gzip).
`fetched_at`	`float`	The Unix timestamp (seconds since epoch) when this entry was fetched from the upstream and stored. This is the baseline for calculating freshness based on `max-age` or `s-maxage`.
`expires_at`	`float`	The calculated Unix timestamp after which this entry is considered **stale**. Derived from `fetched_at` plus the effective TTL (prioritizing `s-maxage`, then `max-age`, then `Expires` header).
`last_used_at`	`float`	The Unix timestamp of the last time this entry was used to serve a request. This field is critical for implementing the **LRU (Least Recently Used)** eviction policy.
`use_count`	`int`	A counter of how many times this entry has been used to serve a request. This field supports **LFU (Least Frequently Used)** eviction policies and analytics for "hot" content.
`surrogate_keys`	`List[str]`	A list of **tags** (often called **surrogate keys**) assigned to this entry via the `Surrogate-Key` response header. These allow for efficient, group-based invalidation (e.g., purging all product images when a product is updated).
`etag`	`Optional[str]`	The `ETag` validator from the response headers, stored for efficient conditional revalidation (`If-None-Match`). Can be a strong (`"xyzzy"`) or weak (`W/"xyzzy"`) validator.
`last_modified`	`Optional[str]`	The `Last-Modified` date string from the response headers, stored for conditional revalidation (`If-Modified-Since`). The string format is preserved for direct comparison in future requests.

> **Key Insight:** Separating `key` (for lookup) from `url` (for content identity) is essential. The `key` is a function of both the resource *and* the request characteristics defined by `Vary`. Two requests for the same `url` with different `Accept-Encoding` headers must generate different `keys` and thus distinct `CacheEntry` objects.

Lifecycle and State Transitions A `CacheEntry` moves through distinct states during its lifetime. Understanding these states is crucial for implementing correct cache logic, especially for **stale-while-revalidate** and conditional revalidation.

The following state machine describes these transitions. Refer to the diagram



for a visual representation.

Current State	Event	Next State	Actions Taken
MISS (Not in cache)	Cache lookup for a request	**FETCHING**	1. Generate cache `key`. 2. `key` not found in storage. 3. Initiate upstream request. 4. Create a placeholder to collapse concurrent requests (at the shield).
FETCHING	Upstream response received	**FRESH**	1. Validate response is cacheable via `is_response_cacheable`. 2. Create `CacheEntry` with `fetched_at` and calculated `expires_at`. 3. Store entry in `CacheStorage`. 4. Serve response to waiting client(s).
FRESH	`current_time` < `expires_at`	**FRESH**	1. Serve entry directly to client. 2. Update `last_used_at` and increment `use_count`.
FRESH	`current_time` >= `expires_at`	**STALE**	Entry is now past its freshness lifetime. It may still be served under certain conditions (e.g., `stale-while-revalidate`).
STALE	Request arrives, `CacheDirectives.stale_while_revalidate` period is active	**REVALIDATING**	1. Serve the stale entry immediately to the client. 2. Asynchronously trigger `_revalidate_in_background`. 3. The entry remains **STALE** for subsequent requests until revalidation completes.
STALE	Request arrives, no `stale-while-revalidate` allowed	**FETCHING**	1. Do not serve the stale entry. 2. Treat as a **MISS** and initiate a synchronous upstream fetch.
REVALIDATING	Background revalidation returns `304 Not Modified`	**FRESH**	1. Update `fetched_at` and `expires_at` (resetting the TTL clock). 2. Update `last_used_at`. 3. The entry is now fresh again.
REVALIDATING	Background revalidation returns new `200 OK`	**FRESH**	1. Replace the existing `CacheEntry` with a new one built from the new response. 2. New entry is stored with updated `key` (if headers changed) or overwrites the old one.
Any State	**Purge** command received (by `key` or tag)	**MISS**	1. Entry is removed from `CacheStorage`. 2. Any associated index entries (tags) are cleaned up.

Structures for Invalidation

Invalidation is the active process of removing or marking content as stale in the cache. Efficient invalidation requires auxiliary data structures that map invalidation targets (like tags or URL patterns) to the actual `CacheEntry` objects stored in the cache.

Mental Model: The Library Index Card Catalog Imagine the cache as a library. Each book is a `CacheEntry`. The primary card catalog (the `CacheStorage`) finds books by their primary ID (`key`). An **invalidation index** is like a secondary catalog:

- **Surrogate Key Index**: Like a "subject" catalog. All books about "Dinosaurs" are filed under the "Paleontology" subject card. Purging the "Paleontology" tag means pulling every book linked to that card.
- **Ban List**: Like a librarian's memo to remove "all books published before 1950 from the Fantasy section". It's a rule that is checked when books are accessed or during a cleanup sweep.

These structures enable powerful, granular cache management beyond simple URL purges.

Surrogate Key Index

To enable tag-based purges, the system must maintain an inverted index mapping each surrogate key (tag) to the set of cache keys for entries that contain that tag. This is typically stored in memory for fast lookup.

Field Name	Type	Description
`key_to_tags`	`Dict[str, List[str]]`	Mapping from a cache entry's `key` to its list of `surrogate_keys`. Maintained when a new entry is stored or updated.
`tag_to_keys`	`Dict[str, Set[str]]`	Inverted index mapping a single surrogate key (tag) to the set of cache `key`s that are tagged with it. This is the core structure for `purge_by_tag` operations.

Algorithm for Tag-Based Purge:

1. Admin sends purge command for tag `T`.
2. Look up `tag_to_keys[T]` to get set `S` of cache keys.
3. For each key in `S`: a. Remove the corresponding `CacheEntry` from the primary `CacheStorage`. b. Remove the entry's key from `key_to_tags`.
4. Clear the set `tag_to_keys[T]` (or remove the key `T` entirely).

> **Warning:** This index must be kept consistent with the primary cache. If an entry is evicted due to capacity (LRU), its tags must also be removed from the `tag_to_keys` index. This requires a cleanup hook during eviction.

Ban Rule

A **ban** is a pattern-based invalidation rule (e.g., "all URLs under `/api/v1/products/*`"). Unlike an immediate purge, bans are often evaluated lazily—when a cached entry is accessed, it's checked against active ban rules. This prevents unbounded memory growth from storing huge lists of purged URLs.

Field Name	Type	Description
`pattern`	`str`	A pattern to match against cache entry URLs. For simplicity, we can support prefix matching (e.g., `/api/v1/products/`) or simple glob patterns.
`created_at`	`float`	Timestamp when the ban was created. Allows for automatic expiration of ban rules after a configured TTL to prevent memory leaks.
`is_soft`	`bool`	If `True`, the ban is a **soft purge**: matching entries are marked as stale but not immediately removed. They can be served during `stale-while-revalidate`. If `False`, entries are hard-purged (deleted).

ADR: Ban Rule Storage and Evaluation

> **Decision: Lazy Evaluation of Ban Rules**
>
> - **Context**: Bans can match a very large number of cached entries. Immediately iterating through all cache entries to find matches during ban creation is an O(n) operation that can block request handling, especially in large caches.
> - **Options Considered**:
>   1. **Eager Evaluation**: On ban creation, scan all `CacheEntry` objects and immediately delete/update those that match.
>   2. **Lazy Evaluation**: Store ban rules in a list. Check each rule when a cache entry is *accessed* (on a potential cache hit). If it matches, treat it as a miss (or stale) and optionally delete it.
>   3. **Hybrid with Background Job**: Store ban rules and run a periodic background job that scans and invalidates matching entries.
> - **Decision**: Implement **Lazy Evaluation** (Option 2).
> - **Rationale**:
>   - **Predictable Latency**: Ban creation becomes an O(1) insert operation, avoiding unpredictable delays during administrative actions.
>   - **Simplicity**: Avoids the complexity of a background scanning thread and associated locking.
>   - **Effective for Common Use**: Most banned content will naturally be re-requested by clients. When it is, it will be invalidated. Truly "cold" banned content that is never requested again will eventually be evicted by the LRU policy anyway.
> - **Consequences**:
>   - A banned item may remain in the cache until it is next requested or evicted by LRU. This is **eventual invalidation**.
>   - Adds a small overhead to every cache lookup (must check against all active bans). The list of active bans must be kept small and have a TTL.

Option	Pros	Cons	Chosen?
Eager Evaluation	Immediate consistency. Guaranteed removal.	O(n) scan blocks request handling. Poor performance for large caches.	No
Lazy Evaluation	Fast ban creation. No scanning overhead.	Eventual consistency. Overhead per cache lookup.	**Yes**
Hybrid	Balances immediacy and latency.	Adds system complexity (background jobs, coordination).	No

Structures for Analytics

To operate and optimize a CDN, you must measure its performance. Analytics structures track metrics at various levels: per edge node, per origin shield, per content type, and over time.

Mental Model: The Power Plant Control Room Think of analytics as the gauges and dials in a power plant control room. You need:

- **Instantaneous meters** (like `current_hits`): What's happening right now?
- **Cumulative totals** (like `total_bytes_served`): How much work have we done today?
- **Rate calculations** (like `requests_per_second`): Is demand increasing?
- **Breakdowns by category** (like `hits_by_status_code`): Are failures concentrated in a specific area?

These metrics allow operators to answer critical questions about cache effectiveness, origin load, and bandwidth usage.

The following structures are intended to be in-memory counters that are periodically flushed to a persistent store (like the **control plane**) or reset for rolling time windows (e.g., last 5 minutes).

Edge Node Metrics

Each **edge node** maintains a set of counters to track its own performance.

Field Name	Type	Description
`total_hits`	`int`	Cumulative count of requests served directly from this edge's cache (HTTP status `200` from cache, or `304` from validation).
`total_misses`	`int`	Cumulative count of requests that could not be served from cache and required an upstream fetch.
`total_bandwidth_served`	`int`	Total bytes served to clients from this edge node (includes response bodies from both hits and misses).
`total_bandwidth_upstream`	`int`	Total bytes fetched from upstream (shield or origin) to fulfill misses and revalidations.
`hits_by_status`	`Dict[int, int]`	Breakdown of cache hits by the HTTP status code of the cached response (e.g., `200`, `404`, `301`).
`hits_by_content_type`	`Dict[str, int]`	Breakdown of cache hits by the `Content-Type` of the response (e.g., `image/jpeg`, `text/html`).
`current_cache_size`	`int`	Current number of `CacheEntry` objects stored in this node's `CacheStorage`.
`current_cache_size_bytes`	`int`	Estimated total memory usage (in bytes) of the cache, including `CacheEntry` structures and their bodies.

Shield & Origin Metrics

The **origin shield** (and potentially each edge node, for its upstream) tracks metrics related to origin protection and request collapsing.

Field Name	Type	Description
`collapsed_requests`	`int`	Count of requests that were deduplicated (collapsed) because another identical request was already in flight to the origin. This directly measures the shield's effectiveness in protecting the origin.
`origin_request_queue_size`	`int`	Current number of requests waiting in the shield's queue (for rate limiting or collapsing). A persistently high queue indicates origin overload or insufficient shield capacity.
`origin_errors`	`int`	Count of errors (5xx, network timeouts) received from the origin. A spike triggers health check failures and potential failover.
`negative_cache_hits`	`int`	Count of requests served from cached negative responses (e.g., `404`, `503`).

Time-Windowed Metrics

For real-time dashboards, metrics are often tracked over sliding time windows (e.g., last 1 minute, last 5 minutes). This can be implemented using a circular buffer or bucket-based approach.

Field Name	Type	Description
`window_start`	`float`	Timestamp for the start of the current measurement window.
`window_duration`	`float`	Duration of the window in seconds (e.g., 60.0 for 1 minute).
`buckets`	`List[Dict]`	A list where each element is a metrics snapshot for a sub-interval within the window. For example, for a 5-minute window with 10-second granularity, there would be 30 buckets.
`current_bucket_index`	`int`	Index pointing to the bucket currently being filled.

Algorithm for Updating Time-Windowed Metrics:

1. On each request completion, identify the relevant metrics (e.g., hit, miss, bytes served).
2. Determine the current time bucket based on `window_start`, `window_duration`, and bucket size.
3. If the current time has moved to a new bucket: a. Increment `current_bucket_index` (wrapping around using modulo). b. Reset the count in the new bucket to zero. c. If the new bucket index has wrapped, update `window_start` to reflect the new start time of the oldest bucket.
4. Increment the counters in the current bucket.

This structure allows for efficient calculation of rates (e.g., hits per second over the last minute) by summing the still-relevant buckets.

Common Pitfalls in Data Modeling

⚠️ **Pitfall: Forgetting to Normalize Header Values in `vary_headers`**

- **Description**: When storing the `vary_headers` dictionary, using raw header values as sent by the client (e.g., `gzip, br` vs `gzip, br, deflate`) can lead to unnecessary cache fragmentation. The same semantic value may be expressed differently.

- **Why it's wrong**: Two clients requesting the same resource with semantically identical `Accept-Encoding: gzip` and `Accept-Encoding: gzip, br` (where the server only supports gzip) would create two separate cache entries, wasting space.
- **Fix**: Normalize header values before using them in the `key` and `vary_headers`. For `Accept-Encoding`, parse the value, intersect it with the server's supported encodings, and store a canonical form (e.g., `gzip`). Apply similar normalization for other `Vary` headers like `Accept-Language`.

⚠️ Pitfall: Storing Compressed Bodies in `CacheEntry.body`

- **Description**: Storing the response body exactly as received from upstream (e.g., gzip-compressed).
- **Why it's wrong**: If a new client requests the same resource but only supports Brotli or no compression, the edge must re-fetch from origin or decompress and recompress on-the-fly, losing performance benefits.
- **Fix**: Always store the **uncompressed** body in `CacheEntry.body`. Store the `Content-Encoding` header from the upstream response. When serving a client, check the `Accept-Encoding` request header, and apply the appropriate compression *at the edge* before sending. This is a form of **transcoding** that maximizes cache efficiency.

⚠️ Pitfall: Letting the `tag_to_keys` Index Grow Unbounded

- **Description**: Failing to clean up the inverted index when a `CacheEntry` is evicted from the primary cache (due to LRU) or when its tags change.
- **Why it's wrong**: Memory leak. The index holds references to cache keys that no longer exist. A `purge_by_tag` operation would iterate over stale keys, wasting CPU and potentially causing errors.
- **Fix**: Implement a **cleanup hook** in the `CacheStorage.evict()` method. When an entry is evicted, iterate through its `surrogate_keys` and remove its key from each corresponding set in `tag_to_keys`. If a set becomes empty, you may optionally delete the key from the dictionary.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option (for Learning)	Advanced Option (for Production)
Cache Storage	In-memory dictionary (`dict`) with LRU eviction using `collections.OrderedDict` or a custom linked list.	External cache system like Redis (for distributed edges) or `lru-dict` library for efficient, bounded in-memory cache.
Metrics Storage	In-memory counters, periodically logged to stdout or a local file.	Time-series database (Prometheus) with a push gateway, or streaming metrics to the Control Plane via a lightweight protocol.
Invalidation Index	Nested Python dictionaries (`Dict[str, Set[str]]`) in memory.	Redis Sets and Hashes for persistence and cross-node sharing.

B. Recommended File/Module Structure

```
blue_origin_cdn/
├── pyproject.toml
├── src/
│   └── blue_origin/
│       ├── __init__.py
│       ├── config.py               # EdgeConfig class and loading
│       ├── models/                 # Data models (this section)
│       │   ├── __init__.py
│       │   ├── cache_entry.py      # CacheEntry class & CacheDirectives
│       │   ├── invalidation.py     # SurrogateKeyIndex, BanRule
│       │   └── analytics.py        # EdgeMetrics, ShieldMetrics
│       ├── cache/                  # Milestone 1 & 5
│       │   ├── storage.py          # CacheStorage abstract interface & LRU impl
│       │   └── key.py              # Cache key generation utilities
│       ├── handlers/               # HTTP request handlers
│       │   ├── edge_handler.py     # EdgeRequestHandler
│       │   └── shield_handler.py
│       ├── http_utils/             # RFC 9111 helper functions
│       │   ├── cache_control.py    # parse_cache_control_header, CacheDirectives.from_header
│       │   └── dates.py            # parse_http_date, format_http_date
│       ├── invalidation/           # Milestone 2
│       │   └── manager.py          # Purge, ban, tag invalidation logic
│       ├── analytics/              # Milestone 5
│       │   └── collector.py        # Metrics collection and aggregation
│       └── cli.py                  # Command-line interface for admin tasks
└── tests/
```

C. Infrastructure Starter Code (Cache Directives & HTTP Utilities)

These utilities are prerequisites for parsing and handling HTTP caching headers correctly. They are provided complete and ready to use.

File: `src/blue_origin/http_utils/cache_control.py`

PYTHON

```python
"""
RFC 9111 Cache-Control header parsing utilities.
"""
from typing import Dict, Optional
from dataclasses import dataclass

@dataclass
class CacheDirectives:
    """Structured representation of Cache-Control response directives."""
    s_maxage: Optional[int] = None
    max_age: Optional[int] = None
    no_cache: bool = False
    no_store: bool = False
    must_revalidate: bool = False
    proxy_revalidate: bool = False
    public: bool = False
    private: bool = False
    stale_while_revalidate: Optional[int] = None
    stale_if_error: Optional[int] = None

    @classmethod
    def from_header(cls, cache_control_header: Optional[str]) -> "CacheDirectives":
        """
        Parse a Cache-Control header string into a structured CacheDirectives object.
        Handles multiple directives separated by commas.
        """
        directives = cls()
        if not cache_control_header:
            return directives

        # Split by comma, then by '=' for key-value directives
        for token in cache_control_header.split(","):
            token = token.strip().lower()
            if "=" in token:
                key, val = token.split("=", 1)
                key = key.strip()
                val = val.strip()
                try:
                    int_val = int(val)
                except ValueError:
                    # Ignore non-integer values for known int fields
                    continue
```

```
                if key == "s-maxage":
                    directives.s_maxage = int_val
                elif key == "max-age":
                    directives.max_age = int_val
                elif key == "stale-while-revalidate":
                    directives.stale_while_revalidate = int_val
                elif key == "stale-if-error":
                    directives.stale_if_error = int_val
            else:
                # Boolean directives
                if token == "no-cache":
                    directives.no_cache = True
                elif token == "no-store":
                    directives.no_store = True
                elif token == "must-revalidate":
                    directives.must_revalidate = True
                elif token == "proxy-revalidate":
                    directives.proxy_revalidate = True
                elif token == "public":
                    directives.public = True
                elif token == "private":
                    directives.private = True
        return directives

def parse_cache_control_header(header_value: Optional[str]) -> Dict[str, Optional[str]]:
    """
    Parse Cache-Control header into a dictionary of directive names to values.
    Returns a dict where keys are directive names and values are either
    the parameter (as string) or None for boolean directives.
    """
    result = {}
    if not header_value:
        return result
    for token in header_value.split(","):
        token = token.strip()
        if "=" in token:
            key, val = token.split("=", 1)
            result[key.strip().lower()] = val.strip()
        else:
            result[token.lower()] = None
```

```
    return result
```

File: `src/blue_origin/http_utils/dates.py`

```python
"""
HTTP date parsing and formatting (RFC 1123 / 822).
"""
import time
from email.utils import parsedate_to_datetime
from datetime import datetime, timezone
from typing import Optional

def parse_http_date(date_str: Optional[str]) -> Optional[float]:
    """
    Parse an RFC 1123/822 date string into a Unix timestamp (float seconds since epoch).
    Returns None if the string cannot be parsed.
    """
    if not date_str:
        return None
    try:
        # parsedate_to_datetime handles most common HTTP date formats
        dt = parsedate_to_datetime(date_str)
        # Ensure it's timezone-aware (UTC)
        if dt.tzinfo is None:
            dt = dt.replace(tzinfo=timezone.utc)
        else:
            dt = dt.astimezone(timezone.utc)
        return dt.timestamp()
    except (ValueError, TypeError):
        return None

def format_http_date(timestamp: float) -> str:
    """
    Format a Unix timestamp to RFC 1123 string.
    """
    # Use UTC for HTTP dates
    dt = datetime.fromtimestamp(timestamp, tz=timezone.utc)
    # Format: 'Wed, 21 Oct 2015 07:28:00 GMT'
    return dt.strftime("%a, %d %b %Y %H:%M:%S GMT")
```

D. Core Logic Skeleton Code (Data Model Classes)

File: `src/blue_origin/models/cache_entry.py`

Data Model: Cache Entry & Relationships

PYTHON

```python
"""
Core CacheEntry data model and associated logic.
"""
import time
from typing import Dict, List, Optional
from dataclasses import dataclass, field

from ..http_utils.cache_control import CacheDirectives
from ..http_utils.dates import parse_http_date

@dataclass
class CacheEntry:
    # Field definitions as per the data model table
    key: str
    url: str
    vary_headers: Dict[str, str]
    status_code: int
    headers: Dict[str, str]
    body: bytes
    fetched_at: float
    expires_at: float
    last_used_at: float = field(default_factory=time.time)
    use_count: int = 0
    surrogate_keys: List[str] = field(default_factory=list)
    etag: Optional[str] = None
    last_modified: Optional[str] = None

    def is_fresh(self, current_time: Optional[float] = None) -> bool:
        """
        Check if cache entry is still fresh (i.e., current_time < expires_at).
        """
        # TODO 1: If current_time is None, use time.time()
        # TODO 2: Compare current_time with self.expires_at
        # TODO 3: Return True if current_time < expires_at, False otherwise
        pass

    def is_stale_but_revalidatable(self, current_time: Optional[float] = None) -> bool:
        """
        Check if a stale entry can be served during background revalidation.
        This depends on the stale-while-revalidate directive.
        """
        # TODO 1: If current_time is None, use time.time()
```

```
        # TODO 2: Check if current_time >= self.expires_at (i.e., stale)
        # TODO 3: Parse Cache-Control from self.headers to get CacheDirectives
        # TODO 4: If directives.stale_while_revalidate exists, calculate the grace period
        # TODO 5: Return True if (current_time - self.expires_at) < grace_period
        # TODO 6: Otherwise return False
        pass

    @classmethod
    def from_upstream_response(
        cls,
        key: str,
        url: str,
        vary_headers: Dict[str, str],
        status_code: int,
        headers: Dict[str, str],
        body: bytes,
        surrogate_keys: List[str],
    ) -> "CacheEntry":
        """
        Factory method to create a CacheEntry from an upstream HTTP response.
        Calculates fetched_at and expires_at based on current time and Cache-Control headers.
        """
        # TODO 1: Record current time as fetched_at
        # TODO 2: Parse Cache-Control header to get CacheDirectives
        # TODO 3: Determine effective TTL (in seconds):
        #   - Prefer s_maxage if present (CDN-specific)
        #   - Then max_age
        #   - Then fall back to parsing Expires header and subtracting fetched_at
        #   - Default to a short TTL (e.g., 60 seconds) if none present
        # TODO 4: Calculate expires_at = fetched_at + effective_ttl
        # TODO 5: Extract etag and last_modified from headers
        # TODO 6: Create and return a new CacheEntry instance with all fields
        pass
```

File: `src/blue_origin/models/invalidation.py`

PYTHON

```python
"""
Data structures for cache invalidation: SurrogateKeyIndex and BanRule.
"""
import time
from typing import Dict, List, Set


class SurrogateKeyIndex:
    """
    Inverted index mapping surrogate keys (tags) to cache entry keys.
    """
    def __init__(self) -> None:
        self.key_to_tags: Dict[str, List[str]] = {}
        self.tag_to_keys: Dict[str, Set[str]] = {}

    def add_entry(self, cache_key: str, tags: List[str]) -> None:
        """
        Register a cache entry and its tags with the index.
        """
        # TODO 1: Store mapping from cache_key -> tags in self.key_to_tags
        # TODO 2: For each tag in tags, add cache_key to the set in self.tag_to_keys[tag]
        #         (create the set if it doesn't exist)
        pass

    def remove_entry(self, cache_key: str) -> None:
        """
        Remove a cache entry from the index, cleaning up tag associations.
        """
        # TODO 1: Look up the tags for this cache_key from self.key_to_tags
        # TODO 2: For each tag in those tags, remove cache_key from self.tag_to_keys[tag]
        # TODO 3: If the set for a tag becomes empty, delete the tag key from the dictionary
        # TODO 4: Remove the cache_key from self.key_to_tags
        pass

    def get_keys_for_tag(self, tag: str) -> Set[str]:
        """Return all cache keys tagged with the given tag."""
        return self.tag_to_keys.get(tag, set())

    def purge_by_tag(self, tag: str) -> Set[str]:
        """
        Return all cache keys for a given tag and clear the index for that tag.
        Called by the invalidation manager.
        """
        # TODO 1: Get the set of keys for this tag (or empty set)
```

```
        # TODO 2: For each key in that set, remove it from self.key_to_tags
        # TODO 3: Delete the tag entry from self.tag_to_keys
        # TODO 4: Return the set of keys (so they can be purged from primary storage)
        pass

@dataclass
class BanRule:
    """Represents a pattern-based invalidation rule."""
    pattern: str  # e.g., "/api/v1/products/*"
    created_at: float = time.time()
    is_soft: bool = False  # True for soft purge (mark stale), False for hard purge

    def matches(self, url: str) -> bool:
        """
        Check if the given URL matches this ban rule.
        For simplicity, implement prefix matching first.
        """
        # TODO 1: For prefix matching, check if url.startswith(self.pattern)
        # TODO 2: Return True if it matches, False otherwise
        # Advanced: Implement simple glob matching (e.g., using fnmatch)
        pass
```

E. Language-Specific Hints (Python)

- Use `@dataclass` from the `dataclasses` module for clean, boilerplate-free data models. It automatically generates `__init__`, `__repr__`, and `__eq__` methods.
- For the `SurrogateKeyIndex`, use Python's built-in `set` for `tag_to_keys` values to ensure uniqueness and fast membership tests.
- Use `time.time()` for timestamps. It returns a float representing seconds since the Unix epoch in UTC (system clock dependent, but sufficient for relative TTLs).
- When parsing dates, the `email.utils.parsedate_to_datetime` function handles the various HTTP date formats correctly and is in the standard library.
- For thread-safe access to shared structures like the metrics counters or invalidation indexes, use `threading.Lock` or `threading.RLock`. Consider using `collections.defaultdict` for metrics breakdowns to avoid key checks.

F. Milestone Checkpoint (Data Model)

After implementing the data structures above, you should be able to run a simple test to verify their basic functionality.

Checkpoint Commands:

```bash
# Run unit tests for the models module
python -m pytest src/blue_origin/models/ -xvs

# Expected: Tests should pass, demonstrating:
# 1. CacheEntry.is_fresh() returns True before TTL, False after.
# 2. SurrogateKeyIndex correctly adds and removes entries, maintaining consistency.
# 3. BanRule.matches() correctly identifies URLs with a given prefix.
```

Manual Verification Script: Create a simple script `test_models.py`:

```python
from blue_origin.models.cache_entry import CacheEntry, CacheDirectives

from blue_origin.models.invalidation import SurrogateKeyIndex, BanRule

import time


# Test CacheEntry freshness

entry = CacheEntry(

    key="test_key",

    url="/test",

    vary_headers={},

    status_code=200,

    headers={},

    body=b"",

    fetched_at=time.time() - 10,  # 10 seconds ago

    expires_at=time.time() + 50,   # expires in 50 seconds

)

assert entry.is_fresh() == True

print("√ CacheEntry freshness check passed")


# Test SurrogateKeyIndex

index = SurrogateKeyIndex()

index.add_entry("key1", ["tag1", "tag2"])

index.add_entry("key2", ["tag2"])

assert index.get_keys_for_tag("tag2") == {"key1", "key2"}

index.remove_entry("key1")

assert index.get_keys_for_tag("tag2") == {"key2"}

print("√ SurrogateKeyIndex consistency passed")


# Test BanRule prefix matching

ban = BanRule(pattern="/api/v1/")

assert ban.matches("/api/v1/products/123") == True

assert ban.matches("/static/image.jpg") == False

print("√ BanRule prefix matching passed")
```

Run it: `python test_models.py`. All assertions should pass silently.

Component: Edge Cache (Milestone 1 & 5)

> **Milestone(s):** Milestone 1 (Edge Cache Implementation), Milestone 5 (CDN Analytics & Performance Optimization)

The edge cache is the beating heart of a CDN. It's the component that directly interacts with end users, intercepting their requests and deciding whether to serve cached content instantly or fetch it from upstream. This section details the design and implementation of this critical component, combining the foundational caching logic from Milestone 1 with the performance optimizations from Milestone 5.

Mental Model: The Smart Mailroom

Imagine a large corporate office building with a central mailroom. All incoming packages (HTTP requests) arrive at this mailroom, and the mailroom clerk (edge cache) has a simple goal: deliver packages to employees (return responses to users) as quickly as possible.

1. **Package Lookup (Cache Key)**: Each package is labeled with a recipient name and department (URL and Vary headers). The clerk first checks organized shelves (the cache) for an identical package already waiting.
2. **Freshness Check (TTL)**: Packages have expiration dates stamped on them (Cache-Control headers). The clerk won't deliver stale milk (expired content).
3. **Smart Validation (Conditional Requests)**: If a package looks old, the clerk calls the sender (origin) and asks, "Has this changed since last Tuesday?" (If-Modified-Since). If not, they reuse the old package (304 Not Modified).
4. **Efficient Storage (Eviction)**: Shelves have limited space. When full, the clerk removes the least recently used packages (LRU) or packages from rarely visited departments (LFU) to make room for new arrivals.
5. **Value-Added Services (Analytics & Compression)**: The clerk keeps a log of how many packages they handled (analytics) and can repack bulky items into smaller boxes (compression) before delivery to save bandwidth.

This mental model captures the core responsibilities: efficient storage/retrieval, freshness management, validation, and optimization—all performed at the "edge" where users first connect.

Interface & HTTP Flow

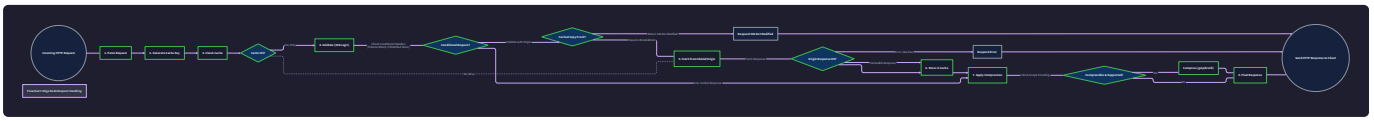
The `EdgeRequestHandler` is the primary interface between the outside world and the cache. It receives HTTP requests, processes them through the caching logic, and returns HTTP responses.

Interface Definition:

Method	Parameters	Returns	Description
`EdgeRequestHandler.handle_request`	`request_headers: Dict[str, str]`, `request_body: bytes`	`Tuple[int, Dict[str, str], bytes]` (status, headers, body)	The main request processing entry point. It follows the caching algorithm defined in RFC 9111.
`EdgeRequestHandler._fetch_from_upstream`	`headers: Dict[str, str]`, `body: bytes`	`Tuple[int, Dict[str, str], bytes]`	Forwards a request to the upstream (shield or origin) when a cache miss occurs.
`EdgeRequestHandler._revalidate_in_background`	`cache_key: str`, `cache_entry: CacheEntry`, `request_headers: Dict[str, str]`	`None`	Asynchronously revalidates a stale-but-revalidatable cache entry while the current request is served stale content.

Request Handling Algorithm:

The flowchart



illustrates the complete decision flow. Here is the step-by-step algorithm executed by `handle_request`:

1. **Request Parsing & Normalization**: Extract the request URL, method, and relevant headers (`Cache-Control`, `If-None-Match`, `If-Modified-Since`, `Accept-Encoding`). Normalize the URL (e.g., remove default ports, ensure consistent casing for the scheme/host).
2. **Cacheability Pre-check**: If the request method is not `GET` or `HEAD`, or if the request contains a `Cache-Control: no-store` directive, bypass the cache entirely and forward directly to `_fetch_from_upstream`.
3. **Cache Key Generation**: Construct a unique `cache_key` by combining the normalized request URL with the values of all headers listed in the `Vary` header of the *cached response* (or if no cached entry exists yet, the current request's `Vary`-relevant headers). This ensures different representations (e.g., gzip vs. plain text) are stored separately.
4. **Cache Lookup**: Query the `CacheStorage` with the generated `cache_key`. The result is either a `CacheEntry` (HIT) or `None` (MISS).
5. **On Cache HIT**: a. **Freshness Determination**: Calculate the current age of the entry (current time - `fetched_at`). Compare against TTL derived from `CacheDirectives` (see TTL Management below). b. **If Fresh**: Serve the cached response immediately. Update `last_used_at` and `use_count` for eviction policies. Record a cache hit in `EdgeMetrics`. c. **If Stale but Revalidatable**: Check `CacheDirectives.stale_while_revalidate`. If within this grace period, serve the stale response immediately (`_revalidate_in_background`), then asynchronously revalidate the entry with the origin and update the cache. Record a hit. d. **If Stale and Must Revalidate**: The entry cannot be served without validation. Proceed to step 6 (Conditional Request) using the cached entry's `etag` and `last_modified`.

6. **On Cache MISS or Need Validation**: a. **Prepare Upstream Request**: If a cached entry exists (for validation), add `If-None-Match: {etag}` and/or `If-Modified-Since: {last_modified}` headers to the upstream request. b. **Fetch from Upstream**: Call `_fetch_from_upstream` with the (potentially conditional) request. c. **Handle Response**:
    - **304 Not Modified (Validation Success)**: The cached entry is fresh. Update its `fetched_at` and `expires_at` timestamps based on new headers in the 304 response, then serve the cached body. Record a hit.
    - **200 OK (Miss or Changed)**: This is a new or updated resource. Determine if the response is cacheable using `is_response_cacheable`. If cacheable, create a new `CacheEntry` via `CacheEntry.from_upstream_response` and store it. Record a miss.
    - **Any other status**: Pass through to client. Cache error responses (like 404, 500) only if they have explicit caching directives (negative caching).
7. **Post-Processing**: Before returning the final response: a. **Compression (Milestone 5)**: If the client supports compression (`Accept-Encoding`) and the content type is compressible (e.g., text/html), and the response isn't already compressed, apply gzip or Brotli compression. Update the `Content-Encoding` and `Content-Length` headers. b. **Add Diagnostic Headers**: Append headers like `X-Cache: HIT` or `X-Cache: MISS` for debugging. c. **Update Analytics**: Increment counters in `EdgeMetrics` (hits, misses, bandwidth served).
8. **Return Response**: Send the final status code, headers, and body to the client.

Cache Key & TTL Management

Two of the most critical and subtle aspects of HTTP caching are correctly constructing cache keys and interpreting the hierarchy of TTL directives.

Cache Key Construction:

The cache key must uniquely identify a *representation* of a resource, not just the resource itself. The primary inputs are:

1. **Request URL**: The full URL including scheme, host, port, path, and query string.
2. **Vary Header Dimensions**: The values of request headers listed in the `Vary` header of the *cached response*.

> **Key Insight:** You cannot know which request headers are relevant for variation until *after* you have received a response from the origin. Therefore, the cache key for lookup (step 4 above) must be constructed using the *Vary header from the previously cached entry*. If no entry exists, you must store the request's headers that *might* be varied on, and then after fetching the response, use its `Vary` header to construct the final storage key.

Header	Role in Cache Key	Example
`Vary: Accept-Encoding`	Cache key must include the `Accept-Encoding` value from the request.	Requests for `image.jpg` with `Accept-Encoding: gzip` and `Accept-Encoding: identity` get separate cache entries.
`Vary: User-Agent`	Cache key includes the `User-Agent` string.	Different cache entries for mobile vs. desktop versions of a page.
`Vary: *`	Special case: The response is **never** stored.	Serves as a directive to never cache this response.

TTL Management & Freshness Calculation:

A `CacheEntry`'s freshness is determined by its `expires_at` timestamp, which is calculated when the entry is created or revalidated. The hierarchy of TTL directives, from most to least authoritative for a shared cache (like our CDN), is:

1. `Cache-Control: s-maxage=<seconds>`: Explicitly for shared caches. Overrides `max-age` and `Expires`.
2. `Cache-Control: max-age=<seconds>`: For both private and shared caches.
3. `Expires: <date>`: An absolute expiration date.
4. Heuristic Freshness: If no explicit directive exists, RFC 9111 allows using a heuristic (e.g., 10% of the time since `Last-Modified`). For simplicity, we treat such responses as non-cacheable unless explicitly configured otherwise.

The `CacheDirectives.from_header` method parses the `Cache-Control` header string into a structured object. The algorithm for calculating `expires_at` is:

```
fetched_at = current_time
if directives.s_maxage is not None:
    expires_at = fetched_at + directives.s_maxage
elif directives.max_age is not None:
    expires_at = fetched_at + directives.max_age
elif Expires header exists and is valid:
    expires_at = parse_http_date(Expires_header)
else:
    // Not cacheable by explicit directives
```

The `CacheEntry.is_fresh(current_time)` method simply returns `current_time < expires_at`.

Stale-While-Revalidate & Stale-If-Error: These directives allow graceful behavior beyond simple freshness:

- `stale-while-revalidate`: After `expires_at`, for the next N seconds, the cache may serve stale data while asynchronously revalidating it.
- `stale-if-error`: If the origin is failing, stale content may be served for N seconds past expiration.

These are checked by `CacheEntry.is_stale_but_revalidatable`.

ADR: Cache Storage & Eviction Policy

> **Decision: In-Memory LRU Cache with Size Limit**
>
> - **Context**: We need a fast, concurrent cache storage layer for the edge node. The cache must hold thousands to millions of responses, evict entries when capacity is reached, and support fast lookups by complex string keys.
> - **Options Considered**:
>   1. **In-memory LRU/LFU cache**: Store everything in the node's RAM using a structure like a hash map + doubly linked list for LRU ordering.
>   2. **Disk-backed cache (e.g., SQLite, RocksDB)**: Use local SSD storage for larger capacity, trading some latency for persistence and size.
>   3. **External cache service (e.g., Redis, Memcached)**: Delegate storage to a dedicated in-memory data store, allowing multiple edge processes to share a cache.
> - **Decision**: Implement an in-memory LRU (Least Recently Used) cache as the default `CacheStorage`.
> - **Rationale**:
>   - **Performance**: RAM access is orders of magnitude faster than disk or network. Edge caching's primary goal is low latency.
>   - **Simplicity**: An in-memory LRU is straightforward to implement and reason about. Python's `functools.lru_cache` or a custom `OrderedDict`-based implementation works well.
>   - **Predictable Memory Footprint**: A size limit (e.g., 1GB) ensures the cache doesn't cause the node to run out of memory.
>   - **Educational Value**: Implementing LRU eviction teaches fundamental data structure concepts.
> - **Consequences**:
>   - **Volatility**: Cache contents are lost on process restart. This is acceptable for a CDN edge (cache warming can repopulate).
>   - **Size Limitation**: The cache cannot exceed available RAM. Large media files may limit the number of objects cached.
>   - **Single-Node Only**: Cache is not shared between edge processes on different machines. This is fine for our architecture where each edge node is independent.

Option	Pros	Cons	Chosen?
In-memory LRU/LFU	Extremely fast, simple to implement, predictable memory use	Volatile, limited to RAM size, not shared	**Yes** (LRU for simplicity)
Disk-backed cache	Larger capacity, persistent across restarts	Higher latency (I/O), more complex eviction (disk space), wear on SSDs	No
External cache service	Shared cache across processes, potentially larger, persistence options	Network latency added to every lookup, extra operational complexity	No

Eviction Policy Details: When the cache reaches its configured capacity (in bytes or item count), it must evict entries. LRU eviction removes the entry with the oldest `last_used_at` timestamp. This assumes recent requests are predictive of future requests, which is a common access pattern. The `CacheEntry.last_used_at` field must be updated on every cache hit.

An alternative is LFU (Least Frequently Used), which evicts the entry with the smallest `use_count`. This can better retain "popular but not recently accessed" items but requires more overhead to track and decay frequencies. For our educational project, LRU provides a good balance.

Common Pitfalls in Edge Caching

⚠️ **Pitfall: Ignoring the Vary Header Description**: Constructing cache keys using only the URL, ignoring the `Vary` header. This causes cache corruption where a user requesting a gzipped version might get a plain-text version cached for a different user. **Why it's wrong**: Violates HTTP semantics, serves incorrect content, and can break websites. **Fix**: Always include the normalized values of *all* headers listed in the cached response's `Vary` header in the cache key. Handle `Vary: *` by not caching at all.

⚠️ **Pitfall: Confusing s-maxage and max-age Description**: Using `max-age` when `s-maxage` is present for shared cache decisions. **Why it's wrong**: `s-maxage` is specifically for shared caches (like CDNs). Using `max-age` instead may cause the CDN to cache content for shorter durations than intended for public users, increasing origin load. **Fix**: In your TTL calculation hierarchy, prioritize `s-maxage` over `max-age` for shared caches.

⚠️ **Pitfall: Cache Key Collisions with Normalization Description**: Different URLs that point to the same resource (e.g., `http://example.com` vs `http://example.com:80`, or differing query parameter order) create duplicate cache entries. **Why it's wrong**: Wastes cache space, reduces hit rates. **Fix**: Normalize URLs before generating cache keys: convert scheme/host to lowercase, remove default ports, sort query parameters alphabetically (if the origin treats them as order-independent).

⚠️ **Pitfall: Forgetting to Cache 304 Responses Description**: When a conditional request returns `304 Not Modified`, not updating the cached entry's freshness metadata. **Why it's wrong**: The cached entry's `expires_at` remains based on the original fetch time, causing it to become stale prematurely. **Fix**: On a 304 response, extract any new `Cache-Control`, `Expires`, or `ETag` headers from the response and update the existing `CacheEntry`'s metadata accordingly, effectively "refreshing" the TTL.

⚠️ **Pitfall: Not Respecting no-store on Requests Description**: Caching responses for requests that contained `Cache-Control: no-store`. **Why it's wrong**: The user explicitly requested that sensitive information not be stored. Violating this is a privacy/security issue. **Fix**: Check request directives at the start of the algorithm and bypass the cache entirely for `no-store` requests.

Implementation Guidance for Edge Cache

A. Technology Recommendations Table:

Component	Simple Option (for Learning)	Advanced Option (for Production)
HTTP Server	Python's `http.server` with threading	`asyncio` + `aiohttp` for high concurrency
Cache Storage	Python `dict` + `collections.OrderedDict` for LRU	`lru-dict` library or custom sharded hash map
Compression	Python's `gzip` library	`brotli` library for Brotli compression
Header Parsing	Manual string splitting using `parse_cache_control_header`	Use `cachetools` library for RFC-compliant parsing

B. Recommended File/Module Structure:

```
blue_origin/
├── README.md
├── pyproject.toml
├── src/
│   └── blue_origin/
│       ├── __init__.py
│       ├── config.py                 # EdgeConfig, config loading
│       ├── constants.py              # HTTP constants, defaults
│       ├── models/                   # Data models
│       │   ├── __init__.py
│       │   ├── cache.py              # CacheEntry, CacheDirectives
│       │   ├── metrics.py            # EdgeMetrics, TimeWindowedMetrics
│       │   └── invalidation.py       # SurrogateKeyIndex, BanRule
│       ├── cache/                    # Core caching logic
│       │   ├── __init__.py
│       │   ├── storage.py            # CacheStorage interface & LRU impl
│       │   ├── key_generator.py      # Cache key generation
│       │   ├── ttl_manager.py        # TTL calculation
│       │   └── validator.py          # Conditional request logic
│       ├── handlers/                 # HTTP handlers
│       │   ├── __init__.py
│       │   ├── edge_handler.py       # EdgeRequestHandler
│       │   └── purge_handler.py      # Admin API for invalidation
│       ├── middleware/               # Response processing
│       │   ├── __init__.py
│       │   ├── compression.py        # gzip/Brotli compression
│       │   └── analytics.py          # Metrics collection
│       ├── utils/                    # Utilities
│       │   ├── __init__.py
│       │   ├── http_utils.py         # parse_http_date, etc.
│       │   └── headers.py            # Header manipulation
│       └── server.py                 # Main server entry point
└── tests/
```

C. Infrastructure Starter Code (COMPLETE HTTP Utilities):

PYTHON

```python
# src/blue_origin/utils/http_utils.py

"""

HTTP utility functions for parsing and formatting dates, headers, etc.

"""

import time

from email.utils import parsedate_to_datetime

from typing import Dict, List, Optional

from urllib.parse import urlparse, urlunparse


def parse_http_date(date_str: str) -> Optional[float]:

    """Parse an RFC 1123/822 date string into a Unix timestamp."""

    if not date_str:

        return None

    try:

        dt = parsedate_to_datetime(date_str)

        return dt.timestamp()

    except (ValueError, TypeError):

        return None


def format_http_date(timestamp: float) -> str:

    """Format a Unix timestamp to RFC 1123 string."""

    return time.strftime('%a, %d %b %Y %H:%M:%S GMT', time.gmtime(timestamp))


def parse_cache_control_header(header_value: Optional[str]) -> Dict[str, Optional[str]]:

    """

    Parse Cache-Control header into dictionary of directives.

    Example: "max-age=60, public" -> {"max-age": "60", "public": None}

    """

    directives = {}

    if not header_value:

        return directives

    for part in header_value.split(','):

        part = part.strip()

        if '=' in part:

            key, val = part.split('=', 1)

            directives[key.strip()] = val.strip()

        else:

            directives[part] = None

    return directives


def get_header_values(headers: Dict[str, str], header_name: str) -> List[str]:

    """
```

```
    Get all values for a header (handling comma-separated and multiple headers).
    HTTP allows multiple headers with same name and comma-separated values.
    """
    # Case-insensitive search
    header_name_lower = header_name.lower()
    matched_values = []
    for name, value in headers.items():
        if name.lower() == header_name_lower:
            # Split by comma, but be careful of quoted strings (simple split works for most caching headers)
            matched_values.extend([v.strip() for v in value.split(',')])
    return matched_values

def normalize_url(url: str) -> str:
    """
    Normalize URL for cache key: lowercase scheme/host, remove default port.
    Does NOT sort query parameters (application-specific).
    """
    parsed = urlparse(url)
    # Lowercase scheme and host
    scheme = parsed.scheme.lower()
    netloc = parsed.hostname.lower() if parsed.hostname else ''
    # Add port if non-default
    if parsed.port:
        if not (scheme == 'http' and parsed.port == 80) and not (scheme == 'https' and parsed.port == 443):
            netloc = f'{netloc}:{parsed.port}'
    # Reconstruct without fragment (fragment is never sent to server)
    return urlunparse((scheme, netloc, parsed.path, parsed.params, parsed.query, ''))
```

D. Core Logic Skeleton Code:

PYTHON

```python
# src/blue_origin/cache/key_generator.py

"""

Cache key generation combining URL and Vary header values.

"""

from typing import Dict, List

from ..utils.http_utils import normalize_url


def generate_cache_key(url: str, vary_headers: Dict[str, str], vary_header_list: List[str]) -> str:

    """

    Generate a cache key from URL and the values of headers specified in Vary.



    Args:

        url: The request URL (will be normalized).

        vary_headers: Dictionary containing ALL request headers (lowercased keys).

        vary_header_list: List of header names from the cached response's Vary header.



    Returns:

        A string cache key.



    TODO 1: Normalize the URL using normalize_url()

    TODO 2: For each header name in vary_header_list:

            a. Get its value from vary_headers (case-insensitive)

            b. If header is not present, use empty string

            c. Normalize header value (strip, lowercase for some headers?)

    TODO 3: Sort the vary_header_list to ensure consistent key generation

            (Vary: Accept-Encoding,User-Agent should match Vary: User-Agent,Accept-Encoding)

    TODO 4: Combine normalized URL with sorted header name-value pairs

            Example format: "GET:{url}:{header1}={value1}:{header2}={value2}"

    TODO 5: Return the combined string (consider hashing for shorter keys if needed)

    """

    pass


# src/blue_origin/models/cache.py

"""

Cache data structures.

"""

from dataclasses import dataclass, field

from typing import Dict, List, Optional

import time


@dataclass

class CacheDirectives:
```

```python
    """Parsed Cache-Control directives."""
    s_maxage: Optional[int] = None
    max_age: Optional[int] = None
    no_cache: bool = False
    no_store: bool = False
    must_revalidate: bool = False
    proxy_revalidate: bool = False
    public: bool = False
    private: bool = False
    stale_while_revalidate: Optional[int] = None
    stale_if_error: Optional[int] = None

    @classmethod
    def from_header(cls, cache_control_header: Optional[str]) -> 'CacheDirectives':
        """
        Parse a Cache-Control header string into a structured object.

        TODO 1: Use parse_cache_control_header() to get dictionary of directives
        TODO 2: Initialize a CacheDirectives instance with default values
        TODO 3: For each key-value in the dictionary:
                a. Handle boolean directives (no-cache, no-store, etc.): set to True if present
                b. Handle numeric directives (s-maxage, max-age, etc.): convert string to int
        TODO 4: Special case: 'public' and 'private' are mutually exclusive
                (private takes precedence if both are present per RFC 9111)
        TODO 5: Return the populated CacheDirectives object
        """
        pass

@dataclass
class CacheEntry:
    """A cached HTTP response."""
    key: str
    url: str
    vary_headers: Dict[str, str]
    status_code: int
    headers: Dict[str, str]
    body: bytes
    fetched_at: float
    expires_at: float
    last_used_at: float = field(default_factory=time.time)
    use_count: int = 1
```

```
    surrogate_keys: List[str] = field(default_factory=list)
    etag: Optional[str] = None
    last_modified: Optional[str] = None

    def is_fresh(self, current_time: float) -> bool:
        """Check if cache entry is still fresh."""
        # TODO: Return True if current_time < expires_at, False otherwise
        pass

    def is_stale_but_revalidatable(self, current_time: float, directives: CacheDirectives) -> bool:
        """
        Check if stale entry can be served during revalidation.

        TODO 1: If entry is fresh, return False (no need for revalidation)
        TODO 2: Calculate staleness: current_time - expires_at
        TODO 3: Check if directives.stale_while_revalidate exists and staleness <= that value
        TODO 4: Return True if conditions met, False otherwise
        """
        pass

    @classmethod
    def from_upstream_response(cls, key: str, url: str, vary_headers: Dict[str, str],
                               status_code: int, headers: Dict[str, str], body: bytes,
                               surrogate_keys: List[str]) -> 'CacheEntry':
        """
        Factory method to create a CacheEntry from an upstream HTTP response.

        TODO 1: Parse Cache-Control header from response headers
        TODO 2: Calculate TTL and set expires_at (use current time for fetched_at)
        TODO 3: Extract ETag and Last-Modified headers if present
        TODO 4: Create and return a CacheEntry instance with all fields populated
        """
        pass

# src/blue_origin/handlers/edge_handler.py
"""
Main edge request handler.
"""
import time
from typing import Dict, Tuple
from ..cache.storage import CacheStorage
```

```python
from ..models.cache import CacheEntry, CacheDirectives
from ..utils.http_utils import parse_http_date, get_header_values

class EdgeRequestHandler:
    def __init__(self, cache: CacheStorage, upstream: str):
        self.cache = cache
        self.upstream = upstream

    def handle_request(self, request_headers: Dict[str, str],
                       request_body: bytes) -> Tuple[int, Dict[str, str], bytes]:
        """
        Main request handling algorithm for the edge node.

        TODO 1: Extract method, URL, and relevant headers from request_headers
        TODO 2: Check if request is cacheable (method GET/HEAD, no no-store directive)
        TODO 3: Generate preliminary cache key (use request's Vary-relevant headers)
        TODO 4: Look up entry in cache
        TODO 5: If cache HIT:
                a. Check freshness and revalidation rules
                b. If fresh or stale-while-revalidate: serve from cache
                c. If needs validation: add If-None-Match/If-Modified-Since
        TODO 6: If cache MISS or validation needed:
                a. Forward request to upstream with conditional headers
                b. Handle response:
                   - 304: Update cached entry metadata, serve cached body
                   - 200: Create new cache entry if cacheable
                   - Other: Pass through
        TODO 7: Apply compression middleware if applicable
        TODO 8: Add diagnostic headers (X-Cache: HIT/MISS)
        TODO 9: Update metrics
        TODO 10: Return final response
        """
        pass

    def _fetch_from_upstream(self, headers: Dict[str, str],
                             body: bytes) -> Tuple[int, Dict[str, str], bytes]:
        """
        Forward request to upstream (shield or origin).

        TODO 1: Add CDN-specific headers (X-Forwarded-For, etc.)
        TODO 2: Make HTTP request to self.upstream
```

```
        TODO 3: Return status, headers, body from upstream response
        """
        pass

    def _revalidate_in_background(self, cache_key: str, cache_entry: CacheEntry,
                                  request_headers: Dict[str, str]) -> None:
        """
        Asynchronously revalidate a stale cache entry.

        TODO 1: Create a background thread/task
        TODO 2: Make conditional request to upstream with cache_entry.etag/last_modified
        TODO 3: If 304: Update cache_entry.fetched_at and expires_at
        TODO 4: If 200: Replace cache entry with new response
        TODO 5: If error: Log and keep stale entry (might be served via stale-if-error later)
        """
        pass
```

E. Language-Specific Hints (Python):

- Use `threading.Lock` or `asyncio.Lock` to protect concurrent access to the cache storage.
- For LRU cache implementation, `collections.OrderedDict` maintains insertion order. On every cache hit, move the key to the end using `move_to_end(key, last=True)`. When evicting, pop the first item.
- Use `email.utils.parsedate_to_datetime` for robust HTTP date parsing (handles multiple formats).
- For background tasks, consider `concurrent.futures.ThreadPoolExecutor` or `asyncio.create_task` depending on your concurrency model.
- When applying compression, check `Accept-Encoding` header for `gzip` or `br` (Brotli). Use Python's `gzip.compress()` and the `brotli` library if installed.

F. Milestone Checkpoint (Edge Cache): After implementing the edge cache, run the following test:

1. Start your edge server: `python -m blue_origin.server --config config.yaml`
2. Use `curl` to make requests:

```
curl -v http://localhost:8080/image.jpg
```

First request should show `X-Cache: MISS`. Second identical request should show `X-Cache: HIT`. 3. Test conditional requests: Add an `If-None-Match` header with a known ETag from a previous response. You should receive `304 Not Modified`. 4. Verify TTL: Set a `Cache-Control: max-age=2` header on your origin server. Request the resource, wait 3 seconds, request again. The second request should trigger a revalidation or miss. 5. Check compression: `curl -H "Accept-Encoding: gzip" -v http://localhost:8080/some-text.html` should show `Content-Encoding: gzip` in the response.

Expected behavior: The edge server responds correctly to cache directives, respects freshness, and shows improved performance on repeated requests.

Component: Cache Invalidation (Milestone 2)

> **Milestone(s):** Milestone 2 (Cache Invalidation)

The edge cache's ability to serve content quickly becomes useless when that content changes at the origin. **Cache invalidation**—the active removal or marking of outdated cached content—is the mechanism that maintains consistency between the cached copies and the origin's truth. This component transforms the CDN from a static snapshotter into a dynamic, synchronized delivery system. Implementing invalidation is a classic distributed systems challenge: we must ensure that updates propagate to potentially hundreds of edge nodes, each holding independent copies, without overwhelming the system or introducing prolonged inconsistency windows.

Mental Model: The Building Eviction Notice

Imagine the CDN's cache as a large apartment building (the cache storage). Each apartment (cache entry) is rented by a tenant (a cached response). When the building owner (the origin) decides to change the terms or reclaim an apartment, they issue different types of notices:

1. **Hard Purge (Immediate Eviction):** A sheriff arrives, immediately removes the tenant, and changes the locks. The apartment is now empty. This is analogous to a **hard purge**—the cache entry is deleted instantly. The next request for that resource will experience a cache miss and fetch a fresh copy from the origin.
2. **Soft Purge (Notice to Renew Lease):** The owner posts a notice saying, "Your current lease expires now, but you can stay while we process a renewal." The tenant remains in the apartment temporarily. This is a **soft purge** (or stale-while-revalidate). The cache marks the entry as stale but continues serving it to clients while asynchronously fetching a fresh version in the background. This provides graceful degradation and avoids a thundering herd of requests to the origin.
3. **Ban (Evicting Everyone on a Floor):** The building manager declares, "Everyone in apartments with numbers ending in '01' must leave." This is a **ban**—a pattern-based invalidation. Instead of specifying each apartment, you define a rule (e.g., a URL pattern like `/blog/*`). The eviction may happen immediately or, in **eventual invalidation**, only when someone next tries to enter that apartment (cache access). Bans are powerful for bulk operations but require careful management to avoid unbounded rule growth.
4. **Surrogate Key Purge (Tag-Based Eviction):** Apartments are tagged with colored stickers representing features (e.g., "renovated," "ocean view"). The owner announces, "All apartments with a 'renovated' sticker must be updated." This is **tag-based invalidation** using **surrogate keys**. Cache entries are tagged (e.g., with `product-12345`) upon storage. Invalidating by that tag removes all related entries in a single operation, which is far more efficient than listing every individual URL.

This mental model clarifies the trade-offs: immediacy vs. origin load, precision vs. management overhead. A production CDN uses all these "notice types" in different scenarios.

Purge, Ban, and Tag-Based Invalidation

The invalidation component exposes administrative APIs and internal logic to execute the three core invalidation strategies. Each strategy targets a different granularity and use case.

1. Purge by Exact URL

The most precise operation. It removes a single cached response identified by its exact cache key (URL + Vary header values).

Algorithm: Hard Purge by URL

1. An administrative request arrives (e.g., `PURGE /images/photo.jpg` with authentication).
2. The handler validates the requestor's permissions.
3. It constructs the exact cache key from the request URL and the relevant `Vary` header dimensions (extracted from the request or by looking up the stored entry).
4. It synchronously deletes the corresponding `CacheEntry` from the `CacheStorage`.
5. It removes the cache key from the `SurrogateKeyIndex` (if indexed).
6. It returns a `200 OK` acknowledging the purge.

Algorithm: Soft Purge by URL

1. Steps 1-3 are identical to hard purge.
2. Instead of deletion, it updates the `CacheEntry.expires_at` to a timestamp in the past (e.g., `current_time - 1`), immediately marking it stale.
3. It optionally sets a flag or updates metadata to indicate the entry is under soft purge.
4. It triggers `_revalidate_in_background` for this key, which will asynchronously fetch a new version from the origin and update the cache.
5. Subsequent client requests for this key will be served the stale content (if `stale-while-revalidate` directives allow) while the background refresh proceeds.
6. Returns a `200 OK` acknowledging the soft purge.

API Specification

Method	Path	Headers	Body	Returns	Description
`PURGE`	`/{path:.*}`	`Authorization: Bearer <token>`, `Soft-Purge: 1` (optional)	Empty	`200 OK` (success), `404 Not Found` (key not in cache), `401 Unauthorized`	Hard or soft purge of the exact URL path. The `Soft-Purge` header triggers the soft purge behavior.

2. Ban by Pattern (Eventual Invalidation)

A ban is a rule that invalidates cache entries matching a pattern (e.g., a regex or glob applied to the URL). For performance, bans are often applied **lazily** (eventually) upon cache access, not by scanning the entire cache store.

Data Structure: `BanRule` We extend the provided structure to support pattern matching and TTL.

Field Name	Type	Description
`pattern`	`str`	The URL-matching pattern (e.g., glob `"/images/*.jpg"` or regex `"^/blog/2024-"`).
`created_at`	`float`	Unix timestamp when the ban was created.
`is_soft`	`bool`	If `True`, matching entries are marked stale (soft purge). If `False`, they are deleted (hard purge).
`ttl_seconds`	`float`	Time-to-live for this ban rule. After expiry, the rule is garbage-collected. Prevents unbounded growth.
`id`	`str`	Unique identifier for the ban rule.

Algorithm: Adding a Ban Rule

1. Admin request (`POST /ban` with pattern and soft/hard flag).
2. Validate and compile the pattern for efficient matching.
3. Create a new `BanRule` with `created_at = current_time` and assign a unique ID.
4. Store the rule in a `BanRuleList` (e.g., a list or priority queue sorted by `created_at`).
5. Return the ban rule ID.

Algorithm: Applying Bans (Lazy Evaluation) This logic integrates into the main `EdgeRequestHandler.handle_request` flow, just after cache key generation but before cache lookup.

1. When a request arrives, generate the cache key (URL).
2. Before checking the cache, iterate through the active `BanRuleList`.
3. For each `BanRule` where `BanRule.matches(cache_key.url)` returns `True`: a. If `is_soft` is `False`: Delete the cache entry for this key (if it exists) and skip to origin fetch (cache miss). b. If `is_soft` is `True`: Mark the existing entry as stale (set `expires_at` in the past) and allow the request to proceed. The stale entry may be served while triggering background revalidation.
4. If no matching ban rule applies, proceed with normal cache lookup.

Garbage Collection of Ban Rules A background task periodically scans the `BanRuleList` and removes rules where `current_time > created_at + ttl_seconds`. A typical TTL for bans might be 24-48 hours.

API Specification

Method	Path	Headers	Body (JSON)	Returns	Description
`POST`	`/ban`	`Authorization: Bearer <token>`	`{"pattern": "/images/*.jpg", "soft": false, "ttl_seconds": 86400}`	`201 Created` with `{"ban_id": "..."}`	Creates a new ban rule.
`DELETE`	`/ban/{ban_id}`	`Authorization: Bearer <token>`	Empty	`200 OK` or `404 Not Found`	Removes a specific ban rule.

3. Tag-Based Invalidation with Surrogate Keys

This method groups cache entries by arbitrary tags (surrogate keys) provided by the origin via the `Surrogate-Key` response header. Invalidating by a tag removes all entries associated with that tag.

Data Structure: `SurrogateKeyIndex` We utilize the provided structure, which maintains a bidirectional mapping.

Field Name	Type	Description
`key_to_tags`	`Dict[str, List[str]]`	Maps a cache key to a list of surrogate key tags associated with that entry.
`tag_to_keys`	`Dict[str, Set[str]]`	Maps a tag to a set of cache keys that are tagged with it.

Algorithm: Tagging on Cache Storage

1. When storing a new `CacheEntry` (on cache miss), extract the `Surrogate-Key` header from the origin response. The header value is a space-separated list of tags (e.g., `Surrogate-Key: product-12345 category-shoes`).
2. Store the list of tags in `CacheEntry.surrogate_keys`.
3. Call `SurrogateKeyIndex.add_entry(cache_key, tags)` to update the index.

Algorithm: Purging by Tag

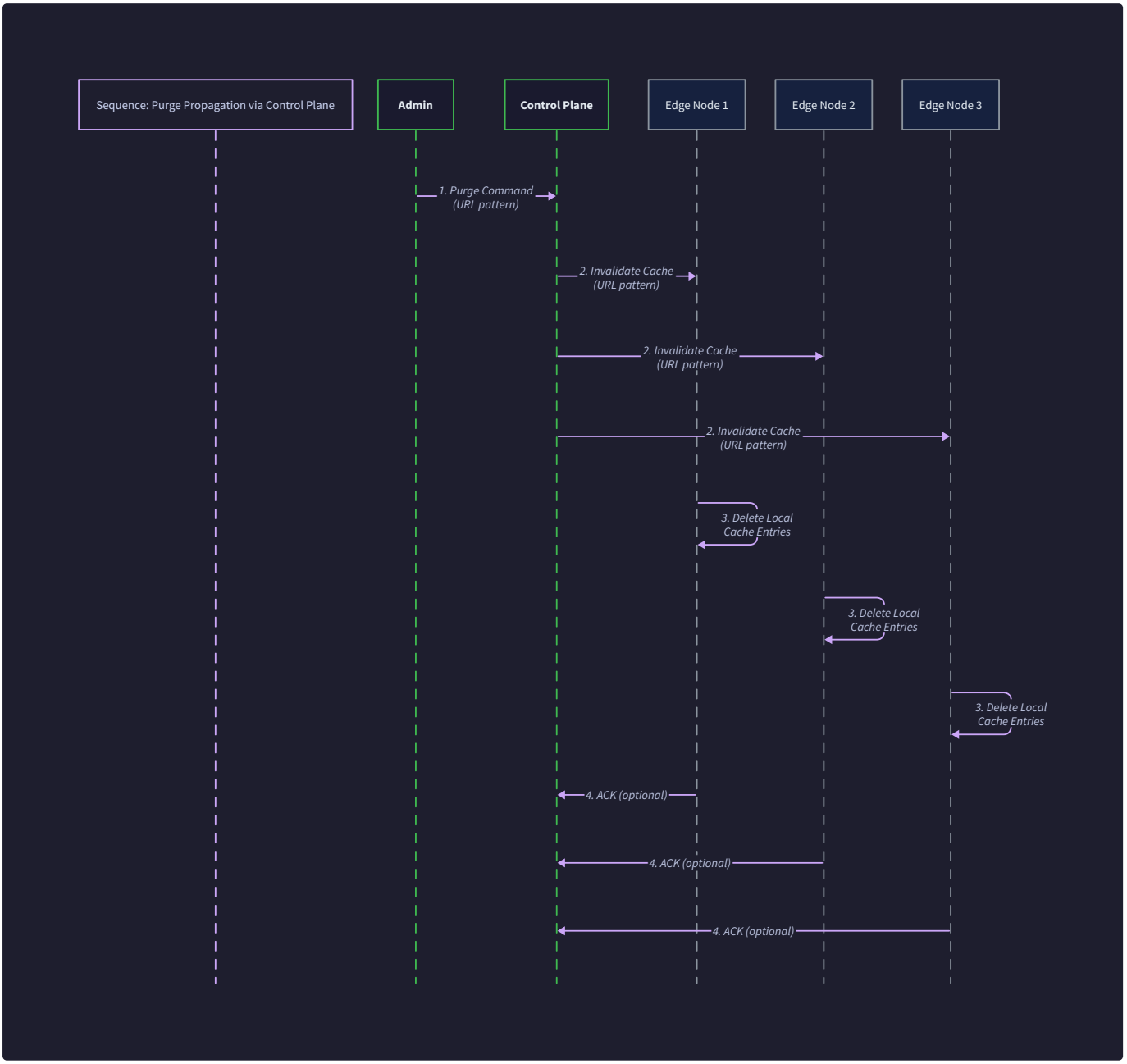
1. Admin request (`PURGE /purge/tag/{tag}`).
2. Call `SurrogateKeyIndex.purge_by_tag(tag)`. This returns the set of cache keys associated with the tag and clears the index for that tag.
3. For each cache key in the returned set: a. Perform a hard or soft purge (based on request headers) of that individual cache key.
4. Return a summary: `200 OK` with `{"purged_keys_count": 123}`.

API Specification

Method	Path	Headers	Body	Returns	Description
`PURGE`	`/purge/tag/{tag}`	`Authorization: Bearer <token>`, `Soft-Purge: 1` (optional)	Empty	`200 OK` with count of purged keys	Purges all cache entries tagged with the given surrogate key.

Invalidation Propagation Strategy

An invalidation command issued to one edge node must eventually take effect across **all** edge nodes in the CDN.



illustrates the ideal flow. The strategy defines how this propagation is achieved, balancing consistency, latency, and system complexity.

Core Requirements:

- **Eventual Consistency:** All nodes should reflect the invalidation within a bounded time window (e.g., < 60 seconds).
- **Reliability:** Invalidation messages must not be lost even if individual nodes or the network temporarily fail.
- **Scalability:** The propagation mechanism should not create a bottleneck as the number of edge nodes grows (e.g., from 10 to 10,000).
- **Ordering (Sometimes):** For certain operations, order matters (e.g., "update item A, then purge its cache"). The system should preserve this order where required.

Propagation Steps:

1. **Ingestion:** An admin issues a purge command to a designated **control plane** API (or directly to any edge node, which forwards it).
2. **Fan-out:** The control plane broadcasts the invalidation message to all registered edge nodes. This can be done via:
   - **Direct HTTP calls** to each node's administrative endpoint (simple but doesn't scale, no persistence).
   - A **publish-subscribe (pub/sub) message queue** (e.g., Redis Pub/Sub, Kafka, NATS). Edge nodes subscribe to an invalidation channel.
   - A **gossip protocol** where nodes propagate the message to their peers (decentralized, resilient, but complex).
3. **Acknowledgment & Retry:** Edge nodes acknowledge receipt. The control plane retries unacknowledged messages with exponential backoff.
4. **Local Execution:** Each edge node receives the message, validates it, and executes the local purge/ban logic described earlier.
5. **Monitoring:** The control plane logs completion and can provide a dashboard showing propagation status.

ADR: Invalidation Propagation Mechanism

> **Decision: Use a Centralized Pub/Sub System for Invalidation Propagation**
>
> - **Context**: The CDN needs to propagate cache invalidation commands (purge, ban, tag) from an administrative source to all edge nodes. The system has a moderate number of edge nodes (10s to 100s initially), and the control plane is already a central component for configuration and analytics. We need a reliable, scalable, and simple-to-implement propagation mechanism.
> - **Options Considered**:
>   1. **Direct HTTP Broadcast from Control Plane**: The control plane maintains a list of edge node admin endpoints and `POST`s the invalidation command to each sequentially or in parallel.
>   2. **Pub/Sub Message Queue (Centralized)**: Edge nodes subscribe to a channel (e.g., `invalidation`) on a centralized message broker (like Redis). The control plane publishes messages to this channel.
>   3. **Gossip Protocol (Decentralized)**: Edge nodes form a peer-to-peer mesh and propagate invalidation messages via a gossip protocol (like SWIM or epidemic broadcasting).
> - **Decision**: We will implement Option 2: A centralized Pub/Sub system using a simple broker. Initially, we can use an in-memory broker within the control plane for simplicity, with the ability to replace it with Redis or NATS for production scaling.
> - **Rationale**:
>   - **Scalability**: Pub/Sub decouples the control plane from edge nodes. The control plane publishes once, and the broker handles fan-out. This scales better than the control plane managing direct connections to hundreds of nodes (Option 1).
>   - **Reliability**: Most message brokers offer persistence and retry mechanisms. If an edge node is temporarily offline, it can reconnect and receive missed messages (depending on broker configuration). Direct HTTP (Option 1) requires the control plane to implement complex retry and state tracking.
>   - **Simplicity**: Implementing a robust gossip protocol (Option 3) is complex and introduces its own consistency and convergence timing issues. For an educational project, Pub/Sub is a more understandable and industry-standard pattern.
>   - **Natural Fit**: The control plane is already a central coordination point. Adding a Pub/Sub broker aligns with its role.
> - **Consequences**:
>   - The control plane and edge nodes now depend on the availability of the message broker. This introduces a new single point of failure. Mitigation: Use a clustered broker or accept that invalidation is a best-effort, non-critical path (serving stale content is often acceptable briefly).
>   - Edge nodes must maintain a persistent connection or polling mechanism to the broker, adding some network overhead.
>   - Message ordering is guaranteed per-channel in most brokers, which simplifies handling of related purge sequences.

Option	Pros	Cons	Chosen?
Direct HTTP Broadcast	Simple, no new components, strong consistency if all calls succeed.	Control plane becomes a bottleneck, hard to scale, must manage connection/retry logic, no persistence if edge node is down.	No
Centralized Pub/Sub	Decouples components, scalable, brokers often provide persistence and retry, industry standard.	Introduces a new component (broker) as a potential SPOF, requires managing broker connections.	**Yes**
Gossip Protocol	Highly resilient, no central broker, scales well.	Complex to implement correctly, convergence time can be unpredictable, debugging is harder.	No

Common Pitfalls in Invalidation

⚠️ Pitfall: Unbounded Growth of the Ban List

- **Description**: Continuously adding ban rules without ever removing them. Over weeks, the list grows to thousands of rules. Every incoming request must be checked against all rules, causing linear performance degradation (`O(n)` per request).
- **Why it's wrong**: It turns a performance optimization (caching) into a performance bottleneck. Eventually, request latency becomes dominated by ban rule matching.
- **How to fix**: Attach a **TTL to every ban rule**. Implement a background garbage collection task that periodically removes expired rules. For long-term bans, consider converting them to a more permanent configuration (like a cache rule that sets a very short TTL for a path pattern).

⚠️ Pitfall: Race Condition During Revalidation

- **Description**: A soft purge triggers background revalidation. Simultaneously, a client request for the same resource arrives. Both the background task and the client request might independently fetch from the origin, causing a duplicate request (wasting origin capacity) or a race in updating the cache entry.
- **Why it's wrong**: It undermines the request collapsing benefits of the origin shield and can lead to inconsistent cache state if the two fetches return slightly different data (e.g., due to timing).
- **How to fix**: Implement a **revalidation lock per cache key**. Use a dictionary mapping cache keys to `asyncio.Lock` (or similar) or a promise/future. The first entity (background job or client request) that finds a stale entry acquires the lock and performs the origin fetch. Others wait on the same lock and reuse its result.

⚠️ Pitfall: Ignoring Propagation Delay

- **Description**: After issuing a global purge, the developer assumes all edge nodes are instantly updated and tests from a single location, seeing fresh content. However, users in other regions may still see stale content for seconds or minutes due to propagation lag.
- **Why it's wrong**: It creates a false sense of consistency and can lead to user complaints ("I cleared the cache but my friend still sees the old version").
- **How to fix**: **Design for eventual consistency**. Document the expected propagation SLA (e.g., < 30s). Provide an API or admin UI to track the propagation status of an invalidation event. For scenarios requiring stronger guarantees, consider using cache versioning in URLs (e.g., `/image.jpg?v=2`) instead of purging.

⚠️ Pitfall: Surrogate Key Collision and Over-Invalidation

- **Description**: Using overly broad surrogate keys (e.g., tagging all blog posts with `"blog"`). Purging `"blog"` invalidates the entire blog cache, causing a massive origin load spike (a "cache miss storm") when those pages are next requested.
- **Why it's wrong**: Loses the granularity benefit of caching. It's equivalent to banning `/blog/*` but with less visibility.
- **How to fix**: **Use granular surrogate keys**. Combine broad and specific tags. For example, a product page could be tagged with `"product-12345"`, `"category-shoes"`, and `"page-type-detail"`. This allows invalidating a single product, all shoes, or all detail pages independently. Educate content authors on key design.

Implementation Guidance for Invalidation

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Invalidation API	HTTP `PURGE` method with bearer token authentication	gRPC admin service with richer semantics (e.g., dry-run, scheduled purge)
Propagation Broker	In-memory Pub/Sub within the Control Plane process (using `asyncio.Queue` or `concurrent.futures`)	External Redis Pub/Sub or Apache Kafka for persistence and scale
Pattern Matching for Bans	Python's `fnmatch` for glob patterns (`/images/*.jpg`)	Regular expressions (`re` module) for maximum flexibility (with careful compilation caching)

B. Recommended File/Module Structure

```
blue_origin/
├── cdn/
│   ├── __init__.py
│   ├── edge/
│   │   ├── __init__.py
│   │   ├── server.py             # Edge HTTP server, includes request handler
│   │   ├── cache.py              # CacheStorage, CacheEntry, SurrogateKeyIndex
│   │   ├── handler.py            # EdgeRequestHandler (includes ban checking)
│   │   └── invalidation.py       # Purge/Ban API handlers, local invalidation logic
│   ├── shield/
│   │   └── ...                   # Origin shield components
│   ├── control/
│   │   ├── __init__.py
│   │   ├── plane.py              # Control plane HTTP server, pub/sub broker
│   │   └── propagation.py        # Message formats, publisher/subscriber clients
│   └── common/
│       ├── __init__.py
│       ├── http_utils.py         # parse_http_date, format_http_date, etc.
│       ├── models.py             # Shared data models (EdgeConfig, etc.)
│       └── auth.py               # Simple token validation utilities
├── scripts/
│   └── purge_cli.py              # Command-line tool for issuing purge commands
└── tests/
    └── test_invalidation.py      # Unit and integration tests for invalidation
```

C. Infrastructure Starter Code

Simple In-Memory Pub/Sub Broker (`control/propagation.py`):

PYTHON

```python
import asyncio

import logging

from typing import Dict, Set, Any, Callable

from dataclasses import dataclass

import json


logger = logging.getLogger(__name__)


@dataclass

class InvalidationMessage:

    """Message format for cache invalidation commands."""

    message_id: str

    command: str  # "PURGE", "BAN", "PURGE_TAG"

    target: str   # URL, pattern, or tag

    soft: bool = False

    timestamp: float = 0.0

    # Additional parameters can be added as a dict

    parameters: Dict[str, Any] = None


    def to_json(self) -> str:

        return json.dumps({

            "message_id": self.message_id,

            "command": self.command,

            "target": self.target,

            "soft": self.soft,

            "timestamp": self.timestamp,

            "parameters": self.parameters or {}

        })


    @classmethod

    def from_json(cls, data: str) -> "InvalidationMessage":

        d = json.loads(data)

        return cls(

            message_id=d["message_id"],

            command=d["command"],

            target=d["target"],

            soft=d.get("soft", False),

            timestamp=d.get("timestamp", 0.0),

            parameters=d.get("parameters", {})

        )


class InMemoryPubSubBroker:

    """A simple in-memory publish-subscribe broker for invalidation messages."""
```

```python
    def __init__(self):
        self._subscribers: Dict[str, Set[Callable[[InvalidationMessage], None]]] = {}
        self._lock = asyncio.Lock()

    async def subscribe(self, channel: str, callback: Callable[[InvalidationMessage], None]):
        """Subscribe a callback function to a channel."""
        async with self._lock:
            if channel not in self._subscribers:
                self._subscribers[channel] = set()
            self._subscribers[channel].add(callback)
            logger.debug(f"New subscriber to channel '{channel}'. Total: {len(self._subscribers[channel])}")

    async def unsubscribe(self, channel: str, callback: Callable[[InvalidationMessage], None]):
        """Unsubscribe a callback from a channel."""
        async with self._lock:
            if channel in self._subscribers:
                self._subscribers[channel].discard(callback)
                if not self._subscribers[channel]:
                    del self._subscribers[channel]

    async def publish(self, channel: str, message: InvalidationMessage):
        """Publish a message to all subscribers of a channel."""
        logger.info(f"Publishing message {message.message_id} to channel '{channel}'")
        async with self._lock:
            subscribers = self._subscribers.get(channel, set()).copy()
        # Call subscribers without holding the lock to avoid deadlocks
        for callback in subscribers:
            try:
                # Fire and forget; consider error handling in production
                asyncio.create_task(self._safe_callback(callback, message))
            except Exception as e:
                logger.error(f"Error scheduling callback for message {message.message_id}: {e}")

    async def _safe_callback(self, callback: Callable[[InvalidationMessage], None], message: InvalidationMessage):
        """Wrapper to log errors in subscriber callbacks."""
        try:
            await callback(message)
        except Exception as e:
            logger.error(f"Subscriber callback failed for message {message.message_id}: {e}")

# Global broker instance (simple singleton)
broker = InMemoryPubSubBroker()
```

D. Core Logic Skeleton Code

Purge API Handler (`edge/invalidation.py`):

PYTHON

```python
import fnmatch

import re

import time

from typing import Set, Optional

from .cache import CacheStorage, SurrogateKeyIndex

from ..common.auth import validate_token


class InvalidationHandler:

    """Handles administrative invalidation commands (purge, ban, tag)."""


    def __init__(self, cache: CacheStorage, key_index: SurrogateKeyIndex, control_plane_publisher=None):

        self.cache = cache

        self.key_index = key_index

        self.ban_rules: List[BanRule] = []  # In-memory list, could be persisted

        self.control_plane_publisher = control_plane_publisher  # Optional publisher for propagation


    async def handle_purge_request(self, request_method: str, request_path: str,

                                   request_headers: Dict[str, str]) -> Tuple[int, Dict[str, str], bytes]:

        """

        Handles HTTP PURGE requests.

        Expected path: /{path...} for URL purge, /purge/tag/{tag} for tag purge.

        Headers: Authorization, Soft-Purge.

        """

        # TODO 1: Validate the Authorization header using validate_token.

        #   If invalid, return 401 Unauthorized.


        # TODO 2: Determine if this is a tag purge or URL purge.

        #   Check if request_path starts with '/purge/tag/'.

        #   If yes, extract the tag and proceed to tag purge logic (step 6).


        # TODO 3: For URL purge: Extract the target URL from request_path.

        #   Generate the cache key. This requires knowing the Vary headers.

        #   Hint: You may need to look up the existing CacheEntry to get its vary_headers.


        # TODO 4: Check for the 'Soft-Purge' header (e.g., '1' or 'true').

        #   Set a boolean flag `is_soft`.


        # TODO 5: Perform the local purge.

        #   If is_soft: call self._soft_purge_key(cache_key).

        #   Else: call self._hard_purge_key(cache_key).

        #   Return 200 OK with appropriate body (e.g., "Purged" or "Soft purged").


        # TODO 6: For tag purge: Extract tag from path.

        #   Call self._purge_by_tag(tag, is_soft).
```

```
        #   Return 200 OK with count of purged keys.

        # TODO 7 (Optional - Propagation): If self.control_plane_publisher is set,
        #   construct an InvalidationMessage and publish it to the 'invalidation' channel.
        #   This allows other edge nodes to learn about this purge.

        pass

    def _hard_purge_key(self, cache_key: str):
        """Immediately delete a cache entry by key."""
        # TODO 1: Delete the entry from self.cache.
        # TODO 2: Remove the entry from self.key_index using key_index.remove_entry(cache_key).

    def _soft_purge_key(self, cache_key: str):
        """Mark a cache entry as stale and trigger background revalidation."""
        # TODO 1: Retrieve the CacheEntry from self.cache.
        # TODO 2: If entry exists, set its expires_at to a past timestamp (e.g., time.time() - 1).
        # TODO 3: Optionally, set a flag in the entry metadata (e.g., entry.is_soft_purged = True).
        # TODO 4: Trigger background revalidation: asyncio.create_task(_revalidate_in_background(...)).
        #   You'll need to pass appropriate request headers (maybe default ones).

    async def _purge_by_tag(self, tag: str, is_soft: bool):
        """Purge all cache entries associated with a surrogate key tag."""
        # TODO 1: Call self.key_index.purge_by_tag(tag) to get the set of cache keys.
        # TODO 2: For each cache key in the set:
        #   if is_soft: self._soft_purge_key(key)
        #   else: self._hard_purge_key(key)
        # TODO 3: Return the count of purged keys.

    async def add_ban_rule(self, pattern: str, is_soft: bool, ttl_seconds: float) -> str:
        """Add a new ban rule and return its ID."""
        # TODO 1: Generate a unique ID for the rule (e.g., using uuid or timestamp).
        # TODO 2: Create a BanRule object with pattern, is_soft, ttl_seconds, and current time.
        # TODO 3: Append the rule to self.ban_rules.
        # TODO 4: Schedule a garbage collection task if not already running.
        # TODO 5: Return the rule ID.

    def check_bans(self, url: str) -> Optional[BanRule]:
        """Check if a URL matches any active ban rule. Returns the first matching rule."""
        current_time = time.time()
        for rule in self.ban_rules:
            # TODO 1: Skip rule if it's expired (current_time > rule.created_at + rule.ttl_seconds).
            # TODO 2: If rule.matches(url): return rule.
            #   For glob matching: fnmatch.fnmatch(url, rule.pattern)
```

```
            #   For regex: use a pre-compiled regex stored in the rule.
        return None

    async def _gc_ban_rules(self):
        """Background task to garbage collect expired ban rules."""
        while True:
            await asyncio.sleep(3600)  # Run every hour
            current_time = time.time()
            # TODO: Filter self.ban_rules, keeping only rules where current_time <= created_at + ttl_seconds.
```

Integration into EdgeRequestHandler (`edge/handler.py`):

```python
class EdgeRequestHandler:
    def __init__(self, cache: CacheStorage, upstream: str, invalidation_handler: InvalidationHandler):
        self.cache = cache
        self.upstream = upstream
        self.invalidation = invalidation_handler

    async def handle_request(self, request_headers: Dict[str, str], request_body: bytes) -> Tuple[int, Dict[str, str], bytes]:
        # ... existing code to parse request ...

        # TODO: Insert ban checking early in the flow (after generating the URL but before cache lookup).
        #   ban_rule = self.invalidation.check_bans(url)
        #   if ban_rule:
        #       if not ban_rule.is_soft:
        #           # Hard ban: delete cache entry if exists, then proceed as cache miss.
        #           self.invalidation._hard_purge_key(cache_key)  # cache_key may need URL only variant
        #       else:
        #           # Soft ban: mark existing entry stale.
        #           self.invalidation._soft_purge_key(cache_key)
        #       # Continue; the cache lookup will now miss or find a stale entry.

        # ... rest of the caching logic ...
```

E. Language-Specific Hints

- Use Python's `asyncio.Lock` or `asyncio.Queue` for coordinating background revalidation and preventing thundering herds.
- For pattern matching in bans, `fnmatch` is simpler and safer than regex for globs. If using regex, pre-compile them with `re.compile(pattern)` and store the compiled object in the `BanRule` to avoid compilation on every request.
- Use `asyncio.create_task` for fire-and-forget background operations like revalidation or garbage collection. Consider using a bounded semaphore (`asyncio.Semaphore`) to limit concurrent revalidation tasks.
- For the simple in-memory Pub/Sub, `asyncio.Queue` can also be used per channel, but the callback-based approach shown is more flexible.

F. Milestone Checkpoint To verify your cache invalidation implementation:

1. **Start the edge server** with an upstream origin (e.g., a simple static file server).
2. **Prime the cache:** Use `curl http://edge:8080/image.jpg` to fetch and cache an image.
3. **Test Hard Purge:**

```
curl -X PURGE -H "Authorization: Bearer admin-token" http://edge:8080/image.jpg
```

`Expected: `200 OK` response. Immediately request the image again. You should see a cache miss (check server logs) and the origin sh`

4. **Test Soft Purge:**

```
curl -X PURGE -H "Authorization: Bearer admin-token" -H "Soft-Purge: 1" http://edge:8080/image.jpg
```

`Expected: `200 OK`. Immediately request the image. You should be served the *stale* (cached) version, but a background revalidation`

5. **Test Tag-Based Purge:** First, ensure your origin returns a `Surrogate-Key: test` header. Fetch a few URLs with that tag. Then:

```
curl -X PURGE -H "Authorization: Bearer admin-token" http://edge:8080/purge/tag/test
```

`Expected: `200 OK` with a JSON body like `{"purged_keys_count": 3}`. All tagged entries should be purged.`

6. **Test Ban Rule:** Add a ban rule via API, then request a matching URL. The entry should be invalidated (either hard or soft). Check that expired ban rules are removed by the GC task after their TTL.

Signs of Trouble:

- **Purge does nothing:** Check cache key generation. Ensure the purge handler is using the same key logic as the cache store. Verify authentication.
- **Background revalidation causes duplicate origin requests:** Implement the revalidation lock per key as described in the pitfalls.
- **Ban rules slow down requests:** You are likely checking all rules linearly. Ensure you are using efficient matching (pre-compiled regex) and implement TTL-based garbage collection.

Component: Origin Shield & Request Collapsing (Milestone 3)

> **Milestone(s):** Milestone 3 (Origin Shield & Request Collapsing)

Mental Model: The Concert Vestibule

Picture a massive concert hall where the main auditorium (the **Origin Server**) can only accommodate a limited number of people at once. Thousands of fans (client requests) arrive at the venue at the same time, all heading to the same section of seats (requesting the same resource). If everyone rushed the main doors simultaneously, they would create a dangerous stampede, overwhelming the entrance and causing a collapse (the **thundering herd problem**).

Now, imagine a well-designed **vestibule** or lobby area just before the main hall doors. This is the **Origin Shield**. As fans arrive, the vestibule staff (the shield logic) quickly groups together everyone who has a ticket for the same seat (identical requests). Instead of letting 100 individuals each try to enter the main hall independently for the same seat, the staff selects one representative from the group (the **collapsed request**) to go and claim the seat. The other 99 wait patiently in the vestibule (their requests are **queued**). When the representative returns with confirmation and details about the seat (the HTTP response), the staff efficiently relays that information to the entire waiting group. This prevents a stampede, protects the main hall's capacity, and serves all fans efficiently.

This mental model captures the core purpose of the shield: it's a **deduplication and queueing layer** that sits between the noisy, high-volume edge nodes and the precious, capacity-constrained origin. It transforms a potential storm of concurrent, identical `GET` requests into a single, orderly fetch, dramatically reducing load on the origin.

Shield Logic & Request Collapsing

The Origin Shield is an HTTP proxy with its own cache (often called a **mid-tier** or **shield** cache). Its primary intelligence lies in handling cache misses. The following algorithm describes its behavior when an edge node forwards a request (a cache miss) to the shield.

1. **Request Arrival & Cache Check:** The shield receives an HTTP request from an edge node. It first checks its own local cache using the same `CacheEntry` data model and `CacheKey` generation logic as the edge (including `Vary` header dimensions). If a **fresh** entry exists, it is returned immediately—this is a **shield hit**.
2. **Cache Miss & Coalescing Map Lookup:** On a cache miss, the shield checks an internal, in-memory data structure we'll call the **Request Coalescing Map**. This map tracks which requests are currently "in flight" to the origin. The key is the normalized cache key for the request (URL + `Vary` header values). The value is a list of callback functions or futures representing all the client connections (edge nodes) waiting for the result of that specific request.
3. **Decision: New Origin Fetch or Join Existing Queue:**
   - If the cache key is **not** in the coalescing map, this is the first request for this resource since the last cached version expired. The shield creates a new entry in the map, initiates a single HTTP request to the origin server, and records the pending client connection. The request proceeds to step 4.
   - If the cache key **is** already in the coalescing map, it means an identical request is already being fetched from the origin. The shield simply adds the new client's callback to the list associated with that key. This request is now **collapsed**—it will wait for the shared result. The algorithm jumps to step 6.
4. **Origin Fetch with Protection:** The shield sends the request to the origin. Crucially, it implements protective measures:
   - **Connection Limits:** A semaphore or rate limiter ensures the total number of concurrent connections to the origin stays below a safe threshold.
   - **Timeouts & Circuit Breakers:** Individual requests have strict timeouts. If the origin becomes slow or fails, circuit breakers can trip to fail fast and prevent cascading failure.
5. **Response Processing & Cache Population:** Upon receiving the origin's response, the shield:

    - Determines if the response is cacheable using `is_response_cacheable`.
    - If cacheable, it creates a new `CacheEntry` (using `CacheEntry.from_upstream_response`), stores it in its cache, and records its TTL.
    - **Negative Caching:** If the response is an error (e.g., `404 Not Found` or `503 Service Unavailable`), the shield may still create a cache entry with a very short TTL (e.g., 5-10 seconds). This prevents a storm of requests for a non-existent or temporarily failing resource from hammering the origin.
6. **Fan-Out to Waiting Requests:** The shield retrieves the list of all waiting callbacks from the coalescing map for the given cache key. It then sends the HTTP response (either the freshly fetched one, a cached one, or a negatively cached error) to **every** waiting client connection.
7. **Cleanup:** Finally, the shield removes the entry for this cache key from the Request Coalescing Map. Any new requests for the same resource will now either be served from the shield's cache (if fresh) or trigger a new origin fetch cycle.

The state of a request at the shield can be summarized in the following table:

State	Description	Trigger for Transition
MISS (No Coalescing)	First request for an uncached resource.	Request arrives, cache key not in coalescing map.
IN_FLIGHT (Coalescing Active)	An origin fetch for this key is ongoing. Other identical requests are queued.	Subsequent identical requests arrive before the origin fetch completes.
CACHED (FRESH/STALE)	Response is stored in the shield's local cache.	Origin fetch completes and response is stored.
NEGATIVE_CACHED	An error response is stored with a short TTL.	Origin returns an error code deemed safe for negative caching.

The core data structure enabling this, the `RequestCoalescingMap`, has the following interface:

Method Name	Parameters	Returns	Description
`get_or_create_future`	`cache_key: str`	`Future`	If `cache_key` exists, returns its associated Future. If not, creates a new Future, stores it with the key, and returns it. The caller who creates the Future is responsible for fulfilling it.
`fulfill_future`	`cache_key: str`, `result: Tuple[int, Dict, bytes]`	`None`	Sets the result for the Future associated with `cache_key` and notifies all waiters. Removes the key from the map.
`fail_future`	`cache_key: str`, `exception: Exception`	`None`	Sets an exception for the Future, notifies waiters, and removes the key.

> **Key Insight:** The shield's value isn't just in its cache hit ratio—it's in its **miss efficiency**. A single origin request satisfying hundreds of simultaneous edge requests is a massive win for origin load and stability.

ADR: To Shield or Not To Shield

> **Decision: Deploy an Origin Shield as a Standard Tier for Multi-Edge CDNs**

- **Context:** Our CDN architecture involves multiple geographically distributed edge nodes. Without a shield, a sudden surge in traffic for an uncached resource (a "hot" article, a product launch) would cause all edge nodes to independently fetch the same content from the origin simultaneously. This "cache miss storm" can overload the origin, increase its latency, and risk its availability. We must decide whether to introduce an intermediate caching layer (the shield) to absorb and deduplicate these requests.
- **Options Considered:**
    1. **No Shield (Edge-to-Origin Direct):** All edge nodes connect directly to the origin.
    2. **Single Global Origin Shield:** Deploy one shield instance (or a cluster behind a load balancer) that all edge nodes use as their upstream.
    3. **Regional Shields:** Deploy multiple shield instances, each serving edge nodes in a specific geographic region (e.g., North America Shield, EU Shield).
- **Decision:** We will implement a **Single Global Origin Shield** as the primary architecture for this educational project. It provides the core learning benefits of request collapsing and origin protection with manageable complexity.
- **Rationale:**
    - **Simplicity vs. Benefit:** A single shield provides the vast majority of the origin protection benefit (request collapsing) with a much simpler operational and implementation model compared to a regionalized shield tier. It's the most effective "next step" after basic edge caching.
    - **Clear Learning Path:** Implementing request coalescing, negative caching, and connection queuing against a single upstream is a coherent, challenging milestone. Adding regionalization introduces networking and cache consistency complexities that are better suited for a later extension.
    - **Effective for Many Scenarios:** For origins that are not themselves globally distributed, a single, well-provisioned shield located in the same data center or region as the origin can eliminate nearly all duplicate traffic, as network latency from the shield to the origin is minimal.
- **Consequences:**

- **Added Latency:** Edge nodes far from the shield's location will experience higher latency on cache misses compared to connecting directly to a nearby origin. This is the primary trade-off.
- **Single Point of Failure:** The shield becomes a critical component. Its failure would cause all cache misses to fail. This necessitates making the shield itself highly available (e.g., via a cluster with a load balancer) and implementing robust health checks and failover logic at the edge (e.g., failover to origin if the shield is unhealthy).
- **Cache Warming:** The shield's cache acts as a global warming layer. Once one edge node populates the shield cache, all other edges benefit, reducing overall origin load more effectively than isolated edge caches.

Option	Pros	Cons	Chosen?
No Shield	Lowest latency for cache misses (direct path). Simplest architecture.	Extremely vulnerable to thundering herd problem. High, unbounded load on origin.	No
Single Global Shield	Excellent request collapsing. Dramatically reduces origin load. Clear, focused implementation milestone.	Introduces a hop of latency for distant edges. Creates a new critical component to manage.	**Yes**
Regional Shields	Good request collapsing within regions. Lower latency for cache misses than a single global shield.	Significantly higher complexity (multiple shield caches, potential for inconsistency). Higher operational cost.	No (Consider as a Future Extension)

Common Pitfalls in Shielding

⚠️ Pitfall 1: Cache Poisoning Between Edge Variants

- **Description:** Edge nodes in different regions might request the same URL but with different `Accept-Encoding` headers (e.g., `gzip` vs. `br`). If the shield's cache key doesn't include all `Vary` header dimensions (just like the edge), it might store and serve a `gzip` response to an edge that requested `brotli`, causing corruption.
- **Why it's Wrong:** The HTTP response body is incompatible with the client's expectation. The edge node might serve incorrect, unreadable content to the user.
- **Fix:** The shield **must** use the exact same cache key generation logic as the edge, meticulously incorporating all headers listed in the origin's `Vary` response header. Reuse the `CacheKey` generation module from Milestone 1.

⚠️ Pitfall 2: Timeout Mismatch Chain Reaction

- **Description:** The edge node has a 30-second client timeout. The shield sets a 10-second timeout to the origin. If the origin is slow and takes 15 seconds to respond, the shield's request fails (timeout). However, 100 edge requests were collapsed behind that single shield request. The shield fails all 100 waiting connections simultaneously. All 100 edge nodes, now seeing a failed upstream (shield), may simultaneously decide to retry directly to the origin or to another shield, creating a new, smaller herd problem.
- **Why it's Wrong:** It negates the benefit of request collapsing and can amplify failures.
- **Fix:** Implement a **graceful degradation** strategy. Use `stale-if-error` directives. If the shield's fetch times out or fails, it should check if it has a **stale** cached version of the resource. If so, it can serve that stale content immediately while logging the error, rather than failing all clients. Additionally, ensure shield-to-origin timeouts are configured to be meaningfully shorter than edge-to-client timeouts.

⚠️ Pitfall 3: Ignoring Shield Health in Edge Routing

- **Description:** The edge node is configured to always forward misses to the shield's hostname. If the shield instance crashes or becomes partitioned, the edge will still send it requests, which will fail or timeout, degrading performance for all users.
- **Why it's Wrong:** It eliminates the fault-tolerance benefits of a multi-tier architecture. A failure in the shield tier should not cause a full outage.
- **Fix:** Implement active **health checking** at the edge. Periodically (e.g., every 5 seconds) make a lightweight request (e.g., `GET /health`) to the shield. If consecutive health checks fail, mark the shield as unhealthy and temporarily **bypass** it, sending cache misses directly to the origin for a defined period. Re-introduce the shield after it passes health checks again.

⚠️ Pitfall 4: Unbounded Growth of the Coalescing Map

- **Description:** The in-memory map that tracks in-flight requests (`RequestCoalescingMap`) is never cleaned up. If an origin fetch hangs forever (due to an origin bug or network issue), the entry remains, and all subsequent requests for that resource pile up in memory indefinitely, causing a memory leak.
- **Why it's Wrong:** Leads to eventual out-of-memory crashes of the shield process.
- **Fix:** Associate every entry in the coalescing map with a **timeout**. If the origin fetch does not complete within this timeout, the future is failed (e.g., with a `TimeoutError`), and the entry is removed from the map. Also, implement a periodic garbage collection task that scans for and removes stale entries.

Implementation Guidance for Origin Shield

A. Technology Recommendations Table

Component	Simple Option (for Learning)	Advanced Option (for Production)
HTTP Server/Proxy	Python's `asyncio` with `aiohttp` (easy concurrency for request collapsing)	Go's `net/http` or Rust's `hyper` (performance, robustness)
Request Coalescing	In-memory `dict` with `asyncio.Future` / `asyncio.Event`	Distributed coordination (Redis, memcached) for multi-process shields
Connection Pool/Limiting	`aiohttp.TCPConnector` with limit	Adaptive concurrency limiting (AIMD, gradient)
Health Checking	Simple periodic HTTP `GET`	Rich health checks with metrics integration (Prometheus)

B. Recommended File/Module Structure

```
blue_origin_cdn/
├── pyproject.toml
├── src/
│   └── blue_origin/
│       ├── __init__.py
│       ├── edge/                   # Milestone 1 & 5
│       │   ├── __init__.py
│       │   ├── handler.py          # EdgeRequestHandler
│       │   ├── cache.py            # CacheStorage, CacheEntry
│       │   └── compression.py      # gzip/brotli middleware
│       ├── shield/                 # Milestone 3
│       │   ├── __init__.py
│       │   ├── handler.py          # ShieldRequestHandler
│       │   ├── coalescing.py       # RequestCoalescingMap
│       │   ├── client.py           # ShieldHTTPClient (with queue/limit)
│       │   └── health.py           # Health check endpoint & logic
│       ├── invalidation/           # Milestone 2
│       │   └── ...
│       ├── routing/                # Milestone 4
│       │   └── ...
│       ├── analytics/              # Milestone 5
│       │   └── ...
│       └── utils/
│           └── http.py             # parse_http_date, is_response_cacheable, etc.
├── configs/
│   └── shield_config.yaml
└── scripts/
    └── run_shield.py
```

C. Infrastructure Starter Code

The following is a complete, working implementation of the core `RequestCoalescingMap` using `asyncio`. This is a prerequisite component that learners can use directly.

PYTHON

```python
# src/blue_origin/shield/coalescing.py

import asyncio

from typing import Dict, Tuple, Optional, Any

import time


class RequestCoalescingMap:

    """

    A map that collapses concurrent requests for the same cache key.

    Each key maps to a Future that will hold the result (status, headers, body).

    """

    def __init__(self, default_timeout: float = 30.0):

        self._futures: Dict[str, asyncio.Future] = {}

        self._timeouts: Dict[str, float] = {}  # key -> expiration time

        self._default_timeout = default_timeout

        self._lock = asyncio.Lock()

        self._gc_task: Optional[asyncio.Task] = None


    async def get_or_create_future(self, cache_key: str) -> asyncio.Future:

        """

        Get the existing Future for `cache_key`, or create a new one.

        The caller that creates the Future MUST fulfill or fail it.

        """

        async with self._lock:

            if cache_key in self._futures:

                return self._futures[cache_key]

            # Create a new future

            fut = asyncio.Future()

            self._futures[cache_key] = fut

            self._timeouts[cache_key] = time.time() + self._default_timeout

            # Start GC task if not running

            if self._gc_task is None or self._gc_task.done():

                self._gc_task = asyncio.create_task(self._gc_loop())

            return fut


    async def fulfill_future(self,

                             cache_key: str,

                             result: Tuple[int, Dict[str, str], bytes]) -> None:

        """Set the result for the Future associated with `cache_key`."""

        async with self._lock:

            fut = self._futures.pop(cache_key, None)

            self._timeouts.pop(cache_key, None)

        if fut and not fut.done():
```

```
            fut.set_result(result)

    async def fail_future(self, cache_key: str, exception: Exception) -> None:
        """Set an exception for the Future associated with `cache_key`."""
        async with self._lock:
            fut = self._futures.pop(cache_key, None)
            self._timeouts.pop(cache_key, None)
        if fut and not fut.done():
            fut.set_exception(exception)

    async def _gc_loop(self, interval: float = 5.0) -> None:
        """Background task to clean up timed-out futures."""
        while True:
            await asyncio.sleep(interval)
            now = time.time()
            to_remove = []
            async with self._lock:
                for key, exp_time in self._timeouts.items():
                    if now > exp_time:
                        to_remove.append(key)
                for key in to_remove:
                    fut = self._futures.pop(key, None)
                    self._timeouts.pop(key, None)
                    if fut and not fut.done():
                        fut.set_exception(asyncio.TimeoutError(f"Request for {key} timed out"))
```

D. Core Logic Skeleton Code

Here is the skeleton for the main shield request handler. Learners must fill in the detailed logic.

```python
# src/blue_origin/shield/handler.py

import asyncio

from typing import Dict, Tuple, Optional

from .coalescing import RequestCoalescingMap

from ..edge.cache import CacheStorage, CacheEntry, generate_cache_key

from ..utils.http import (

    is_response_cacheable,

    parse_cache_control_header,

    CacheDirectives,

    get_header_values

)


class ShieldRequestHandler:

    """

    HTTP request handler for the Origin Shield.

    """

    def __init__(self,

                 cache: CacheStorage,

                 origin_upstream: str,

                 coalescing_map: RequestCoalescingMap,

                 max_concurrent_origin_requests: int = 100):

        self.cache = cache

        self.origin_upstream = origin_upstream.rstrip('/')

        self.coalescing_map = coalescing_map

        # Semaphore to limit concurrent origin requests

        self._origin_semaphore = asyncio.Semaphore(max_concurrent_origin_requests)


    async def handle_request(self,

                             request_headers: Dict[str, str],

                             request_body: bytes) -> Tuple[int, Dict[str, str], bytes]:

        """

        Main request handling algorithm for the shield.

        Called by the shield's HTTP server for each incoming request (from an edge node).

        """

        # TODO 1: Extract the request method, URL, and headers.

        #   - The URL is in request_headers[':path'] (or similar, depending on your HTTP lib).

        #   - The method is in request_headers[':method'].


        # TODO 2: Generate the cache key for this request.

        #   - Use the same `generate_cache_key` function as the edge node.

        #   - It must consider the `Vary` header from the *original client request*

        #     which should be forwarded by the edge. Assume it's in a header like `X-Forwarded-Vary`.
```

```
    # TODO 3: Check the shield's local cache for a fresh entry.
    #   - Use `self.cache.get(cache_key)`.
    #   - If found and `entry.is_fresh(current_time)` is True, return the cached response.
    #   - If found but stale, check `stale-while-revalidate` directive.
    #     If allowed, serve stale and trigger background revalidation (TODO 8).

    # TODO 4: Check the Request Coalescing Map.
    #   - Call `self.coalescing_map.get_or_create_future(cache_key)`.
    #   - If this returns an *existing* Future, it means a request is already in flight.
    #     - Wait on that Future (`await future`).
    #     - Return the result it provides. This is the collapsed request path.

    # TODO 5: If we are here, we are the first/only request (we created the Future).
    #   Now we must fetch from the origin.
    #   - Acquire the origin semaphore: `async with self._origin_semaphore:`.
    #   - Make an HTTP request to `self.origin_upstream + url` with the appropriate headers.
    #   - Enforce a timeout (e.g., 10 seconds).

    # TODO 6: Process the origin's response.
    #   - If the request succeeded (status code 2xx, 3xx, 404, etc.):
    #     a. Determine if response is cacheable via `is_response_cacheable`.
    #     b. If cacheable, create a `CacheEntry` and store it in `self.cache`.
    #     c. If it's an error (4xx, 5xx), consider negative caching with a short TTL.
    #   - If the request failed (network error, timeout):
    #     Consider serving stale content from cache if available (graceful degradation).

    # TODO 7: Fulfill the Future in the coalescing map.
    #   - Call `self.coalescing_map.fulfill_future(cache_key, result)` where result is
    #     (status_code, headers, body).
    #   - This will wake up all other requests waiting on this same cache key.

    # TODO 8: (Background) If serving stale content under `stale-while-revalidate`,
    #   spawn a background task to revalidate the cache entry with the origin.
    #   - Use `asyncio.create_task(self._revalidate_in_background(cache_key, stale_entry, request_headers))`.

    # TODO 9: Return the HTTP response (status, headers, body) to the caller.
    pass  # Placeholder return (will be filled by learner)
    return 500, {"Content-Type": "text/plain"}, b"Not implemented"

async def _revalidate_in_background(self,
                                    cache_key: str,
                                    stale_entry: CacheEntry,
                                    request_headers: Dict[str, str]) -> None:
```

```
        """
        Asynchronously revalidate a stale cache entry with the origin.
        If successful, update the cache with the new entry.
        """
        # TODO 1: Make a conditional request to the origin.
        #   - Include `If-None-Match: stale_entry.etag` and/or `If-Modified-Since: stale_entry.last_modified`.
        #   - Use the same request headers as the original (but potentially with conditional headers).

        # TODO 2: If origin returns 304 Not Modified, update the `fetched_at` and `expires_at`
        #   of the stale_entry to mark it fresh again, and store it back in the cache.

        # TODO 3: If origin returns a new response (200 OK), store it as a new CacheEntry,
        #   replacing the old one.

        # TODO 4: If the origin request fails, log the error. The stale entry remains stale.
        pass
```

E. Language-Specific Hints (Python)

- **Concurrency:** Use `asyncio` throughout the shield. The `aiohttp` library provides excellent asynchronous HTTP client and server capabilities. The `RequestCoalescingMap` heavily relies on `asyncio.Future` for synchronization.
- **Timeouts:** Always use `asyncio.wait_for` or the `timeout` parameter in `aiohttp` client calls to prevent hung requests from consuming resources indefinitely.
- **Graceful Shutdown:** Ensure your shield server has a shutdown handler that waits for in-flight origin requests to complete (or times them out) before exiting.
- **Metrics Integration:** Use the `ShieldMetrics` data structure to track counts of collapsed requests, origin errors, and queue sizes. Export these via a `/metrics` endpoint for monitoring.

F. Milestone Checkpoint

After implementing the `ShieldRequestHandler` and the `RequestCoalescingMap`, you can verify basic functionality with the following steps:

1. **Start the Origin:** Run a simple static file server (e.g., `python -m http.server 8000`).
2. **Start the Shield:** Run your shield server, configured to upstream to `http://localhost:8000`.

```bash
python scripts/run_shield.py --config configs/shield_config.yaml
```

3. **Simulate Concurrent Edge Requests:** Write a small test script that uses `asyncio` to send 10 identical `GET` requests to the shield at the same time.

```python
import aiohttp
import asyncio

async def fetch(session, url):
    async with session.get(url) as resp:
        return await resp.text()

async def main():
    url = "http://localhost:8080/shield/path/to/image.jpg"
    async with aiohttp.ClientSession() as session:
        tasks = [fetch(session, url) for _ in range(10)]
        responses = await asyncio.gather(*tasks)
        print(f"Sent 10 concurrent requests. All received responses.")
        # Verify all responses are identical
        assert all(r == responses[0] for r in responses)

asyncio.run(main())
```

4. **Expected Behavior:**
   - The shield's access log should show only **one** request to the origin (`localhost:8000`) for `/path/to/image.jpg` .
   - The origin server's log should also show only **one** request.
   - All 10 client requests should receive the same, correct response.
   - The shield's metrics (if exposed) should show `collapsed_requests: 9` .
5. **Signs of Trouble:**
   - **Origin sees 10 requests:** The request collapsing logic is not working. Check the `RequestCoalescingMap` implementation and the logic in `handle_request` steps 4 and 5.
   - **Clients receive errors or hang:** The Future fulfillment logic (`fulfill_future` / `fail_future`) may be broken, or the shield's HTTP client might be failing. Check error handling and timeouts.
   - **Responses differ:** Cache key generation is inconsistent, possibly not handling `Vary` headers correctly. Ensure the edge node forwards the necessary headers (like `Accept-Encoding`) to the shield.

Component: Edge Node Distribution & Routing (Milestone 4)

> **Milestone(s):** Milestone 4 (Edge Node Distribution & Routing)

Mental Model: The Air Traffic Control Map

Imagine a global air traffic control system for web requests. Planes (user requests) depart from airports worldwide (user locations) and need to land at the nearest available, operational airport (edge node) for their destination. The air traffic controller (routing system) must:

1. **Know all airports**: Maintain a real-time map of all edge nodes, their geographic locations, and operational status
2. **Direct to nearest**: Calculate the geographically closest operational airport for each incoming flight based on its departure point (user IP)
3. **Handle emergencies**: When an airport closes (node fails), redirect incoming flights to the next-nearest airport within seconds
4. **Balance traffic**: Evenly distribute flights across airports with similar proximity to prevent congestion

This mental model captures the essence of a multi-node CDN: we have multiple geographically distributed edge servers (airports), clients make requests from various locations (flights departing), and our routing system (air traffic control) must intelligently direct each request to the optimal edge node based on proximity and health. Just as a flight from Tokyo shouldn't be directed to an airport in London when there's one in Osaka, a user in Japan shouldn't be served from a European edge node when there's a Tokyo PoP available.

Geo-Routing & Health Checking

Geo-routing ensures users connect to the geographically closest edge node, minimizing network latency. Health checking ensures we don't route traffic to nodes that are malfunctioning or offline.

How Client IPs Map to Edge Nodes

The mapping process follows this algorithm:

1. **IP Geolocation**: When a request arrives (or during DNS resolution), we determine the client's approximate geographic location from their IP address using a GeoIP database
2. **Node Location Database**: Each edge node registers its physical location (latitude/longitude or region code) with the control plane during startup
3. **Distance Calculation**: For each healthy edge node, calculate the geographic distance to the client using the Haversine formula (great-circle distance)
4. **Selection**: Select the edge node with the smallest distance that also meets capacity and health constraints
5. **Fallback**: If the nearest node is unhealthy, select the next nearest healthy node

The geographic mapping can be implemented at different layers:

Mapping Layer	How It Works	Pros	Cons
DNS-Based	DNS server returns different A/AAAA records based on the resolver's IP location	Simple, works with any client	Depends on client's DNS resolver location, not end-user location
Anycast	Same IP address announced from multiple locations; BGP routes to nearest	Transparent to client, fastest failover	Complex BGP configuration, all nodes share same IP
HTTP Redirect	Central gateway receives all requests, responds with 302 to nearest edge	Most accurate (sees real client IP), flexible	Adds extra round-trip for first request

Health Checking Implementation

Health checking is a continuous process that monitors each edge node's operational status:

Passive Health Checking: Monitor actual traffic patterns

- Track error rates (5xx responses) from the node
- Monitor request latency percentiles
- Observe TCP connection failures

Active Health Checking: Periodic synthetic requests

- HTTP GET to `/health` endpoint on each edge node
- Validate response status code (200 OK) and optionally content
- Check within configurable intervals (e.g., every 10 seconds)
- Consider network path health (latency, packet loss)

The health state machine for an edge node follows these transitions:

Current State	Event	Next State	Action Taken
HEALTHY	3 consecutive health checks fail	**UNHEALTHY**	Remove from DNS rotation, stop routing new traffic
UNHEALTHY	2 consecutive health checks succeed	**DEGRADED**	Add back to rotation with reduced traffic weight
DEGRADED	5 consecutive health checks succeed	**HEALTHY**	Restore full traffic weight
DEGRADED	1 health check fails	**UNHEALTHY**	Remove from rotation again
Any state	Manual admin command (disable)	**MAINTENANCE**	Gracefully drain connections, no new traffic
MAINTENANCE	Manual admin command (enable)	**HEALTHY**	Add to rotation, full weight

Health check responses include detailed metrics:

```python
# Example health check response structure
{
  "status": "healthy",
  "timestamp": 1678901234.567,
  "metrics": {
    "cache_hit_ratio": 0.89,
    "request_rate": 1250.5,
    "error_rate": 0.001,
    "cpu_utilization": 0.45,
    "memory_used_mb": 2048,
    "active_connections": 347
  },
  "version": "1.2.3",
  "region": "us-west-2"
}
```

PYTHON

Consistent Hashing for Cache Distribution

When we have multiple edge nodes, we face a critical decision: should each node cache independently (edge-local caching) or should we distribute cache entries across nodes (shared caching)? For our educational CDN, we implement **consistent hashing** to distribute cache load while minimizing reshuffling when nodes join or leave.

The Problem: Cache Distribution & Reshuffling

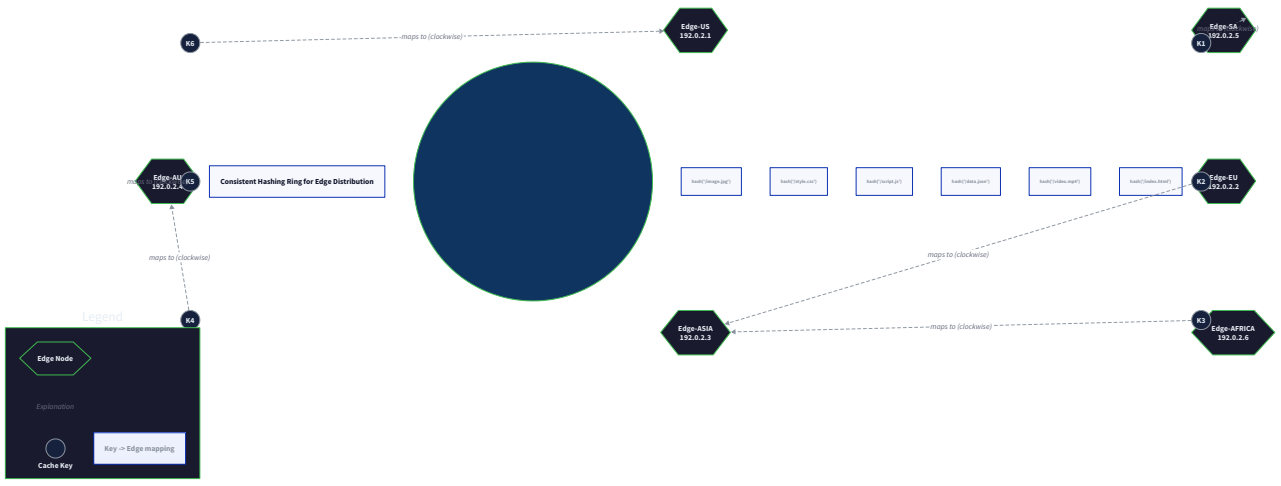
Consider three approaches:

1. **Independent Caches**: Each edge node caches what it serves. Simple but inefficient - popular content gets cached redundantly on every node that serves it, wasting memory. Cache hit ratio lower due to divided cache space.
2. **Centralized Cache**: All edge nodes query a shared cache cluster. Efficient memory use but single point of failure, adds latency.
3. **Distributed with Hashing**: Each URL maps to a specific edge node via hashing; that node becomes the "owner" of that cached content. Other nodes can fetch from the owner on cache miss.

The naive hashing approach (`hash(url) % num_nodes`) has a major problem: when nodes are added or removed, nearly all URLs remap to different nodes, causing massive cache invalidation (the "reshuffling problem").

Consistent Hashing Solution

Consistent hashing solves the reshuffling problem by hashing both nodes and keys onto a virtual ring:



Algorithm:

1. Create a virtual circle representing the hash space (0 to 2^m - 1, where m is typically 160 for SHA-1)
2. For each edge node, compute multiple hash points (virtual nodes) around the circle (e.g., 100-200 replicas per physical node)
3. For each cache key (URL + Vary headers), compute its hash and find the next hash point clockwise around the circle
4. The node owning that hash point is responsible for caching that content

When a node joins:

1. Compute virtual node positions for the new node
2. Only keys that hash between the new node's positions and the previous node's positions get reassigned
3. This affects only `1/(n+1)` of keys on average, not all keys

When a node fails:

1. Its virtual nodes are removed from the ring
2. Keys that mapped to those virtual nodes now map to the next node clockwise
3. Affected keys can be re-fetched from origin or replicated in advance

Data Structures for Consistent Hashing:

Structure Name	Fields	Description
`HashRingNode`	`node_id: str`, `physical_node: EdgeNode`, `position: int`	A virtual node on the hash ring
`EdgeNode`	`node_id: str`, `region: str`, `address: str`, `weight: int`, `health_status: NodeHealth`	Physical edge node information
`ConsistentHashRing`	`nodes: SortedDict[int, HashRingNode]`, `replicas_per_node: int`, `hash_function: Callable`	Main hash ring structure

The `ConsistentHashRing` interface:

Method	Parameters	Returns	Description
`add_node`	`node: EdgeNode`	`None`	Add a physical node with virtual replicas to the ring
`remove_node`	`node_id: str`	`None`	Remove a node and all its virtual replicas
`get_node`	`key: str`	`EdgeNode`	Find the node responsible for the given key
`get_replica_nodes`	`key: str`, `count: int`	`List[EdgeNode]`	Get the primary node plus N-1 successor nodes for replication

> **Key Insight**: Virtual nodes (replicas) serve two purposes: 1) They provide better load distribution by preventing clustering, and 2) They allow weighting - nodes with more capacity get more virtual nodes, receiving proportionally more keys.

ADR: Client Routing Mechanism

> **Decision: Hybrid DNS-Based Routing with HTTP Fallback**
>
> **Context**: We need to direct clients to the nearest healthy edge node. The routing must work with unmodified web browsers (standard HTTP/HTTPS), provide fast failover (<5 seconds), and handle GeoIP inaccuracy gracefully. We're building an educational CDN, not a commercial anycast network.
>
> **Options Considered**:
>
> 1. **Pure DNS-based geo-routing**: DNS server returns different A records based on resolver IP location
> 2. **Anycast BGP routing**: Same IP advertised from all locations, BGP routes to nearest
> 3. **HTTP redirect-based**: All clients hit central endpoint, get 302 redirected to nearest edge
> 4. **Hybrid DNS+HTTP**: DNS returns multiple edges, client tries nearest, falls back via 302
>
> **Decision**: Implement hybrid DNS-based routing with HTTP fallback. DNS returns 2-3 nearest edge IPs ordered by proximity. Client tries the first; if unhealthy (connection fails or receives 503), tries the next. For cache misses that need to reach a different edge (due to consistent hashing), the serving edge issues an HTTP 302 to the correct edge.
>
> **Rationale**:
>
> - DNS-based routing works with all clients without special software
> - Multiple IPs in DNS provide client-side failover when first edge is down
> - HTTP redirects handle cases where GeoIP is wrong or cache location differs from optimal edge
> - Avoids BGP anycast complexity (requires ISP relationships, not feasible for educational project)
> - Provides flexibility: we can implement smart routing logic at both DNS and HTTP layers
>
> **Consequences**:
>
> - Adds one extra HTTP redirect for cache misses that hash to different edge than client's nearest
> - DNS TTL (Time-To-Live) must be balanced: too short causes excessive DNS queries, too long delays failover
> - Need to maintain both DNS infrastructure and HTTP redirect logic
> - Clients with broken DNS caching may experience suboptimal routing

Comparison of Routing Options:

Option	Pros	Cons	Chosen?
DNS-Based Geo-Routing	Simple, works everywhere, scales well	GeoIP inaccuracy, DNS caching delays failover, all-or-nothing	Partially - as primary
Anycast BGP	Transparent, fastest failover, optimal routing	Complex setup, requires ISP partnerships, expensive	No - too complex for educational
HTTP Redirect	Most accurate (sees real client IP), flexible control	Extra round-trip, breaks some client assumptions	Partially - as fallback
Hybrid DNS+HTTP	Best of both: DNS for speed, HTTP for accuracy	Implementation complexity, two systems to maintain	**Yes** - optimal balance

Common Pitfalls in Distribution

⚠️ Pitfall: GeoIP Database Inaccuracy

- **Description**: Assuming GeoIP databases are perfectly accurate. In reality, they're often wrong by hundreds of miles, especially for mobile networks and VPNs.
- **Why It's Wrong**: A user in New York might be mapped to a Los Angeles edge node due to ISP routing through California, adding 100ms+ latency.
- **How to Fix**:
  1. Use multiple GeoIP databases and cross-reference
  2. Implement latency-based verification: ping potential edges from client location
  3. Allow manual override via cookie or URL parameter for testing
  4. Use HTTP redirect as correction mechanism when wrong edge is selected

⚠️ Pitfall: Thundering Herd on Edge Failover

- **Description**: When an edge node fails, all its traffic immediately shifts to the next-nearest node, potentially overloading it.
- **Why It's Wrong**: The backup node might have 50% spare capacity, but gets 100% more traffic instantly, causing cascading failure.
- **How to Fix**:
  1. Implement gradual traffic shift: redirect 10% of users every second, not all at once

2. Monitor backup node load and throttle redirects if approaching capacity
3. Use multiple backup nodes with weighted distribution
4. Implement client-side exponential backoff when connecting to overloaded nodes

⚠️ Pitfall: Cache Inconsistency ("Cold Starts") on New Nodes

- **Description**: When a new edge node is added to the cluster, it starts with empty cache. All requests become cache misses until it warms up.
- **Why It's Wrong**: Performance degrades just when trying to add capacity, and origin gets hammered by cache fill requests.
- **How to Fix**:
  1. Implement cache warming: pre-fetch popular content before adding to rotation
  2. Use consistent hashing with replication: new node can fetch from neighboring nodes
  3. Gradual traffic ramp-up: start with 1% of traffic, increase as cache fills
  4. Share cache data between nodes for hottest content

⚠️ Pitfall: DNS TTL Misconfiguration

- **Description**: Setting DNS TTL too high (hours) delays failover; setting too low (seconds) causes excessive DNS load.
- **Why It's Wrong**: With 1-hour TTL, clients continue trying dead edge for up to an hour. With 5-second TTL, DNS servers get hammered.
- **How to Fix**:
  1. Use adaptive TTL: 300 seconds normally, 30 seconds when changes are pending
  2. Implement DNS-based health checks that override TTL for unhealthy nodes
  3. Use HTTP-level failover as backup when DNS fails
  4. Consider EDNS Client Subnet for more accurate GeoIP at DNS level

⚠️ Pitfall: Ignoring Network Topology

- **Description**: Routing based solely on geographic distance, ignoring network congestion, peering arrangements, and ISP relationships.
- **Why It's Wrong**: An edge 50 miles away through a congested peer might be slower than one 200 miles away through a direct fiber link.
- **How to Fix**:
  1. Augment geographic distance with latency measurements
  2. Implement network-aware routing that considers AS (Autonomous System) paths
  3. Use real-time latency data from monitoring to adjust routing weights
  4. Consider cost: some network paths might be more expensive despite being faster

Implementation Guidance for Distribution

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
GeoIP Database	MaxMind GeoLite2 (free)	MaxMind GeoIP2 Enterprise or IPinfo.io API
DNS Server	Python `dnslib` library for custom DNS	PowerDNS with GeoIP backend or AWS Route 53
Health Checking	HTTP endpoint with simple status	Complex health checks with dependency verification
Consistent Hashing	In-memory hash ring with virtual nodes	Ring with replication, persistence, and load balancing
Node Discovery	Static configuration file	etcd or ZooKeeper for dynamic registration

B. Recommended File/Module Structure

```
blue_origin/
├── edge/
│   ├── __init__.py
│   ├── server.py              # Edge HTTP server (from Milestone 1)
│   ├── cache.py              # Cache storage (from Milestone 1)
│   └── health.py             # Health check endpoint
├── routing/
│   ├── __init__.py
│   ├── geo.py                # GeoIP lookup utilities
│   ├── hash_ring.py          # Consistent hashing implementation
│   ├── dns_server.py         # Custom DNS server for geo-routing
│   └── node_registry.py      # Edge node registration and discovery
├── control/
│   ├── __init__.py
│   ├── plane.py              # Control plane logic
│   └── metrics_aggregator.py # Collect metrics from edges
└── config/
    ├── __init__.py
    ├── edge.yaml             # Edge node configuration
    └── routing.yaml          # Routing configuration
```

C. Infrastructure Starter Code

Complete Health Check Endpoint:

PYTHON

```python
# routing/health.py

import time

import json

import asyncio

from dataclasses import dataclass, asdict

from typing import Dict, Any, Optional

from enum import Enum


class NodeHealth(Enum):

    HEALTHY = "healthy"

    DEGRADED = "degraded"

    UNHEALTHY = "unhealthy"

    MAINTENANCE = "maintenance"


@dataclass

class HealthMetrics:

    """Metrics reported in health check response."""

    cache_hit_ratio: float = 0.0

    request_rate: float = 0.0  # requests per second

    error_rate: float = 0.0    # error rate (0.0-1.0)

    cpu_utilization: float = 0.0  # 0.0-1.0

    memory_used_mb: int = 0

    memory_total_mb: int = 0

    active_connections: int = 0

    cache_size_bytes: int = 0

    cache_capacity_bytes: int = 0



    def to_dict(self) -> Dict[str, Any]:

        return asdict(self)


class HealthChecker:

    """Manages health state and provides health check endpoint."""



    def __init__(self, node_id: str, region: str, version: str):

        self.node_id = node_id

        self.region = region

        self.version = version

        self.health_status = NodeHealth.HEALTHY

        self.metrics = HealthMetrics()

        self.start_time = time.time()

        self._check_interval = 10  # seconds

        self._check_task: Optional[asyncio.Task] = None
```

```python
async def start(self):
    """Start background health monitoring."""
    self._check_task = asyncio.create_task(self._monitor_loop())

async def stop(self):
    """Stop health monitoring."""
    if self._check_task:
        self._check_task.cancel()
        try:
            await self._check_task
        except asyncio.CancelledError:
            pass

async def _monitor_loop(self):
    """Background task to update health metrics."""
    while True:
        try:
            await self._update_metrics()
            await self._evaluate_health()
        except Exception as e:
            print(f"Health monitor error: {e}")
        await asyncio.sleep(self._check_interval)

async def _update_metrics(self):
    """Update health metrics from system monitoring."""
    # In a real implementation, this would collect:
    # - Cache statistics from cache storage
    # - System metrics (CPU, memory)
    # - Network connection counts
    # For now, we'll use placeholder values
    self.metrics.cache_hit_ratio = 0.85  # Example
    self.metrics.request_rate = 1250.5   # Example
    self.metrics.error_rate = 0.001      # Example
    self.metrics.active_connections = 347 # Example

async def _evaluate_health(self):
    """Evaluate overall health status based on metrics."""
    if self.metrics.error_rate > 0.1:  # 10% error rate
        self.health_status = NodeHealth.UNHEALTHY
```

```
        elif self.metrics.error_rate > 0.05:  # 5% error rate
            self.health_status = NodeHealth.DEGRADED
        elif self.metrics.cpu_utilization > 0.9:  # 90% CPU
            self.health_status = NodeHealth.DEGRADED
        else:
            self.health_status = NodeHealth.HEALTHY

    async def handle_health_request(self) -> tuple[int, Dict[str, str], bytes]:
        """Handle HTTP GET /health request."""
        uptime = time.time() - self.start_time

        response_data = {
            "status": self.health_status.value,
            "node_id": self.node_id,
            "region": self.region,
            "version": self.version,
            "timestamp": time.time(),
            "uptime_seconds": uptime,
            "metrics": self.metrics.to_dict()
        }

        headers = {
            "Content-Type": "application/json",
            "Cache-Control": "no-cache, no-store, must-revalidate"
        }

        return 200, headers, json.dumps(response_data).encode('utf-8')

    def get_health_status(self) -> NodeHealth:
        """Get current health status."""
        return self.health_status
```

D. Core Logic Skeleton Code

Consistent Hash Ring Implementation:

PYTHON

```python
# routing/hash_ring.py

import bisect

import hashlib

from typing import Dict, List, Optional, Tuple

from dataclasses import dataclass


@dataclass

class EdgeNode:

    """Represents a physical edge node."""

    node_id: str

    region: str

    address: str  # IP:port

    weight: int = 100  # Relative capacity (100 = normal)

    health_status: str = "healthy"


@dataclass

class HashRingNode:

    """A virtual node on the consistent hash ring."""

    node_id: str

    physical_node: EdgeNode

    position: int  # Hash position on the ring (0 to 2^m-1)


class ConsistentHashRing:

    """

    Consistent hashing implementation with virtual nodes.


    The ring maps cache keys to edge nodes for distributed caching.

    """


    def __init__(self, replicas_per_node: int = 100, hash_bits: int = 160):

        """

        Initialize an empty hash ring.


        Args:

            replicas_per_node: Number of virtual nodes per physical node

            hash_bits: Number of bits in hash space (default 160 for SHA-1)

        """

        self.replicas_per_node = replicas_per_node

        self.hash_bits = hash_bits

        self.hash_space = 2 ** hash_bits

        self.ring: Dict[int, HashRingNode] = {}  # position -> virtual node

        self.sorted_positions: List[int] = []  # Sorted list of positions
```

```
        self.nodes: Dict[str, EdgeNode] = {}  # node_id -> physical node

    def _hash(self, key: str) -> int:
        """
        Hash a string key to a position on the ring.

        Args:
            key: String to hash

        Returns:
            Integer position in hash space
        """
        # TODO 1: Use SHA-1 hash function (or configurable)
        # TODO 2: Convert hash digest to integer in range [0, hash_space-1]
        # TODO 3: Return the integer position
        pass

    def add_node(self, node: EdgeNode) -> None:
        """
        Add a physical node to the hash ring.

        Creates multiple virtual nodes (replicas) for better distribution.

        Args:
            node: EdgeNode to add
        """
        # TODO 1: Check if node already exists, return if it does
        # TODO 2: Calculate number of replicas based on node weight
        # TODO 3: For each replica i in range(num_replicas):
        #   - Create key: f"{node.node_id}:{i}"
        #   - Hash the key to get position
        #   - Create HashRingNode
        #   - Add to self.ring and self.sorted_positions
        # TODO 4: Sort self.sorted_positions for binary search
        # TODO 5: Add physical node to self.nodes dict
        pass

    def remove_node(self, node_id: str) -> Optional[EdgeNode]:
        """
        Remove a node from the hash ring.
```

```
        Args:
            node_id: ID of node to remove

        Returns:
            Removed EdgeNode or None if not found
        """
        # TODO 1: Check if node exists in self.nodes
        # TODO 2: Remove all virtual nodes for this physical node
        # TODO 3: Update self.sorted_positions list
        # TODO 4: Remove from self.nodes dict
        # TODO 5: Return the removed node
        pass

    def get_node(self, key: str) -> Optional[EdgeNode]:
        """
        Get the node responsible for a given cache key.

        Args:
            key: Cache key (URL + vary headers)

        Returns:
            EdgeNode responsible for this key, or None if ring is empty
        """
        # TODO 1: If ring is empty, return None
        # TODO 2: Hash the key to get position
        # TODO 3: Use bisect to find first position >= hashed position
        # TODO 4: If at end of list, wrap around to first position
        # TODO 5: Get virtual node at that position
        # TODO 6: Return the physical node
        pass

    def get_replica_nodes(self, key: str, count: int = 3) -> List[EdgeNode]:
        """
        Get the primary node plus replica nodes for a key.

        Used for cache replication to handle node failures.

        Args:
            key: Cache key
```

```
            count: Total number of nodes to return (including primary)

        Returns:
            List of EdgeNodes, starting with primary
        """
        # TODO 1: Get primary node with get_node(key)
        # TODO 2: If ring is empty, return empty list
        # TODO 3: Find position of primary node's virtual node
        # TODO 4: Get next N-1 distinct physical nodes clockwise
        # TODO 5: Return list of nodes
        pass

    def redistribute_keys(self, old_node: EdgeNode, new_node: EdgeNode) -> List[Tuple[str, EdgeNode]]:
        """
        Calculate which keys need to move when replacing a node.

        Args:
            old_node: Node being removed
            new_node: Node being added

        Returns:
            List of (key, new_node) pairs for keys that should move
        """
        # TODO 1: Get all virtual node positions for old_node
        # TODO 2: For each virtual node position:
        #   - Find keys that hash to positions between this node and previous
        #   - These keys will now map to new_node
        # TODO 3: Return list of affected keys and their new node
        # Note: In practice, we'd need to know all existing keys to do this
        pass
```

Simple GeoIP Lookup Utility:

PYTHON

```python
# routing/geo.py

import maxminddb

from dataclasses import dataclass

from typing import Optional, Tuple


@dataclass

class GeoLocation:

    """Geographic location information."""

    country_code: Optional[str]

    country_name: Optional[str]

    region_code: Optional[str]

    region_name: Optional[str]

    city: Optional[str]

    latitude: Optional[float]

    longitude: Optional[float]

    metro_code: Optional[int]  # US metro code

    timezone: Optional[str]


class GeoIPLookup:

    """GeoIP lookup using MaxMind database."""



    def __init__(self, database_path: str):

        """

        Initialize GeoIP lookup.



        Args:

            database_path: Path to MaxMind GeoLite2 database file

        """

        # TODO 1: Open MaxMind database file

        # TODO 2: Verify database is readable and valid

        # TODO 3: Store reader instance

        pass



    def lookup(self, ip_address: str) -> Optional[GeoLocation]:

        """

        Look up geographic location for an IP address.



        Args:

            ip_address: IP address string (IPv4 or IPv6)



        Returns:
```

```
            GeoLocation or None if not found
        """
        # TODO 1: Validate IP address format
        # TODO 2: Query MaxMind database
        # TODO 3: Parse response into GeoLocation dataclass
        # TODO 4: Handle errors (invalid IP, not in database)
        pass

    def distance_between(self, loc1: GeoLocation, loc2: GeoLocation) -> float:
        """
        Calculate great-circle distance between two locations.

        Uses Haversine formula.

        Args:
            loc1: First location
            loc2: Second location

        Returns:
            Distance in kilometers
        """
        # TODO 1: Check if both locations have latitude/longitude
        # TODO 2: Convert degrees to radians
        # TODO 3: Apply Haversine formula
        # TODO 4: Return distance in kilometers
        pass

    def find_nearest_node(self, client_ip: str, nodes: List[EdgeNode]) -> Optional[EdgeNode]:
        """
        Find nearest edge node for a client IP.

        Args:
            client_ip: Client IP address
            nodes: List of available edge nodes

        Returns:
            Nearest EdgeNode or None if can't determine
        """
        # TODO 1: Look up client location from IP
        # TODO 2: For each node with known location:
```

```
    #   - Calculate distance to client
    # TODO 3: Return node with minimum distance
    # TODO 4: Handle edge cases (no location data, etc.)
    pass
```

E. Python-Specific Hints

1. **MaxMind Database**: Use `maxminddb` library: `pip install maxminddb`. The free GeoLite2 database is available from MaxMind (requires account registration).
2. **DNS Server**: For a simple DNS server, use the `dnslib` library: `pip install dnslib`. It provides DNS protocol handling without needing bind/named.
3. **Consistent Hashing Performance**: Use `bisect` module for O(log n) lookups in the sorted positions list. For production, consider a balanced tree structure.
4. **Health Checks**: Use `asyncio` for non-blocking health checks. The `aiohttp` library is good for HTTP health checks to other nodes.
5. **Node Discovery**: For a simple implementation, use HTTP registration: nodes POST to control plane on startup. For advanced, use `etcd3` or `kazoo` (ZooKeeper) libraries.
6. **Monitoring Metrics**: Use `psutil` library to gather system metrics (CPU, memory, disk) for health reporting.

F. Milestone Checkpoint

To verify Milestone 4 implementation:

1. **Start multiple edge nodes**:

```bash
# Terminal 1 (US East)
python -m blue_origin.edge.server --config config/edge_us_east.yaml

# Terminal 2 (US West)
python -m blue_origin.edge.server --config config/edge_us_west.yaml

# Terminal 3 (Europe)
python -m blue_origin.edge.server --config config/edge_eu.yaml
```

2. **Start control plane and DNS server**:

```bash
python -m blue_origin.routing.dns_server --config config/routing.yaml
python -m blue_origin.control.plane --config config/control.yaml
```

3. **Test DNS-based geo-routing**:

```bash
# Simulate queries from different locations
dig @localhost -p 5353 mycdn.example.com A +subnet=8.8.8.0/24  # US query
dig @localhost -p 5353 mycdn.example.com A +subnet=194.158.0.0/16  # EU query
```

Expected: Different IPs returned based on source subnet.

4. **Test health checking**:

```bash
# Check health endpoint
curl http://localhost:8080/health  # Should return JSON with status

# Stop one edge node, verify DNS stops returning it within 15 seconds
# (kill US West node, then query again)
```

5. **Test consistent hashing**:

```python
# Python test script

from routing.hash_ring import ConsistentHashRing, EdgeNode



ring = ConsistentHashRing()

ring.add_node(EdgeNode("us-east-1", "us-east", "10.0.1.1:8080"))

ring.add_node(EdgeNode("us-west-2", "us-west", "10.0.2.1:8080"))



# Same URL should always map to same node

node1 = ring.get_node("/images/cat.jpg")

node2 = ring.get_node("/images/cat.jpg")

assert node1.node_id == node2.node_id



# Different URLs should distribute across nodes

urls = [f"/page/{i}" for i in range(1000)]

distribution = {}

for url in urls:

    node = ring.get_node(url)

    distribution[node.node_id] = distribution.get(node.node_id, 0) + 1

print(f"Distribution: {distribution}")  # Should be roughly 50/50
```

6. **Test failover**:

```bash
# Simulate node failure

pkill -f "edge_us_west.yaml"



# Wait 15 seconds for health checks to detect

# Then query DNS - should not return west IP

# HTTP requests to west should get connection refused

# Requests should redirect to east or europe
```

Expected Behaviors to Verify:

- DNS returns different IPs for clients from different continents
- Health endpoint returns 200 OK with metrics when node is healthy
- When a node is stopped, DNS stops returning its IP within 30 seconds
- Cache keys consistently map to same node (until ring changes)
- Adding/removing nodes affects only ~1/N of cache mappings
- HTTP redirects work when cache needs to be fetched from different edge

Signs of Problems:

- DNS always returns same IP regardless of source: GeoIP database not loaded
- Health checks fail: firewall blocking health endpoint
- Inconsistent node mapping: hash function or ring implementation bug
- Slow failover (>30 seconds): DNS TTL too high or health check interval too long

Interactions and Data Flow

> **Milestone(s):** Milestone 1 (Edge Cache Implementation), Milestone 2 (Cache Invalidation), Milestone 3 (Origin Shield & Request Collapsing), Milestone 4 (Edge Node Distribution & Routing)

Understanding the static architecture of components is only half the picture. This section animates the system by tracing the journey of real user requests and administrative commands through the complete CDN pipeline. We examine three critical scenarios: the optimal path where content is served directly from the edge, the more complex path involving a cache miss and origin shield, and the control-plane operation of globally invalidating cached content. These traces reveal how the components defined in previous sections collaborate to deliver a seamless, high-performance caching service while maintaining consistency and resilience.

Data Flow: Cache Hit at Edge

Mental Model: The Express Checkout Lane Imagine a supermarket with a dedicated express lane for customers purchasing common, pre-packaged items. When you approach with a loaf of bread, the cashier immediately recognizes the item (cache key), retrieves it from a nearby shelf (cache storage), scans it (applies headers), and completes the transaction in seconds. No need to visit the store's warehouse (origin). This is a **cache hit** – the fastest possible path through the CDN, where a valid, fresh copy of the requested resource resides at the edge node closest to the user.

A cache hit represents the CDN's primary value proposition: dramatically reduced latency. The following sequence describes each step, from the user's initial request to the cached response being served. Refer to the flowchart `![Flowchart: Edge Node Request Handling](/api/project/cdn-implementation/architecture-doc/asset?path=diagrams%2Fflowchart-request-handling.svg)` for a visual guide to the decision logic within the `EdgeRequestHandler`.

1. **Client Request Initiation:** An end-user's browser or application sends an HTTP GET request for a resource (e.g., `https://cdn.example.com/images/photo.jpg`). This request may include conditional headers like `If-None-Match` (with an ETag) or `If-Modified-Since` (with a timestamp).
2. **Geo-Routing & Edge Selection:** The client's DNS resolver, or a global load balancer using Anycast, directs the request to the nearest healthy edge node based on the client's IP address and geographic proximity. The selected edge node's `EdgeRequestHandler` receives the request.
3. **Request Parsing & Cache Key Construction:** The handler parses the request URL and headers. It constructs a **cache key** by combining the normalized request URL (including query string) and the values of any headers listed in a potential `Vary` header from a previous cached response (e.g., `Accept-Encoding`). If the request itself contains a `Vary: *` header, the handler immediately classifies this as uncacheable and proceeds to a fetch from upstream.
4. **Cache Lookup:** The handler queries the `CacheStorage` (e.g., an in-memory LRU store) with the generated cache key. In this scenario, the lookup succeeds, returning a `CacheEntry` object.
5. **Freshness Validation:** The handler checks if the cached entry is fresh by calling `CacheEntry.is_fresh(current_time)`. This evaluates the cached `CacheDirectives` (primarily `s-maxage` or `max-age`) against the current time and the entry's `expires_at` timestamp. If the entry is fresh, the system proceeds to step 7.
6. **Conditional Revalidation (If Stale):** If the entry is stale but `stale-while-revalidate` is present and its period has not expired, the handler may immediately serve the stale data (proceeding to step 7) while simultaneously spawning a background task via `_revalidate_in_background` to fetch an updated version from the origin. Alternatively, if the client provided conditional headers (`If-None-Match`, `If-Modified-Since`), the handler forwards these to the origin in a conditional request. A `304 Not Modified` response from the origin would refresh the entry's TTL without transferring the body, still resulting in a cache hit from the user's perspective.
7. **Ban Rule Check:** Before serving the cached content, the handler calls `InvalidationHandler.check_bans(url)` to verify the request URL doesn't match any active `BanRule`. In a hit scenario with no ban, this returns `None`.
8. **Response Preparation:** The handler prepares the HTTP response. It uses the status code, headers, and body stored in the `CacheEntry`. Critical caching headers like `Age` (calculated as `current_time - fetched_at`) and `Cache-Control` are updated to reflect the current state. If the client supports compression and the cache holds a compressed variant (per the `Vary: Accept-Encoding` dimension), the appropriate `Content-Encoding` header is set.
9. **Analytics Update:** The `EdgeMetrics` are updated: `total_hits` is incremented, `total_bandwidth_served` is increased by the size of the response body, and counters for the specific status code and content type are updated.
10. **Response Delivery:** The fully formed HTTP response is transmitted back to the client. The transaction is complete, with the origin server entirely uninvolved.

Key Data Structures Involved:

Structure	Role in Cache Hit
`CacheEntry`	The stored response object containing body, headers, and metadata.
`CacheDirectives`	Parsed directives that determine the entry's freshness and revalidation behavior.
`EdgeMetrics`	Accumulates performance data for analytics.

Data Flow: Cache Miss with Origin Shield

Mental Model: The Concert Vestibule Returning to the concert analogy from the Origin Shield component section, consider a group of fans arriving at the venue for a popular band. They all want entry to the same section (the resource). Instead of each fan individually bothering the busy main ticket scanner (origin), a staff member in the vestibule (shield) takes one representative's request, goes to the scanner to get the ticket validated, and returns with the approval. The other waiting fans receive copies of the same approval. The **shield** orchestrates this **request collapsing**, preventing a stampede.

This data flow is the most complex, involving multiple tiers and sophisticated coordination to protect the origin. The sequence diagram `![Sequence: Cache Miss with Origin Shielding](/api/project/cdn-implementation/architecture-doc/asset?path=diagrams%2Fseq-cache-miss-shield.svg)` illustrates the full interaction.

1. **Initial Steps (1-4):** The flow begins identically to a cache hit: client request, geo-routing, and cache key generation at the edge node. However, the cache lookup (step 4) fails, resulting in a **miss**. The `EdgeMetrics.total_misses` is incremented.
2. **Edge to Shield Request:** The `EdgeRequestHandler` calls `_fetch_from_upstream`, which forwards the original request (potentially with added CDN-specific headers like `X-Forwarded-For`) to the configured **origin shield** URL, not directly to the origin.
3. **Shield Receives Request:** The `ShieldRequestHandler` at the shield node receives the request. It first performs its own **cache lookup** using the same cache key logic. In this miss scenario, the shield also does not have a fresh entry.
4. **Request Collapsing Logic:** The shield uses its `RequestCoalescingMap` to prevent a **thundering herd**. It calls `RequestCoalescingMap.get_or_create_future(cache_key)`.
   - If this is the **first concurrent request** for this key, the method creates a new `asyncio.Future`, registers it in the map, and allows this request to proceed to the origin.
   - If other **identical requests are already in-flight**, the method returns the existing `Future`. These duplicate requests will `await` the result of that single Future, effectively "collapsing" into one origin fetch.
5. **Origin Fetch with Concurrency Limit:** The shield thread/proces that won the right to fetch from the origin first acquires a permit from the `_origin_semaphore`, which enforces a maximum concurrency limit to the origin. It then forwards the request to the **origin server**.
6. **Origin Response:** The origin server processes the request and returns a standard HTTP response (e.g., `200 OK` with the image data). It should include proper caching headers (`Cache-Control`, `ETag`, `Last-Modified`).
7. **Shield Cache Storage & Future Fulfillment:** The shield receives the origin response. It validates if the response is cacheable via `is_response_cacheable`. If cacheable, it creates a `CacheEntry` via `CacheEntry.from_upstream_response` and stores it in its own `CacheStorage`. The shield then calls `RequestCoalescingMap.fulfill_future(cache_key, result)` with the response. This action awakens all collapsed requests waiting on that `Future`, providing each with the same response data.
8. **Negative Caching:** If the origin returns an error (e.g., `404 Not Found` or `5xx`), the shield may still **cache this negative response** for a very short TTL (e.g., 10 seconds) to prevent overwhelming the origin with repeated requests for the same non-existent or failing resource. This is a critical protection mechanism.
9. **Response to Edge:** The shield sends the origin's response (or the cached negative response) back to the requesting edge node.
10. **Edge Cache Storage:** The edge node receives the response from the shield. It performs its own cacheability check, creates its own `CacheEntry`, stores it locally, and updates the `SurrogateKeyIndex` if the response includes a `Surrogate-Key` header.
11. **Analytics Update:** Both shield and edge update their metrics (`ShieldMetrics.collapsed_requests`, `EdgeMetrics.total_bandwidth_upstream`).
12. **Final Delivery:** Finally, the edge node sends the HTTP response back to the original client. The client receives the data, unaware of the multi-tiered fetching and collapsing that occurred.

Architecture Decision Record: Request Collapsing Timeout Strategy

Decision: Use a Short, Configurable Timeout for Collapsed Requests

- **Context:** When requests are collapsed at the shield, duplicate clients wait for a single origin fetch. If that fetch hangs or is very slow, all waiting clients could experience unacceptable latency or timeouts.
- **Options Considered:**
  1. **Infinite Wait:** Duplicate requests wait indefinitely for the lead request's future. Simple but risks massive client timeouts and resource exhaustion during origin slowness.
  2. **Client-Propagated Timeout:** Use the shortest remaining client timeout among the collapsed requests. Complex to coordinate and requires intercepting/client timeout awareness.
  3. **Fixed, Configurable Shield Timeout:** Set a shield-specific timeout (e.g., 3-5 seconds) shorter than typical client HTTP timeouts. If the lead request exceeds this, its future fails, releasing waiting requests to potentially retry or bypass collapsing.
- **Decision:** Implement a fixed, configurable timeout in the `RequestCoalescingMap` (Option 3).
- **Rationale:** This provides a straightforward safety valve. It ensures the shield doesn't become a bottleneck if the origin is severely impaired. The timeout should be configured to be shorter than the edge node's upstream timeout and typical client timeouts, ensuring the edge can fall back to a direct fetch or error before the client disconnects.
- **Consequences:** Requires logic to clean up timed-out futures and potentially allows a subsequent request to become the new "lead" for fetching, preventing a complete deadlock. Adds a configuration parameter to tune based on origin performance SLAs.

Option	Pros	Cons	Chosen?
Infinite Wait	Simple implementation	Causes cascading client failures on origin slowness	No
Client-Propagated Timeout	Fair, respects each client's tolerance	Extremely complex; requires parsing timeouts from unknown client implementations	No
Fixed Shield Timeout	Simple, predictable, provides circuit-breaker	May break slow-but-valid origin responses; requires tuning	**Yes**

Control Flow: Global Cache Invalidation

Mental Model: The Building Eviction Notice System Imagine a property manager needing to remove specific items from all branch offices. For a single filing cabinet (URL purge), they call each branch to have it removed immediately. For a policy change affecting all documents of a certain category (tag purge), they send a broadcast memo listing the category code. For a rule that all documents matching a pattern (e.g., "draft-*") should be discarded upon next access (ban), they update a central rulebook distributed to all branches. The **control plane** is the manager's office, coordinating these distributed operations.

Invalidation is a control-plane operation that ensures consistency between the origin's truth and the cached copies. Its propagation must be reliable and efficient. The sequence diagram `![Sequence: Purge Propagation via Control Plane](/api/project/cdn-implementation/architecture-doc/asset?path=diagrams%2Fseq-invalidation-propagation.svg)` depicts this process.

1. **Administrative Command:** An administrator or an automated system (e.g., triggered by a content management system update) issues an invalidation command. This is typically done via a dedicated API endpoint (e.g., `PURGE /purge` on the control plane) or a management console. The command specifies the type (`purge`, `soft-purge`, `ban`, `tag`) and target (URL, pattern, or surrogate key).
2. **Control Plane Reception & Validation:** The control plane's API server receives the command, authenticates/authorizes the request, and validates its parameters. It constructs a standardized `InvalidationMessage` containing a unique `message_id`, `command`, `target`, `timestamp`, and other metadata.
3. **Message Broadcast:** The control plane publishes the `InvalidationMessage` to a specific **pub/sub channel** (e.g., `invalidation:global`) using the `InMemoryPubSubBroker.publish` method. All edge nodes that have previously subscribed to this channel via `InMemoryPubSubBroker.subscribe` will receive the message. This decouples the sender from the receivers and allows for scalable fan-out.
4. **Edge Node Reception:** Each edge node's `InvalidationHandler` has a registered callback that is invoked with the `InvalidationMessage`. The handler first may deduplicate messages using the `message_id` to avoid processing the same command multiple times (e.g., in case of network retries).
5. **Local Invalidation Execution:** The handler executes the command locally on its node's cache:
   - **For a URL Purge:** It generates the cache key for the given URL (considering `Vary` headers) and calls `_hard_purge_key` or `_soft_purge_key`.
   - **For a Tag Purge:** It calls `_purge_by_tag(tag, is_soft)`, which uses the local `SurrogateKeyIndex.get_keys_for_tag` to find all affected cache keys and purges them.
   - **For a Ban Rule:** It calls `add_ban_rule(pattern, is_soft, ttl_seconds)` to add the rule to the local list of `BanRule`s. Future requests matching the pattern will be invalidated (lazily, upon access).
6. **Acknowledgment (Optional):** In a more advanced implementation, edge nodes may send an acknowledgment message back to the control plane, allowing it to track propagation completeness. For our foundational design, we assume **best-effort, eventually consistent** propagation.
7. **Cleanup:** For ban rules, a background garbage collection task (`_gc_ban_rules`) periodically runs on each edge node to remove expired rules, preventing memory leaks.

Common Pitfalls in Invalidation Propagation:

⚠️ Pitfall: Ignoring Propagation Delay

- **Description:** Assuming that once a purge API call returns, the content is immediately invalidated worldwide.
- **Why it's Wrong:** Network latency, node failures, and message queue delays mean there is a window (seconds to minutes) where some edge nodes may still serve stale content. This violates consistency expectations.
- **Fix:** Design the system with **eventual consistency** in mind. Document this behavior. For stronger guarantees, implement synchronous purges to critical nodes or use versioned URLs (e.g., `/image.jpg?v=2`) to bypass caching entirely.

⚠️ Pitfall: Inefficient Tag-Based Purge on Large Indexes

- **Description:** Iterating over all cache entries to find those matching a tag when the `SurrogateKeyIndex` is large can block the request thread and cause latency spikes.
- **Why it's Wrong:** A tag like "homepage" might be associated with thousands of entries (CSS, JS, images). Linearly scanning a list or performing many individual cache deletions is slow.
- **Fix:** Use the reverse index in `SurrogateKeyIndex` (`tag_to_keys`) for O(1) lookups of affected keys. Perform the actual cache evictions in a background thread to avoid blocking the purge API response.

⚠️ Pitfall: Ban Rule Explosion

- **Description:** Continuously adding new ban rules (e.g., for every user-generated content update) without a TTL or cleanup mechanism.
- **Why it's Wrong:** The list of `BanRule` objects grows indefinitely, consuming memory and slowing down the `check_bans(url)` function which must evaluate every rule against every requested URL.
- **Fix:** Always attach a sensible TTL to programmatically added ban rules. Implement the `_gc_ban_rules` background task to regularly purge expired rules. For high-volume patterns, consider using a more scalable data structure like a Bloom filter for certain prefix-based bans (with the understanding of its probabilistic nature).

Implementation Guidance

This guidance provides the foundational code to connect the components and realize the data flows described above. Focus is on the glue logic and critical coordination structures.

A. Technology Recommendations Table

Component	Simple Option (for Learning)	Advanced Option (for Production-readiness)
Inter-Component Messaging	In-memory Pub/Sub within a single process (for simulation)	Redis Pub/Sub or Apache Kafka for distributed, persistent messaging
Request Collapsing	`asyncio.Future` and `asyncio.Lock` in Python	Dedicated library with support for timeouts, circuit breaking, and metrics (e.g., `aiocache` patterns)
Health Checking & Discovery	Periodic HTTP `GET /health` polls from control plane	Gossip protocol (SWIM) for decentralized failure detection and membership
Metrics Collection	In-memory counters flushed periodically to logs	OpenTelemetry SDK exporting to Prometheus or Grafana

B. Recommended File/Module Structure Add the following files to manage the flow and interactions:

```
blue_origin_cdn/
├── internal/
│   ├── flow/
│   │   ├── __init__.py
│   │   ├── coordinator.py          # Control plane logic for message broadcasting
│   │   └── sequences.py           # Integration tests for the data flows
│   ├── messaging/
│   │   ├── __init__.py
│   │   ├── pubsub.py              # InMemoryPubSubBroker implementation
│   │   └── message.py             # InvalidationMessage and other DTOs
│   └── utils/
│       └── timeouts.py            # Utility for managing timeouts in collapsing
└── simulations/
    ├── multi_edge_hit.py          # Script to simulate cache hit flow
    ├── miss_with_shield.py        # Script to simulate miss/collapsing flow
    └── global_purge.py            # Script to simulate invalidation propagation
```

C. Infrastructure Starter Code

1. In-Memory Pub/Sub Broker (`internal/messaging/pubsub.py`): This is a complete, thread-safe implementation for simulation and testing. In a real distributed CDN, this would be replaced with Redis or Kafka.

PYTHON

```python
import asyncio

from typing import Callable, Set, Dict, Any

from dataclasses import dataclass

import time

import json


@dataclass

class InvalidationMessage:

    message_id: str

    command: str  # "purge", "soft_purge", "ban", "tag_purge"

    target: str   # URL, pattern, or tag

    soft: bool

    timestamp: float

    parameters: Dict[str, Any]


class InMemoryPubSubBroker:

    """Simple in-memory publish-subscribe broker for simulation."""



    def __init__(self):

        self._subscribers: Dict[str, Set[Callable[[InvalidationMessage], None]]] = {}

        self._lock = asyncio.Lock()



    async def subscribe(self, channel: str, callback: Callable[[InvalidationMessage], None]):

        """Subscribe a callback to a channel."""

        async with self._lock:

            if channel not in self._subscribers:

                self._subscribers[channel] = set()

            self._subscribers[channel].add(callback)



    async def unsubscribe(self, channel: str, callback: Callable[[InvalidationMessage], None]):

        """Unsubscribe a callback from a channel."""

        async with self._lock:

            if channel in self._subscribers:

                self._subscribers[channel].discard(callback)

                if not self._subscribers[channel]:

                    del self._subscribers[channel]



    async def publish(self, channel: str, message: InvalidationMessage):

        """Publish a message to all subscribers of a channel."""

        async with self._lock:

            subscribers = self._subscribers.get(channel, set()).copy()
```

```
        # Call each subscriber in the background (fire-and-forget)
        for callback in subscribers:
            # In a real system, you might want to handle exceptions
            asyncio.create_task(self._safe_callback(callback, message))

    async def _safe_callback(self, callback, message):
        """Wrapper to catch and log exceptions in subscriber callbacks."""
        try:
            await callback(message)
        except Exception as e:
            print(f"Error in pub/sub callback: {e}")
```

2. Request Coalescing Map with Timeout (`internal/flow/coordinator.py`): This is a critical piece for the origin shield. It manages the futures for collapsing.

PYTHON

```python
import asyncio

from typing import Dict, Optional

import time


class RequestCoalescingMap:

    """

    Tracks in-flight requests by cache key to enable request collapsing.

    """



    def __init__(self, default_timeout: float = 3.0, cleanup_interval: float = 30.0):

        self._futures: Dict[str, asyncio.Future] = {}

        self._timeouts: Dict[str, float] = {}  # key -> creation time

        self._default_timeout = default_timeout

        self._lock = asyncio.Lock()

        self._gc_task: Optional[asyncio.Task] = None

        self._start_gc_loop(cleanup_interval)



    def _start_gc_loop(self, interval: float):

        """Start background garbage collection loop."""

        async def gc_loop():

            while True:

                await asyncio.sleep(interval)

                await self._gc_loop(interval)



        self._gc_task = asyncio.create_task(gc_loop())



    async def get_or_create_future(self, cache_key: str) -> asyncio.Future:

        """

        Get existing Future for key or create a new one.

        Returns a Future that will resolve to the (status, headers, body) tuple.

        """

        async with self._lock:

            if cache_key in self._futures:

                return self._futures[cache_key]



            # Create new future

            future = asyncio.Future()

            self._futures[cache_key] = future

            self._timeouts[cache_key] = time.time()

            return future
```

```
async def fulfill_future(self, cache_key: str, result: tuple):
    """Set result for Future associated with key and clean up."""
    async with self._lock:
        future = self._futures.pop(cache_key, None)
        self._timeouts.pop(cache_key, None)

    if future and not future.done():
        future.set_result(result)

async def fail_future(self, cache_key: str, exception: Exception):
    """Set exception for Future associated with key and clean up."""
    async with self._lock:
        future = self._futures.pop(cache_key, None)
        self._timeouts.pop(cache_key, None)

    if future and not future.done():
        future.set_exception(exception)

async def _gc_loop(self, interval: float):
    """Background cleanup of timed-out entries."""
    now = time.time()
    to_remove = []

    async with self._lock:
        for key, created_at in self._timeouts.items():
            if now - created_at > self._default_timeout:
                to_remove.append(key)

        for key in to_remove:
            future = self._futures.pop(key, None)
            self._timeouts.pop(key, None)
            if future and not future.done():
                future.set_exception(asyncio.TimeoutError(f"Request coalescing timeout for {key}"))

async def stop(self):
    """Stop the garbage collection loop."""
    if self._gc_task:
        self._gc_task.cancel()
        try:
            await self._gc_task
```

```
        except asyncio.CancelledError:
            pass
```

D. Core Logic Skeleton Code

1. Shield Request Handler with Collapsing (`internal/shield/handler.py`):

```python
import asyncio

from typing import Tuple, Dict

from ..cache.storage import CacheStorage

from ..flow.coordinator import RequestCoalescingMap


class ShieldRequestHandler:

    """

    Handles requests at the origin shield layer with request collapsing.

    """



    def __init__(self, cache: CacheStorage, origin_upstream: str,

                 coalescing_map: RequestCoalescingMap, max_concurrent_requests: int = 10):

        self.cache = cache

        self.origin_upstream = origin_upstream

        self.coalescing_map = coalescing_map

        self._origin_semaphore = asyncio.Semaphore(max_concurrent_requests)



    async def handle_request(self, request_headers: Dict[str, str],

                             request_body: bytes) -> Tuple[int, Dict[str, str], bytes]:

        """

        Main request handling algorithm for the shield.

        Steps:

        1. Generate cache key from URL and Vary headers.

        2. Check local cache; if fresh, serve and return.

        3. If stale but revalidatable, serve stale and start background revalidation.

        4. On miss, use request collapsing map to deduplicate concurrent requests.

        5. Acquire semaphore to limit origin concurrency.

        6. Fetch from origin.

        7. Cache the response (including negative caching for errors).

        8. Fulfill the future in the coalescing map.

        9. Return response.

        """

        # TODO 1: Parse request method, URL, and headers from request_headers

        # TODO 2: Generate a cache key using URL and Vary header dimensions

        # TODO 3: Check local cache storage for the key

        # TODO 4: If cache hit and fresh, return cached response immediately

        # TODO 5: If cache hit but stale, check if stale-while-revalidate applies

        #         - If yes, serve stale and call _revalidate_in_background

        # TODO 6: On cache miss, call self.coalescing_map.get_or_create_future(key)

        # TODO 7: If the returned future is already done (another request filled it), return its result

        # TODO 8: If this is the first request (future not done), proceed to fetch from origin:
```

```
        #         a. Acquire self._origin_semaphore
        #         b. Make HTTP request to self.origin_upstream
        #         c. Parse response status, headers, body
        #         d. Determine if response is cacheable
        #         e. If cacheable, store in self.cache
        #         f. Call self.coalescing_map.fulfill_future(key, (status, headers, body))
        #         g. Return the response
        # TODO 9: Implement proper exception handling:
        #         - On origin timeout/error, call self.coalescing_map.fail_future
        #         - Consider negative caching for 404/5xx errors with short TTL
        pass


    async def _revalidate_in_background(self, cache_key: str, stale_entry, request_headers: Dict[str, str]):
        """
        Asynchronously revalidate a stale cache entry.
        Should update the cache if the origin returns 200, or delete it if 304.
        """
        # TODO 1: Make a conditional request to origin with If-None-Match/If-Modified-Since
        # TODO 2: If origin returns 200 OK, replace stale_entry in cache with new response
        # TODO 3: If origin returns 304 Not Modified, update stale_entry's expires_at/metadata
        # TODO 4: Log errors but don't propagate (this is background)
        pass
```

2. Control Plane Invalidation Endpoint (`internal/control/api.py`):

PYTHON

```python
from fastapi import FastAPI, HTTPException, BackgroundTasks

from ..messaging.pubsub import InMemoryPubSubBroker, InvalidationMessage

import uuid

import time


app = FastAPI()

broker = InMemoryPubSubBroker()


@app.post("/invalidate/purge")

async def purge_url(url: str, soft: bool = False, background_tasks: BackgroundTasks = None):

    """

    Admin API to purge a specific URL from all edge caches.

    """

    # TODO 1: Validate admin authentication/authorization (basic implementation could use API key)

    # TODO 2: Create an InvalidationMessage with command="purge", target=url, soft=soft

    message = InvalidationMessage(

        message_id=str(uuid.uuid4()),

        command="soft_purge" if soft else "purge",

        target=url,

        soft=soft,

        timestamp=time.time(),

        parameters={"reason": "manual_api_call"}

    )



    # TODO 3: Publish the message to the global invalidation channel

    # Use background task to avoid blocking the API response

    background_tasks.add_task(broker.publish, "invalidation:global", message)



    return {"status": "accepted", "message_id": message.message_id}


@app.post("/invalidate/tag")

async def purge_tag(tag: str, soft: bool = False, background_tasks: BackgroundTasks = None):

    """

    Admin API to purge all resources with a given surrogate key tag.

    """

    # TODO 1: Validate input (tag format)

    # TODO 2: Create InvalidationMessage with command="tag_purge"

    # TODO 3: Publish to broker

    pass


@app.post("/invalidate/ban")

async def add_ban_rule(pattern: str, ttl_seconds: int = 3600,
```

```
                    soft: bool = False, background_tasks: BackgroundTasks = None):
    """
    Admin API to add a new ban rule.
    """
    # TODO 1: Validate pattern (e.g., check for reasonable length, prevent injection)
    # TODO 2: Create InvalidationMessage with command="ban"
    # TODO 3: Include ttl_seconds in parameters
    # TODO 4: Publish to broker
    pass
```

E. Language-Specific Hints (Python)

- **Use `asyncio` for Concurrency:** The request collapsing and background revalidation are natural fits for async/await. Use `asyncio.create_task()` for fire-and-forget background operations.
- **Type Hints:** Use Python type hints extensively for the data structures defined in the naming conventions. This will catch many errors early and serve as documentation.
- **Testing Async Code:** Use `pytest-asyncio` for testing async components. Mock the `CacheStorage` and origin HTTP calls to simulate different flow scenarios.
- **Configuration Management:** Use Pydantic models for `EdgeConfig` to get validation and type-safe access to configuration values loaded from YAML or environment variables.

F. Milestone Checkpoint: Flow Integration After implementing the core components, test the integrated flows:

1. **Cache Hit Flow Test:**

```bash
# Start a mock origin server (serves static files)
python tests/mock_origin.py --port 8001

# Start an edge node pointing to the origin
python -m blue_origin_cdn.edge --config config/edge_local.yaml

# Make a request, then make it again (should hit)
curl -v http://localhost:8080/image.jpg
curl -v http://localhost:8080/image.jpg  # Check Age header increases
```

Expected: First request logs `[MISS]`, second logs `[HIT]`. The `Age` header in the second response should be >0.

2. **Cache Miss with Shield Test:**

```bash
# Start shield pointing to origin
python -m blue_origin_cdn.shield --config config/shield_local.yaml

# Start edge pointing to shield (not directly to origin)
# Update edge config upstream_url to shield address

# Use Apache Bench to simulate concurrent requests
ab -n 100 -c 10 http://localhost:8080/new_image.jpg
```

Expected: Shield logs should show `[COLLAPSED]` messages. Origin server logs should show only 1 request for `new_image.jpg`, not 100.

3. **Global Invalidation Test:**

```bash
# Start two edge nodes and one control plane
python -m blue_origin_cdn.control_plane &
python -m blue_origin_cdn.edge --config config/edge_node1.yaml &
python -m blue_origin_cdn.edge --config config/edge_node2.yaml &

# Populate cache on both edges
curl http://node1:8080/image.jpg
curl http://node2:8080/image.jpg

# Send purge command via control plane API
curl -X POST http://control-plane:9000/invalidate/purge?url=/image.jpg

# Request again - should miss on both nodes
curl -v http://node1:8080/image.jpg
curl -v http://node2:8080/image.jpg
```

Expected: After purge, both edges should log `[MISS]` and fetch fresh content from upstream. Check control plane logs for message broadcast.

Error Handling and Edge Cases

> **Milestone(s):** Milestone 1 (Edge Cache Implementation), Milestone 2 (Cache Invalidation), Milestone 3 (Origin Shield & Request Collapsing), Milestone 4 (Edge Node Distribution & Routing), Milestone 5 (CDN Analytics & Performance Optimization)

The true test of a CDN's architectural soundness isn't how it performs under ideal conditions, but how it degrades when components fail and how it handles the messy reality of HTTP traffic. This section systematically documents failure modes, recovery strategies, and edge case handling—transforming the CDN from a fragile prototype into a robust production-ready system.

Failure Modes & Recovery Strategies

Mental Model: The Redundant Bridge System

Imagine a network of bridges connecting islands. The primary bridge (origin) sometimes collapses, detour bridges (edge nodes) can become congested, and storms (network partitions) can isolate entire islands. A resilient transportation system has redundant routes, detour signs, and temporary shelters. Our CDN employs similar strategies: when the origin fails, we serve stale content from shelters (cache); when edges fail, we redirect traffic via detours (failover); when storms isolate nodes, they operate independently until reconnected (degraded mode).

Catalog of Failure Modes and Recovery Strategies

The CDN must handle failures at multiple levels while maintaining some level of service. The following table documents each failure mode, its detection mechanism, and the corresponding recovery strategy.

Failure Mode	Detection Method	Recovery Strategy	Impact on Clients	Implementation Notes
Origin Server Unavailable	HTTP request timeout (5s) or status code 5xx from origin	Serve stale content using `stale-if-error` directive; if no stale content available, return 502 Bad Gateway	Temporary degradation: clients may see older content or error pages	Monitor `origin_errors` in `ShieldMetrics`; implement exponential backoff for retries
Edge Node Failure	Health checks failing (TCP connect timeout, HTTP 5xx on `/health`)	Geo-routing redirects clients to next-nearest healthy edge node within failover timeout	Brief interruption (<5s) during failover, then normal service	Update consistent hash ring to redistribute keys from failed node
Network Partition (Edge ↔ Origin)	Repeated timeouts when contacting upstream	Edge nodes operate in "degraded mode": serve stale content exclusively, disable cache updates	Clients see potentially outdated content until partition heals	Implement circuit breaker pattern for upstream connections
Network Partition (Edge ↔ Control Plane)	Heartbeat timeout with control plane	Edge nodes continue serving traffic with local configuration; invalidation commands queue locally	Cache may become stale (eventual consistency issues)	Buffer invalidation messages in local queue for eventual sync
Cache Storage Exhaustion	`current_cache_size_bytes` approaches `cache_capacity_bytes`	LRU/LFU eviction removes least valuable entries; log warnings; optionally compress existing entries	Slight increase in cache misses for evicted content	Monitor `current_cache_size_bytes` in `EdgeMetrics`; implement compression for text responses
Disk I/O Errors (if persistent cache)	Failed read/write operations with errno	Fall back to in-memory cache only; log critical error; alert operator	Reduced cache capacity, increased origin load	Use `try/except` around all filesystem operations; implement graceful degradation
Memory Exhaustion	`MemoryError` exceptions or OOM killer	Reject new cache entries while serving existing ones; aggressively evict old entries	Increased cache misses, potential 503 errors for new content	Monitor memory usage via `HealthMetrics`; implement soft limits
DNS Resolution Failure	`socket.gaierror` when resolving upstream hostnames	Use cached DNS results with TTL; fall back to backup origin IPs if configured	Service interruption if no cached resolution available	Implement DNS caching with TTL validation; pre-resolve critical hostnames
SSL/TLS Handshake Failure	`ssl.SSLError` during upstream connection	Attempt plain HTTP if origin supports it (configurable); otherwise fail with 502	Potential security downgrade or service interruption	Implement TLS version fallback; maintain certificate cache
Clock Skew Between Nodes	Comparison of timestamps in invalidation messages	Use logical clocks (Lamport timestamps) for invalidation ordering; NTP synchronization required	Potential cache inconsistency if skew > TTL window	Include logical timestamp in `InvalidationMessage`; implement NTP client

Design Principle: *Graceful degradation beats catastrophic failure.* The system should never completely stop serving traffic unless absolutely necessary. Each failure mode has a defined fallback that maintains some level of service.

ADR: Stale Content Serving Strategy

Decision: Serve Stale Content with `stale-if-error` During Origin Failures

- **Context**: When the origin server becomes unavailable, the CDN faces a choice: return errors (502/504) or serve potentially outdated content from cache. Returning errors provides accuracy but poor user experience during origin outages.
- **Options Considered**:
  1. **Always return 502/504**: Strict correctness but poor availability during origin failures
  2. **Serve stale content indefinitely**: High availability but potentially serving very outdated content
  3. **Serve stale content with `stale-if-error` and time limits**: Balanced approach with configurable staleness windows
- **Decision**: Implement `stale-if-error` semantics per RFC 9111, with configurable maximum staleness (default: 1 hour).
- **Rationale**: The `stale-if-error` extension is standard HTTP semantics that clients understand. It provides availability during temporary origin issues while bounding how stale content can become. This aligns with CDN industry practices where serving slightly stale content is preferable to complete unavailability.
- **Consequences**: Clients may receive outdated content during origin failures, but the system remains available. Operators must monitor staleness duration and set appropriate limits based on content volatility.

Option	Pros	Cons	Chosen?
Always return 502/504	Strictly correct behavior, no stale content	Poor availability during origin outages	No
Serve stale indefinitely	Maximum availability	May serve very outdated content without bound	No
Serve with `stale-if-error` and limits	**Balances availability and freshness, standard-compliant**	**Adds complexity to cache freshness logic**	**Yes**

Implementation of Failure Recovery

The recovery strategies translate to concrete implementation patterns:

1. **Circuit Breaker for Upstream Connections**: Implement the circuit breaker pattern to prevent cascading failures when the origin is unhealthy. After a threshold of failures, the circuit opens and all requests immediately fail fast to stale cache, periodically allowing a test request to check if the origin has recovered.
2. **Health-Based Routing Fallback**: When the primary edge node is unhealthy, the geo-routing system should redirect to the next-nearest node. This requires maintaining a sorted list of backup nodes per region and implementing fast health check propagation.
3. **Degraded Mode Flag**: Each edge node should maintain a `degraded_mode` boolean that gets set when critical dependencies (origin, control plane) are unavailable. In degraded mode, the node:
    - Serves stale content regardless of freshness
    - Logs all cache misses (but doesn't attempt to fetch from origin)
    - Returns custom `X-CDN-Degraded: true` header
    - Periodically attempts to reconnect to dependencies
4. **Request Queue Management During Failover**: When an edge node fails, the consistent hashing ring redistributes its keys to other nodes. These nodes may experience a "cold start" problem with empty caches. Implement request queuing and rate limiting to prevent thundering herd on the origin during redistribution.

HTTP Edge Cases

Mental Model: The Quirky Mailroom Clerk

Imagine a mailroom clerk who must handle every possible mailing scenario: packages with contradictory instructions (Vary: *), massive crates that need partial delivery (range requests), and envelopes with unreadable postmarks (malformed headers). A good clerk has procedures for each edge case—knowing when to reject, when to improvise, and when to consult the rulebook. Our CDN edge cache acts as this clerk, implementing specific logic for HTTP's corner cases.

Handling Complex HTTP Caching Scenarios

HTTP Edge Case	Standard Behavior	CDN Implementation	Rationale
`Vary: *` Header	Per RFC 9111, means "never cache" because the response varies on all request headers	Check for `Vary: *` in response; if present, do not store in cache and bypass cache for future requests	Prevents cache key explosion and ensures correct content negotiation
`Cache-Control: no-store`	Response must not be stored in any cache	Respect directive: do not store response in cache; existing entries must be invalidated	Privacy/security requirement; override any other caching directives
`Cache-Control: private`	Response is intended for a single user and must not be stored in shared caches	CDN as shared cache must not store; edge caches skip storage, but shield may still cache if configured	Respects user privacy while allowing some optimization
Malformed `Cache-Control` Header	Invalid syntax (e.g., `max-age=not-a-number`)	Parse leniently: ignore malformed directives, log warning, use defaults (or `max-age=0` if uncertain)	Robustness over strictness; better to cache with short TTL than break
Conflicting Directives (e.g., `max-age=3600` and `no-cache`)	`no-cache` takes precedence for validation, but `max-age` still sets freshness lifetime	Implement precedence hierarchy: `no-store` > `no-cache` > `private` > `s-maxage` > `max-age` > default	Follows RFC 9111 Section 5.2.2 directive precedence
Multiple `Cache-Control` Headers	Headers should be concatenated with commas	Use `get_header_values()` to collect all values, then parse combined string	Handles HTTP/1.x and HTTP/2 differences in header serialization
`Expires` in the Past	Resource is already stale when received	Treat as `max-age=0`, requiring immediate revalidation	Prevents serving already-expired content
`ETag` with `W/` Prefix (Weak Validator)	Allows semantic equivalence (e.g., reformatted HTML)	Support weak validation: match weak ETags only with `If-None-Match`, not with `If-Match`	Enables more efficient caching when content changes insignificantly
`Vary` with Multiple Headers (e.g., `Vary: Accept-Encoding, User-Agent`)	Response varies on all listed headers	Include all specified header values in cache key; store separate entry for each combination	Correctly handles multi-dimensional content negotiation
`Accept-Encoding` with `identity` or `*`	Client accepts any encoding, including none	Normalize: treat `*` as `identity`; include normalized encoding in cache key	Prevents cache fragmentation from semantically equivalent headers
Range Requests (`Range: bytes=0-499`)	Request for partial content; requires `206 Partial Content` response	Check cache for full response; if present, extract range and return 206; otherwise fetch range or full from origin	Supports video streaming and large file downloads efficiently
`If-Range` Conditional Range Requests	Continue interrupted downloads if unchanged	If ETag/Last-Modified matches, serve requested range; otherwise serve full entity	Optimizes recovery from interrupted downloads
`POST` to Cacheable URL	POST generally invalidates cached GET to same URL	Invalidate cache entry for the URL on successful POST (status 2xx/3xx)	Maintains cache consistency when content is modified
`PUT` / `DELETE` to Cached URL	Modifies or removes resource	Invalidate cache entry for the URL	Ensures cache reflects resource state changes
Responses Exceeding Cache Size Limit	Too large to store given capacity constraints	Stream response to client without caching; log warning; increment `too_large_to_cache` metric	Prevents single large response from evicting many small ones
Chunked Transfer Encoding	Response body delivered in chunks	Reassemble complete body before caching; stream chunks to client immediately	Cache stores complete entities, not transfer encoding state
`304 Not Modified` Responses	Conditional request found resource unchanged	Update freshness of existing cache entry without replacing body; update headers from 304 response	Optimizes bandwidth by avoiding body retransmission
`206 Partial Content` from Origin	Origin supports range requests natively	Cache full response if possible; otherwise cache partial response marked as incomplete	Future range requests may need full response for other ranges

ADR: Handling Vary: * - To Cache or Not to Cache

> **Decision: Never Cache Responses with `Vary: *`**
>
> - **Context**: The `Vary: *` header indicates the response varies on all possible request headers. This creates a cache key explosion problem since every request is essentially unique.
> - **Options Considered**:
>   1. **Cache with URL-only key**: Ignore the `Vary: *` and cache based only on URL (incorrect, violates spec)
>   2. **Cache with all headers as key**: Include all request headers in cache key (theoretically correct but impractical)
>   3. **Do not cache at all**: Treat `Vary: *` as "never cache" as suggested by RFC 9111
> - **Decision**: Implement "never cache" behavior for `Vary: *` responses.
> - **Rationale**: RFC 9111 Section 4.1 suggests that `Vary: *` "always fails to match" subsequent requests, implying it should not be stored. Practical implementations (Varnish, NGINX) treat it as uncacheable. The alternative of including all headers would make the cache effectively useless due to key explosion.
> - **Consequences**: Some dynamic content that could theoretically be cached (if we ignored certain headers) won't be cached. This is an acceptable trade-off for correctness and simplicity.

Option	Pros	Cons	Chosen?
Cache with URL-only key	Would cache some dynamic content	Violates HTTP spec, serves wrong content to clients with different headers	No
Cache with all headers as key	Theoretically correct	Cache key explosion makes caching ineffective; high memory overhead	No
Do not cache at all	**Spec-compliant, simple to implement**	**Misses caching opportunities for some content**	**Yes**

Range Request Implementation Details

Range requests present particular challenges for caching. The CDN must handle:

1. **Cache Storage for Range Requests**: When a range request misses the cache but the origin returns a `206 Partial Content`, we have two options:
   - Store the partial response separately
   - Fetch the entire resource and cache it, then extract the range

   The recommended approach is to attempt to fetch the full resource (using `Range: bytes=0-` or no Range header) when cache storage is desirable. If the origin only supports range requests, we can cache partial responses with metadata indicating which ranges are available.
2. **Range Combining**: If multiple concurrent requests ask for different ranges of the same resource, the CDN should combine them into a single origin request for the full resource (or a superset range) to optimize bandwidth.
3. **Range Validation**: When serving a cached response to a range request, validate that the requested range is within the cached entity length. If not, either fetch the needed range from origin or return `416 Range Not Satisfiable`.

Algorithm for Handling Range Requests:

1. Parse `Range` header using `range-units` syntax (only `bytes` unit required)
2. Generate cache key including range information (or use full resource key)
3. Check cache for matching entry:
   - If full entity cached and fresh: extract range, return `206`
   - If partial entity cached containing requested range: serve from cache
   - Otherwise: proceed to origin
4. Forward to origin with appropriate `Range` header
5. On `206` response: cache full entity if possible, otherwise cache partial with range metadata
6. On `200` response (origin doesn't support ranges): cache full entity, extract requested range for response
7. Always set `Content-Range` and `Accept-Ranges: bytes` headers in response

Common Pitfalls in Error Handling

⚠️ Pitfall: Silent Degradation to Stale Content Indefinitely

- **Description**: When the origin fails, serving stale content via `stale-if-error` without time limits.
- **Why it's wrong**: Users may see hours-old content without knowing, violating content freshness expectations.
- **Fix**: Implement maximum staleness window (e.g., 1 hour) and add `X-CDN-Stale: true` header to responses.

⚠️ Pitfall: Cache Poisoning via Malformed Headers

- **Description**: Accepting malformed `Cache-Control` headers without validation leads to incorrect caching decisions.

- **Why it's wrong**: `max-age=999999999` (overflow) could cache content for decades; `no-cache=` (missing value) might be misinterpreted.
- **Fix**: Validate directive values: clamp `max-age` / `s-maxage` to reasonable maximum (e.g., 1 year), treat malformed directives as `max-age=0`.

⚠️ Pitfall: Range Request Cache Fragmentation

- **Description**: Storing each requested range as separate cache entry wastes memory and causes redundancy.
- **Why it's wrong**: Requests for `bytes=0-999` and `bytes=1000-1999` of same file create two cache entries storing overlapping data.
- **Fix**: Prefer caching full entities; for partial-only origins, implement range metadata tracking and combine overlapping ranges.

⚠️ Pitfall: Thundering Herd on Health Check Failover

- **Description**: When an edge node fails, all clients simultaneously reconnect to the next-nearest node, overwhelming it.
- **Why it's wrong**: The backup node experiences sudden load spike, potentially causing cascade failure.
- **Fix**: Implement staggered failover using DNS TTLs, client-side randomization, or anycast with BGP dampening.

⚠️ Pitfall: Ignoring Clock Skew in Expiration

- **Description**: Using local system time for TTL calculations without considering time synchronization.
- **Why it's wrong**: Nodes with skewed clocks expire content at different times, causing inconsistency.
- **Fix**: Use NTP-synchronized clocks; for critical timing, use relative durations from receipt time rather than absolute expiry.

Implementation Guidance

Technology Recommendations Table

Component	Simple Option	Advanced Option
Circuit Breaker	Manual failure counting with timeout	`aiocircuitbreaker` library with multiple strategies
Health Checking	Periodic HTTP GET to `/health` endpoint	Active health checks + passive monitoring + weighted scoring
Range Request Handling	Simple byte range slicing of cached bodies	Multipart range support with `boundary` parsing
Stale Content Serving	Basic `stale-if-error` with fixed window	Dynamic staleness based on content type and historical patterns
Error Monitoring	Logging to stdout/stderr	Structured logging with metrics export to Prometheus

Recommended File/Module Structure

```
blue_origin_cdn/
├── src/
│   ├── error_handling/
│   │   ├── __init__.py
│   │   ├── circuit_breaker.py          # Circuit breaker implementation
│   │   ├── degraded_mode.py            # Degraded mode flag and logic
│   │   ├── stale_content_handler.py    # stale-if-error logic
│   │   └── health_checker.py           # Health checking utilities
│   ├── http_edge_cases/
│   │   ├── __init__.py
│   │   ├── range_request.py           # Range request handler
│   │   ├── vary_header.py             # Vary header processor
│   │   ├── cache_control_parser.py    # Robust Cache-Control parser
│   │   └── etag_handler.py            # ETag weak/strong validation
│   └── failure_recovery/
│       ├── __init__.py
│       ├── failover_manager.py        # Geo-routing failover logic
│       ├── request_queuing.py         # Request queue for cold starts
│       └── partition_handler.py       # Network partition detection
└── tests/
    ├── test_error_handling.py
    ├── test_http_edge_cases.py
    └── test_failure_recovery.py
```

Infrastructure Starter Code: Circuit Breaker Implementation

PYTHON

```python
"""
Circuit breaker pattern implementation for upstream connections.
"""

import time
import asyncio
from enum import Enum
from typing import Optional, Callable, Any
from dataclasses import dataclass


class CircuitState(Enum):
    CLOSED = "closed"      # Normal operation: requests pass through
    OPEN = "open"          # Circuit open: requests fail fast
    HALF_OPEN = "half_open"  # Testing if service has recovered


@dataclass
class CircuitBreakerConfig:
    failure_threshold: int = 5           # Failures needed to open circuit
    reset_timeout: float = 30.0          # Time in seconds before attempting recovery
    half_open_max_requests: int = 3      # Max requests allowed in half-open state
    half_open_success_threshold: int = 2 # Successes needed to close circuit
    max_failure_history: int = 100       # Max failures to track for statistics


class CircuitBreaker:
    """Circuit breaker for upstream service calls."""

    def __init__(self, name: str, config: Optional[CircuitBreakerConfig] = None):
        self.name = name
        self.config = config or CircuitBreakerConfig()
        self.state = CircuitState.CLOSED
        self.failure_count = 0
        self.success_count = 0
        self.last_failure_time: Optional[float] = None
        self._lock = asyncio.Lock()

    async def execute(self, func: Callable, *args, **kwargs) -> Any:
        """Execute function with circuit breaker protection."""

        # Check if circuit is open
        if self.state == CircuitState.OPEN:
```

```
            # Check if reset timeout has elapsed
            if (self.last_failure_time and
                time.time() - self.last_failure_time > self.config.reset_timeout):
                async with self._lock:
                    self.state = CircuitState.HALF_OPEN
                    self.success_count = 0
            else:
                raise CircuitOpenError(f"Circuit '{self.name}' is OPEN")

        # Execute the function
        try:
            result = await func(*args, **kwargs)
            await self._record_success()
            return result
        except Exception as e:
            await self._record_failure()
            raise

    async def _record_success(self) -> None:
        """Record a successful execution."""
        async with self._lock:
            if self.state == CircuitState.HALF_OPEN:
                self.success_count += 1
                if self.success_count >= self.config.half_open_success_threshold:
                    self.state = CircuitState.CLOSED
                    self.failure_count = 0
            else:
                # In CLOSED state, reset failure count on consecutive successes
                self.failure_count = max(0, self.failure_count - 1)

    async def _record_failure(self) -> None:
        """Record a failed execution."""
        async with self._lock:
            self.failure_count += 1
            self.last_failure_time = time.time()

            if self.state == CircuitState.HALF_OPEN:
                # Failure in half-open state: re-open circuit
                self.state = CircuitState.OPEN
                self.success_count = 0
```

```
            elif (self.state == CircuitState.CLOSED and
                  self.failure_count >= self.config.failure_threshold):
                # Too many failures in closed state: open circuit
                self.state = CircuitState.OPEN

    def get_status(self) -> dict:
        """Return current circuit status for monitoring."""
        return {
            "name": self.name,
            "state": self.state.value,
            "failure_count": self.failure_count,
            "success_count": self.success_count,
            "last_failure_time": self.last_failure_time,
            "is_open": self.state == CircuitState.OPEN,
        }

class CircuitOpenError(Exception):
    """Raised when circuit breaker is open and request is rejected."""
    pass
```

Core Logic Skeleton: Range Request Handler

```python
"""
HTTP range request handler for CDN edge cache.
"""

import re
from typing import Optional, Tuple, List
from dataclasses import dataclass


@dataclass
class ByteRange:
    """Represents a byte range request."""
    start: int
    end: Optional[int]  # None means "to end of resource"

    @property
    def length(self) -> Optional[int]:
        """Calculate length of range if end is specified."""
        if self.end is None:
            return None
        return self.end - self.start + 1

    def within(self, content_length: int) -> bool:
        """Check if range is within content bounds."""
        if self.start >= content_length:
            return False
        if self.end is not None and self.end >= content_length:
            return False
        return True

class RangeRequestHandler:
    """Handles HTTP Range and If-Range headers."""

    # Regex for bytes range specifier: bytes=start-end
    BYTES_RANGE_RE = re.compile(r'bytes=(\d+)-(\d*)')

    def __init__(self, cache_storage):
        self.cache = cache_storage

    def parse_range_header(self, range_header: str,
                           content_length: int) -> List[ByteRange]:
```

```
        """
        Parse Range header value into list of ByteRange objects.

        Args:
            range_header: Value of Range header (e.g., "bytes=0-499")
            content_length: Total length of the resource in bytes

        Returns:
            List of valid ByteRange objects
        """
        ranges = []

        # Handle single range: bytes=start-end
        match = self.BYTES_RANGE_RE.match(range_header)
        if match:
            start_str, end_str = match.groups()
            start = int(start_str)
            end = int(end_str) if end_str else None

            # Validate range
            if end is not None and end < start:
                # Invalid range: end before start
                return []

            # Adjust "to end of resource" syntax
            if end is None:
                # bytes=start- means from start to end
                if start < content_length:
                    ranges.append(ByteRange(start, None))
            else:
                # bytes=start-end
                if start < content_length and end < content_length:
                    ranges.append(ByteRange(start, end))

        # TODO: Support multiple ranges: bytes=0-499,1000-1499
        # TODO: Support suffix ranges: bytes=-500 (last 500 bytes)

        return ranges

    async def handle_range_request(self, request_headers: dict,
```

```
                            request_body: bytes,
                            cache_key: str) -> Tuple[int, dict, bytes]:
    """
    Handle HTTP request with Range header.

    Args:
        request_headers: Client request headers
        request_body: Client request body
        cache_key: Cache key for this resource

    Returns:
        Tuple of (status_code, headers, body)
    """
    range_header = request_headers.get('range')
    if_range_header = request_headers.get('if-range')

    # TODO 1: Check cache for full entity using cache_key
    # TODO 2: If full entity cached and fresh:
    #   - Parse range_header to get requested byte ranges
    #   - Extract ranges from cached body
    #   - If single range: return 206 with Content-Range
    #   - If multiple ranges: return 206 with multipart/byteranges
    # TODO 3: If full entity cached but stale, check If-Range header
    #   - If If-Range ETag/date matches: serve range from stale content
    #   - Otherwise: revalidate with origin
    # TODO 4: If cache miss or partial entity only:
    #   - Forward request to upstream with Range header
    #   - On 206 response: cache if possible (prefer full entity)
    #   - On 200 response: cache full entity, extract range for response
    # TODO 5: Handle edge cases:
    #   - Range not satisfiable (416)
    #   - Malformed Range header (ignore, serve full response)
    #   - If-Range with weak validator

    # Placeholder return (implementer should replace)
    return 200, {}, b""

def extract_range_from_body(self, body: bytes,
                            byte_range: ByteRange) -> bytes:
    """
```

```
        Extract byte range from full response body.

        Args:
            body: Complete response body
            byte_range: ByteRange specification

        Returns:
            Extracted byte range
        """
        # TODO 1: Validate that body length >= byte_range.start
        # TODO 2: If byte_range.end is None, extract from start to end
        # TODO 3: If byte_range.end specified, extract start to end (inclusive)
        # TODO 4: Return extracted bytes

        return b""
```

Language-Specific Hints

- **Python's `http.client` vs `aiohttp`**: For range requests, `aiohttp` automatically handles `Range` headers in responses but not in requests. Use manual `Range` header setting with `aiohttp.ClientSession.get(headers={'Range': 'bytes=0-499'})`.
- **Memory Management for Large Files**: When handling range requests for large files, avoid loading entire files into memory. Use Python's `memoryview` or `io.BytesIO` with slicing for efficient partial access.
- **Exception Hierarchy**: Create a custom exception hierarchy for CDN-specific errors:

```python
class CDNError(Exception): pass
class OriginError(CDNError): pass
class CacheError(CDNError): pass
class ConfigurationError(CDNError): pass
```

- **Structured Logging**: Use Python's `structlog` or `logging.config.dictConfig` to create structured logs that include request IDs, cache keys, and failure modes for easier debugging.
- **Testing Network Failures**: Use `unittest.mock.patch` to simulate network timeouts and failures: `@patch('aiohttp.ClientSession.get', side_effect=asyncio.TimeoutError())`.

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Clients receive 502 errors during origin maintenance	Circuit breaker stuck in OPEN state	Check circuit breaker status endpoint; monitor failure count	Implement manual circuit reset API; adjust failure threshold
Range requests return full content instead of partial	Range header parsing failure	Log parsed range values; check for malformed Range headers	Improve Range header parser; handle edge cases like bytes=0-
`Vary: *` responses are being cached	Missing check for `Vary: *`	Add debug logging when `Vary` header is processed	Implement explicit check for `Vary: *` before caching
Stale content served indefinitely after origin recovery	`stale-if-error` logic doesn't reset on origin recovery	Monitor origin health separately from circuit breaker	Reset stale serving when origin returns to healthy state
Different edge nodes serve different versions of same content	Clock skew causing different expiration times	Compare `fetched_at` and `expires_at` across nodes	Implement NTP synchronization; use monotonic clocks for relative TTLs
Thundering herd on new edge node	Failover directs all clients simultaneously	Monitor request rate spike after failover	Implement staggered client redirection using DNS TTL randomization
Memory exhaustion during video streaming	Large range requests cached in full	Monitor cache entry sizes; check for partial vs full caching	Implement size-based cache admission policy; stream without caching large files

Testing Strategy

> **Milestone(s):** All milestones (1 through 5)

Building a Content Delivery Network is a complex distributed systems undertaking. Without a rigorous testing strategy, subtle bugs in cache semantics, race conditions during invalidation, or thundering herd scenarios could undermine the entire system. This section provides a comprehensive blueprint for verifying the CDN works correctly across all five milestones. We'll establish a **multi-layered testing pyramid** that combines unit tests for algorithmic correctness, property-based tests for cache behavior invariants, integration tests for multi-component interactions, and load tests for performance validation. Additionally, we provide concrete **milestone checkpoints**—specific commands to run and expected outputs—that let you verify each milestone's acceptance criteria before moving forward.

Think of testing a CDN as **calibrating a network of smart thermostats**. Each thermostat (edge node) must independently make correct decisions about when to heat (fetch from origin) and when to rely on stored warmth (serve from cache). However, they must also coordinate when the building manager sends a system-wide temperature change (invalidation). Our testing strategy ensures each thermostat's local logic is flawless and that their collective behavior remains consistent even during network partitions or command storms.

Testing Approaches & Properties

A robust CDN testing strategy employs four complementary approaches, each targeting different aspects of the system:

Test Type	Primary Focus	Key Properties Verified	Tools/Techniques
Unit Tests	Individual functions and classes in isolation	- Cache key generation correctness - TTL calculation accuracy - Header parsing edge cases - Data structure operations	`pytest`, `unittest`, mock objects
Property-Based Tests	System invariants under random inputs	- Cache never serves stale fresh content - Invalidation eventually removes banned content - No race conditions in request collapsing - Consistent hashing distribution properties	`hypothesis`, randomized test generators
Integration Tests	Multi-component interactions	- Edge → Shield → Origin request flow - Invalidation propagation across nodes - Geo-routing and failover behavior - Analytics aggregation	Docker containers, test HTTP clients, simulated network partitions
Load & Performance Tests	System behavior under production-like load	- Origin protection during cache miss storms - Memory usage under cache pressure - Request collapsing efficiency - Latency percentiles	`locust`, `k6`, custom load generators, monitoring dashboards

Mental Model: The Four-Layer Inspection Protocol

Imagine inspecting a chain of grocery stores (our CDN). **Unit tests** are like checking each cash register's calculation logic in isolation. **Property-based tests** verify that no matter what combination of items a customer buys (random inputs), the store never charges tax on exempt items (invariants). **Integration tests** simulate the full supply chain—delivery trucks arriving from warehouses (origin) to distribution centers (shield) to stores (edge)—ensuring produce stays fresh through the entire journey. **Load tests** are the Black Friday simulation: we flood the stores with thousands of customers to ensure neither the cash registers nor the stockrooms collapse under pressure.

Property-Based Testing for Cache Correctness

The most subtle bugs in caching systems involve edge cases in the HTTP caching specification. Property-based testing generates thousands of random but valid HTTP headers and verifies our implementation always respects RFC 9111 semantics. Key properties to test include:

1. **Freshness Invariant**: If a cache entry is fresh (current time < `expires_at`), serving it must produce the same semantic result as fetching from origin (excluding explicit `no-cache` directives).
2. **Staleness Propagation**: Once a cache entry becomes stale, the next request must trigger revalidation (unless `stale-while-revalidate` is in effect).
3. **Invalidation Completeness**: After a hard purge of a URL, the next request must result in a cache miss.
4. **Vary Header Consistency**: Two requests with different `Vary` header values must never return the wrong cached variant.
5. **Request Collapsing Idempotence**: N identical concurrent requests must produce exactly one origin fetch.

We express these properties as test functions that use `hypothesis` to generate random cache-control headers, status codes, and request patterns:

```python
# Example property test structure (shown here for illustration, not in main body)

@given(random_cache_control(), random_status_code(), random_headers())

def test_fresh_content_never_triggers_origin_fetch(cache_control, status_code, headers):

    # Hypothesis generates thousands of combinations

    # Test that when content is fresh, edge cache serves it without upstream call

    pass
```

Integration Testing Strategy

Integration tests verify that the CDN's distributed components work together correctly. Since we're building an educational project, we'll use lightweight containerization (Docker Compose) to spin up a mini-CDN with 2 edge nodes, 1 shield, and 1 origin server for testing.

Test Scenarios to Cover:

Scenario	Components Involved	What to Verify
Cache miss cascade	Client → Edge → Shield → Origin	- Request collapsing at shield works - Response flows back correctly - Cache populated at both edge and shield
Geo-routing decision	Client resolver → Multiple edges	- Client directed to nearest healthy edge - Consistent hashing distributes load
Invalidation storm	Admin API → Control plane → All edges	- Purge commands propagate within time window - No stale content served after propagation
Shield failure	Edge → Fallback to origin	- Circuit breaker opens after threshold - Requests bypass shield when unhealthy
Request with `Range` header	Client → Edge (with cached content)	- Partial content correctly served - 206 status code returned

Load Testing for Origin Protection

The primary value of a CDN is protecting the origin during traffic spikes. Load tests verify this protection actually works. We'll simulate:

1. **Cache Miss Storm**: 1000 concurrent requests for an uncached resource with 1-second TTL.
2. **Popular Content Refresh**: 500 requests/second for a resource that expires simultaneously across all edges.
3. **Invalidation Cascade**: Purge a heavily cached resource while it's receiving 200 req/s.

Success criteria:

- Origin request rate stays below configured limit (e.g., 10 req/s during miss storm)
- No client requests time out (all served within 2 seconds)
- Memory usage remains bounded (no leaks during cache churn)

Milestone Checkpoints

Each milestone has specific acceptance criteria. These checkpoints provide concrete commands to run and outputs to verify, serving as **quality gates** before proceeding to the next milestone.

Milestone 1: Edge Cache Implementation

Verification Goal: Ensure the edge cache correctly stores, retrieves, and validates HTTP responses according to RFC 9111.

Test	Command to Run	Expected Output/Behavior	What Could Go Wrong
Basic cache hit	`pytest tests/test_edge_cache.py::test_cache_hit -v`	Test passes: shows cache served response without origin fetch	Cache key mismatch or TTL calculation error
Cache miss with origin fetch	`pytest tests/test_edge_cache.py::test_cache_miss -v`	Test passes: shows cache fetched from origin and stored	Upstream connection failure or response parsing error
Cache key with Vary headers	`pytest tests/test_edge_cache.py::test_vary_header_caching -v`	Test passes: different `Accept-Encoding` values create separate cache entries	Vary header parsing incomplete or cache key collision
Conditional request (304)	`pytest tests/test_edge_cache.py::test_conditional_request -v`	Test passes: `If-None-Match` with valid ETag returns 304, saves bandwidth	ETag comparison logic flawed or headers not passed upstream
LRU eviction under pressure	`python -m tests.capacity_test --requests 1000 --capacity 100`	Memory usage stabilizes, oldest entries evicted, hit ratio reasonable	Memory leak or eviction algorithm not triggered
Cache-Control directive parsing	`pytest tests/test_directives.py -v`	All 15+ directives parsed correctly, precedence rules followed	`s-maxage` ignored in favor of `max-age`, `no-cache` misinterpreted

Manual Verification Steps:

1. Start a test origin server: `python tests/origin_server.py`
2. Start the edge cache: `python edge_server.py --config configs/edge_test.yaml`
3. Make first request: `curl -v http://localhost:8080/image.jpg`
   - Should show `X-Cache: MISS` header
4. Make identical second request:
   - Should show `X-Cache: HIT` header
   - Response identical to first
5. Request with `Accept-Encoding: gzip`:
   - Should create separate cache entry (check logs)
6. Wait for TTL expiration, request again:
   - Should trigger revalidation or fresh fetch

Milestone 2: Cache Invalidation

Verification Goal: Ensure purge, ban, and tag-based invalidation work correctly and propagate to all nodes.

Test	Command to Run	Expected Output/Behavior	What Could Go Wrong
Hard purge by URL	`pytest tests/test_invalidation.py::test_hard_purge -v`	Cache entry removed, next request misses	Entry not fully removed or ghost data remains
Soft purge with revalidation	`pytest tests/test_invalidation.py::test_soft_purge -v`	Entry marked stale, served while background fetch updates	Background thread not started or stale content not served
Surrogate key purge	`pytest tests/test_invalidation.py::test_tag_purge -v`	All entries with tag removed, others remain	Tag index corruption or incomplete cleanup
Ban rule pattern matching	`pytest tests/test_invalidation.py::test_ban_rules -v`	URLs matching pattern invalidated on access	Regex compilation errors or matching logic flawed
Invalidation propagation	`python -m tests.propagation_test --nodes 3`	All nodes receive purge within 500ms	Pub/sub message loss or network partition not handled
Ban rule GC	`pytest tests/test_invalidation.py::test_ban_garbage_collection -v`	Expired ban rules automatically removed	Memory leak from unbounded ban list

Manual Verification Steps:

1. Start 3 edge nodes in separate terminals with shared control plane
2. Cache content on all nodes by making requests to each
3. Execute purge via admin API:

```bash
curl -X PURGE http://edge1:8080/image.jpg
```

4. Verify all nodes show purge in logs
5. Request same URL on each node:
   - All should show `X-Cache: MISS`
6. Test surrogate key purge:
   - Request with `Surrogate-Key: product-123`
   - Purge by tag: `curl -X PURGE -H "Surrogate-Key: product-123" http://edge1:8080/purge-by-tag`
   - Verify all tagged content purged across nodes

Milestone 3: Origin Shield & Request Collapsing

Verification Goal: Ensure shield reduces origin load through request collapsing and protects against thundering herds.

Test	Command to Run	Expected Output/Behavior	What Could Go Wrong
Request collapsing	`pytest tests/test_shield.py::test_request_collapsing -v`	100 concurrent requests → 1 origin fetch	Race condition in coalescing map or timeout too short
Negative caching	`pytest tests/test_shield.py::test_negative_caching -v`	Origin 404s cached briefly, not repeatedly fetched	Negative cache TTL not honored or error responses cached incorrectly
Shield circuit breaker	`pytest tests/test_shield.py::test_circuit_breaker -v`	After origin failures, shield bypasses to origin	Circuit state transitions incorrectly or health checks flawed
Queue overflow protection	`python -m tests.shield_load_test --concurrent 1000`	Queue limits respected, excess requests get 503	Unlimited queue growth causes memory exhaustion
Stale-while-revalidate at shield	`pytest tests/test_shield.py::test_shield_stale_while_revalidate -v`	Shield serves stale while revalidating in background	Background revalidation not triggered or stale content blocked

Manual Verification Steps:

1. Start origin, shield, and single edge node
2. Use `ab` or `wrk` to send 100 concurrent requests for uncached resource:

```bash
wrk -t10 -c100 -d5s http://edge:8080/uncached.html
```

3. Check shield logs: should show exactly 1 origin request

4. Check origin logs: should show 1 request, not 100
5. Cause origin failure (kill origin server)
6. Make several requests:
   - First few might fail through
   - Circuit should open after threshold
   - Subsequent requests should bypass shield (check headers)
7. Restart origin, verify circuit half-open then closed

Milestone 4: Edge Node Distribution & Routing

Verification Goal: Ensure clients route to nearest edge, consistent hashing distributes load, and failover works.

Test	Command to Run	Expected Output/Behavior	What Could Go Wrong
GeoIP routing	`pytest tests/test_routing.py::test_geoip_routing -v`	US IP routes to US edge, EU IP to EU edge	GeoIP database missing or incorrect mapping
Consistent hashing distribution	`python -m tests.distribution_test --keys 10000 --nodes 5`	Keys distributed evenly (within 20% variance)	Hash function skew or virtual nodes not balanced
Node failure and redistribution	`pytest tests/test_routing.py::test_node_failure -v`	Keys remapped minimally (< 30% movement on node loss)	Hash ring update incorrect or health check false positive
Health check integration	`pytest tests/test_health.py -v`	Unhealthy nodes removed from rotation, marked degraded	Health check too aggressive or recovery not detected
Failover latency	`python -m tests.failover_test --kill-node edge-us-1`	Traffic redirects to next-nearest within 5 seconds	DNS TTL too long or health propagation delayed

Manual Verification Steps:

1. Deploy 3 edge nodes in different regions (simulate with different ports)
2. Start GeoDNS resolver or HTTP redirector
3. From simulated US client (`curl -H "X-Client-IP: 8.8.8.8"`):
   - Should reach US edge node (check logs)
4. From simulated EU client (`curl -H "X-Client-IP: 1.1.1.1"`):
   - Should reach EU edge node
5. Kill US edge node process
6. Health check should detect within 2 seconds
7. Subsequent US client requests should route to next closest (EU or ASIA)
8. Verify consistent hashing: cache specific key on node A, kill node A, bring up replacement node A', request same key - should route to different node initially, then re-cache on A'

Milestone 5: CDN Analytics & Performance Optimization

Verification Goal: Ensure analytics track performance accurately and optimizations (compression, range requests) work.

Test	Command to Run	Expected Output/Behavior	What Could Go Wrong
Cache hit ratio tracking	`pytest tests/test_analytics.py::test_hit_ratio -v`	After mixed hits/misses, ratio calculated correctly	Race conditions in counter updates or reset logic flawed
Compression negotiation	`pytest tests/test_compression.py -v`	`Accept-Encoding: gzip` returns compressed, others uncompressed	Compression applied to already binary content or CPU spike
Range request support	`pytest tests/test_range_requests.py -v`	`Range: bytes=0-499` returns 206 with correct content	Byte range math errors or multi-range not handled
Bandwidth accounting	`python -m tests.bandwidth_test --duration 30`	Analytics show correct bytes served, matches actual transfer	Counting omitted for cached responses or compression size not considered
Stale-while-revalidate behavior	`pytest tests/test_stale_optimizations.py -v`	Stale content served while async revalidation occurs	Client receives stale content indefinitely if revalidation fails

Manual Verification Steps:

1. Enable analytics dashboard: `python analytics_dashboard.py`

2. Generate mixed traffic pattern:

```
python tests/traffic_generator.py --pattern mixed --duration 60
```
BASH

3. Check dashboard:
   - Hit ratio should converge to expected value
   - Bandwidth should show savings from cache hits
4. Test compression:

```
curl -H "Accept-Encoding: gzip" -v http://edge:8080/large.json 2>&1 | grep "Content-Encoding"
```
BASH

- Should show `gzip` and reduced size

5. Test range request:

```
curl -H "Range: bytes=1000-1999" -v http://edge:8080/large_video.mp4
```
BASH

- Should return 206 Partial Content
- Should serve from cache if already cached

6. Verify stale-while-revalidate:
   - Request resource, wait for it to go stale
   - Make concurrent requests: should serve stale while one revalidates

Implementation Guidance

Technology Recommendations:

Component	Simple Option	Advanced Option
Test Framework	`pytest` with `pytest-asyncio`	`pytest` with custom fixtures and plugins
Property Testing	`hypothesis` for random test generation	Custom fuzzers with corpus from real traffic
Integration Testing	`docker-compose` with test containers	Kubernetes Kind cluster for realistic orchestration
Load Testing	`locust` for Python-based load scenarios	`k6` for JavaScript-based performance tests
Mock HTTP Servers	`aiohttp` test server or `responses` library	Custom TCP server simulating origin behaviors

Recommended Test Directory Structure:

```
blue-origin-cdn/
├── tests/
│   ├── __init__.py
│   ├── conftest.py                    # Shared pytest fixtures
│   ├── unit/
│   │   ├── test_edge_cache.py        # Milestone 1
│   │   ├── test_directives.py        # Cache-Control parsing
│   │   ├── test_invalidation.py      # Milestone 2
│   │   ├── test_shield.py            # Milestone 3
│   │   ├── test_routing.py           # Milestone 4
│   │   └── test_analytics.py         # Milestone 5
│   ├── integration/
│   │   ├── test_multi_node.py        # Multi-edge scenarios
│   │   ├── test_propagation.py       # Invalidation across nodes
│   │   └── test_full_stack.py        # End-to-end flows
│   ├── property/
│   │   ├── test_cache_properties.py  # Hypothesis-based invariants
│   │   └── test_hashing_properties.py
│   ├── load/
│   │   ├── miss_storm.py             # Cache miss load test
│   │   └── invalidation_storm.py     # Purge load test
│   ├── fixtures/
│   │   ├── origin_server.py          # Test origin implementation
│   │   └── geoip_test_db.mmdb        # Test GeoIP database
│   └── utils/
│       ├── test_client.py            # Async HTTP test client
│       └── assertions.py             # Custom assertions
├── docker-compose.test.yml           # Integration test environment
└── loadtest/
    ├── locustfile.py                 # Locust load test definition
    └── k6_script.js                  # k6 load test script
```

Infrastructure Starter Code: Test Origin Server

Here's a complete, ready-to-use test origin server that simulates various origin behaviors for testing:

PYTHON

```python
# tests/fixtures/origin_server.py

import asyncio

import time

from typing import Dict, List, Optional, Tuple

from aiohttp import web

import gzip

import json

import random


class TestOriginServer:

    """Configurable test origin server for CDN testing."""



    def __init__(self, port: int = 8000):

        self.port = port

        self.request_count = 0

        self.responses: Dict[str, Dict] = {

            '/image.jpg': {

                'status': 200,

                'headers': {'Content-Type': 'image/jpeg', 'Cache-Control': 'public, max-age=3600'},

                'body': b'fake-jpeg-content-' + b'x' * 1000,

                'etag': '"abc123"',

                'delay': 0.1

            },

            '/api/data': {

                'status': 200,

                'headers': {'Content-Type': 'application/json', 'Cache-Control': 'no-cache'},

                'body': json.dumps({'data': 'sensitive', 'version': 1}).encode(),

                'delay': 0.05

            },

            '/large.txt': {

                'status': 200,

                'headers': {'Content-Type': 'text/plain', 'Cache-Control': 'public, max-age=60, stale-while-revalidate=30'},

                'body': b'Line ' + b'\n'.join([f'Line {i}'.encode() for i in range(10000)]),

                'etag': '"large123"',

                'delay': 0.2

            }

        }

        self.failure_mode = None  # 'timeout', '500', 'slow'



    async def handler(self, request: web.Request) -> web.Response:

        """Handle incoming request with configurable behavior."""
```

```
        self.request_count += 1
        path = request.path
        method = request.method

        # Apply failure modes if active
        if self.failure_mode == 'timeout':
            await asyncio.sleep(10)  # Longer than client timeout
            return web.Response(status=504)
        elif self.failure_mode == '500':
            return web.Response(status=500, text='Internal Server Error')
        elif self.failure_mode == 'slow':
            await asyncio.sleep(2)  # Slow but not timeout

        # Check for conditional requests
        if_none_match = request.headers.get('If-None-Match')
        if_modified_since = request.headers.get('If-Modified-Since')

        # Find response configuration
        resp_config = self.responses.get(path)
        if not resp_config:
            resp_config = {
                'status': 404,
                'headers': {'Content-Type': 'text/plain'},
                'body': b'Not Found',
                'delay': 0
            }

        # Apply delay if specified
        if resp_config.get('delay', 0) > 0:
            await asyncio.sleep(resp_config['delay'])

        # Handle conditional requests for resources with ETag
        if if_none_match and resp_config.get('etag'):
            if if_none_match == resp_config['etag']:
                return web.Response(status=304, headers=resp_config['headers'])

        # Build response
        headers = dict(resp_config['headers'])
        if resp_config.get('etag'):
            headers['ETag'] = resp_config['etag']
```

```
        # Handle compression if requested
        accept_encoding = request.headers.get('Accept-Encoding', '')
        body = resp_config['body']
        if 'gzip' in accept_encoding and headers.get('Content-Type', '').startswith('text/'):
            body = gzip.compress(body)
            headers['Content-Encoding'] = 'gzip'
            headers['Vary'] = 'Accept-Encoding'

        # Handle range requests
        range_header = request.headers.get('Range')
        if range_header and path == '/large.txt':
            # Simple byte range handling for testing
            if range_header.startswith('bytes='):
                range_spec = range_header[6:]
                if '-' in range_spec:
                    start_end = range_spec.split('-')
                    start = int(start_end[0]) if start_end[0] else 0
                    end = int(start_end[1]) if len(start_end) > 1 and start_end[1] else len(body) - 1
                    chunk = body[start:end+1]
                    headers['Content-Range'] = f'bytes {start}-{end}/{len(body)}'
                    return web.Response(status=206, headers=headers, body=chunk)

        return web.Response(
            status=resp_config['status'],
            headers=headers,
            body=body
        )

    async def start(self):
        """Start the test origin server."""
        app = web.Application()
        app.router.add_route('*', '/{path:.*}', self.handler)
        runner = web.AppRunner(app)
        await runner.setup()
        site = web.TCPSite(runner, 'localhost', self.port)
        await site.start()
        print(f"Test origin server running on http://localhost:{self.port}")
        return runner
```

```
    def set_failure_mode(self, mode: Optional[str]):
        """Set failure mode for testing circuit breakers."""
        self.failure_mode = mode

    def get_request_count(self) -> int:
        """Get total requests received."""
        return self.request_count

    def reset_counter(self):
        """Reset request counter."""
        self.request_count = 0

# Quick test runner
if __name__ == '__main__':
    server = TestOriginServer(port=9999)
    async def run():
        await server.start()
        await asyncio.Future()  # Run forever
    asyncio.run(run())
```

Core Test Skeleton: Property-Based Cache Validation

PYTHON

```python
# tests/property/test_cache_properties.py

import pytest

from hypothesis import given, strategies as st, settings, assume

from datetime import datetime, timedelta

import time


from blue_origin.cache.edge_cache import CacheEntry, CacheDirectives

from blue_origin.cache.validation import is_response_cacheable


# Strategy for generating random cache control headers

cache_control_strategy = st.dictionaries(

    keys=st.sampled_from([

        'max-age', 's-maxage', 'no-cache', 'no-store', 'must-revalidate',

        'proxy-revalidate', 'public', 'private', 'stale-while-revalidate',

        'stale-if-error'

    ]),

    values=st.one_of(st.none(), st.integers(min_value=0, max_value=31536000)),

    max_size=5

)


@given(

    status_code=st.integers(min_value=200, max_value=599),

    method=st.sampled_from(['GET', 'POST', 'HEAD']),

    cache_control=cache_control_strategy,

    vary_header=st.one_of(st.none(), st.text(max_size=20))

)

@settings(max_examples=1000, deadline=2000)

def test_cacheability_property(status_code, method, cache_control, vary_header):

    """

    Property: is_response_cacheable must be consistent with RFC 9111.


    TODO 1: Only GET and HEAD responses are potentially cacheable

    TODO 2: Status codes 200, 203, 204, 206, 300, 301, 308, 404, 405, 410,

            414, 501 are cacheable by default (unless headers forbid)

    TODO 3: no-store directive makes response uncacheable

    TODO 4: private directive means response cacheable but not by shared caches (CDN)

    TODO 5: Vary: * means response is uncacheable

    TODO 6: Implement the actual logic from RFC 9111 Section 3

    TODO 7: Return True/False based on all conditions above


    Run with: pytest tests/property/test_cache_properties.py -v

    """
```

```python
    # This is a skeleton - hypothesis will generate 1000 random inputs
    # Your implementation should deterministically decide cacheability
    result = is_response_cacheable(status_code, method, cache_control, vary_header)

    # Add assertions that must ALWAYS hold
    if method not in ['GET', 'HEAD']:
        assert result is False, f"Non-GET/HEAD method {method} must not be cacheable"

    if cache_control.get('no-store') is not None:
        assert result is False, "no-store directive must prevent caching"

    if vary_header == '*':
        assert result is False, "Vary: * must prevent caching"

@given(
    fetched_at=st.floats(min_value=0, max_value=1e9),
    max_age=st.one_of(st.none(), st.integers(min_value=0, max_value=31536000)),
    s_maxage=st.one_of(st.none(), st.integers(min_value=0, max_value=31536000)),
    request_time=st.floats(min_value=0, max_value=1e9)
)
def test_freshness_calculation_property(fetched_at, max_age, s_maxage, request_time):
    """
    Property: Freshness calculation must follow Cache-Control precedence.

    TODO 1: s-maxage takes precedence over max-age for shared caches (CDN)
    TODO 2: Calculate expires_at based on fetched_at + relevant TTL
    TODO 3: Entry is fresh if request_time < expires_at
    TODO 4: Handle None values correctly (no expiration)
    TODO 5: Ensure math doesn't overflow or produce negative TTLs

    Run with: pytest tests/property/test_cache_properties.py::test_freshness_calculation_property -v
    """
    assume(request_time >= fetched_at)  # Request can't be before fetch

    # Create directives object
    directives = CacheDirectives(
        s_maxage=s_maxage,
        max_age=max_age,
        no_cache=False,
        no_store=False,
        must_revalidate=False,
```

```
        proxy_revalidate=False,
        public=True,
        private=False,
        stale_while_revalidate=None,
        stale_if_error=None
    )

    # Create cache entry
    entry = CacheEntry(
        key="test",
        url="/test",
        vary_headers={},
        status_code=200,
        headers={},
        body=b"test",
        fetched_at=fetched_at,
        expires_at=0,  # Will be calculated
        last_used_at=fetched_at,
        use_count=1,
        surrogate_keys=[],
        etag=None,
        last_modified=None
    )

    # TODO: Calculate expires_at based on directives
    # TODO: Determine if entry is fresh at request_time
    # TODO: Add assertions about correctness

    # Placeholder assertions
    assert True  # Replace with actual property checks
```

Integration Test Helper: Multi-Node Test Environment

PYTHON

```python
# tests/integration/conftest.py

import pytest

import asyncio

import aiohttp

from typing import Dict, List

import subprocess

import time

import os


@pytest.fixture(scope="session")

def event_loop():

    """Create event loop for async tests."""

    loop = asyncio.get_event_loop_policy().new_event_loop()

    yield loop

    loop.close()


@pytest.fixture(scope="module")

async def test_origin():

    """Start test origin server."""

    from tests.fixtures.origin_server import TestOriginServer

    origin = TestOriginServer(port=8888)

    runner = await origin.start()

    yield origin

    await runner.cleanup()


@pytest.fixture(scope="module")

async def edge_nodes(test_origin):

    """Start multiple edge nodes for testing."""

    nodes = []

    processes = []



    # Start 3 edge nodes on different ports

    for i in range(3):

        port = 9000 + i

        env = os.environ.copy()

        env['EDGE_PORT'] = str(port)

        env['UPSTREAM_URL'] = 'http://localhost:8888'

        env['NODE_ID'] = f'edge-{i}'



        proc = subprocess.Popen(

            ['python', '-m', 'blue_origin.edge_server'],

            env=env,
```

```
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE
        )
        processes.append(proc)
        nodes.append(f'http://localhost:{port}')
        time.sleep(0.5)  # Stagger startup

    # Wait for nodes to be ready
    for node_url in nodes:
        for _ in range(30):  # 30 second timeout
            try:
                async with aiohttp.ClientSession() as session:
                    async with session.get(f'{node_url}/health'):
                        break
            except:
                time.sleep(1)
        else:
            raise RuntimeError(f"Edge node {node_url} failed to start")

    yield nodes

    # Cleanup
    for proc in processes:
        proc.terminate()
        proc.wait(timeout=5)

@pytest.fixture
async def http_client():
    """Create HTTP client for tests."""
    async with aiohttp.ClientSession() as session:
        yield session
```

Milestone Checkpoint Test Commands

Add these to your `Makefile` or test scripts:

```makefile
# Makefile test targets
test-milestone1:
	pytest tests/unit/test_edge_cache.py -v --tb=short
	pytest tests/unit/test_directives.py -v
	python -m tests.capacity_test --requests 1000 --capacity 100

test-milestone2:
	pytest tests/unit/test_invalidation.py -v
	python -m tests.propagation_test --nodes 3 --timeout 500

test-milestone3:
	pytest tests/unit/test_shield.py -v
	python -m tests.shield_load_test --concurrent 100 --duration 10

test-milestone4:
	pytest tests/unit/test_routing.py -v
	python -m tests.distribution_test --keys 10000 --nodes 5
	pytest tests/unit/test_health.py -v

test-milestone5:
	pytest tests/unit/test_analytics.py -v
	pytest tests/unit/test_compression.py -v
	pytest tests/unit/test_range_requests.py -v

test-integration:
	pytest tests/integration/ -v --tb=short

test-property:
	pytest tests/property/ -v --hypothesis-show-statistics

test-all: test-milestone1 test-milestone2 test-milestone3 test-milestone4 test-milestone5 test-integration

load-test:
	cd loadtest && locust -f locustfile.py --host=http://localhost:8080
```

Language-Specific Hints for Python Testing:

1. **Async Testing**: Use `pytest-asyncio` and mark async tests with `@pytest.mark.asyncio`. Ensure event loop fixtures are properly scoped.
2. **Mocking**: Use `unittest.mock` to mock external dependencies. For async mocks, use `AsyncMock`:

```python
from unittest.mock import AsyncMock

upstream_fetch = AsyncMock(return_value=(200, {}, b'response'))
```

3. **Timing-Dependent Tests**: For tests involving timeouts or TTLs, use time mocking:

```python
from unittest.mock import patch

with patch('time.time', return_value=1000):
    # Test logic with fixed time
```

4. **Property-Based Testing with Hypothesis**: Use `@given` decorators with strategies. Build custom strategies for complex data structures. Use `assume()` to filter out invalid test cases.
5. **Testing Concurrent Behavior**: Use `asyncio.gather()` to simulate concurrent requests in tests. Add small jitter to avoid synchronized timing.
6. **Checking Log Output**: Use `caplog` fixture to verify log messages:

```python
def test_cache_hit_logs(caplog):
    with caplog.at_level('INFO'):
        make_request()
        assert 'Cache hit' in caplog.text
```

Debugging Tips for Failed Tests:

Symptom	Likely Cause	How to Diagnose	Fix
Test passes locally but fails in CI	Timing differences or race conditions	Add more logging, increase timeouts in CI environment	Use `asyncio.wait_for` with reasonable timeouts, mock time in sensitive tests
Property-based test finds rare failure	Edge case in cache logic	Hypothesis shrinks input to minimal failing case	Examine shrunk input, add special handling for that case
Integration test flakes randomly	Network race conditions or cleanup issues	Add retry logic, check for proper resource cleanup in fixtures	Ensure proper async cleanup, add unique IDs to requests for tracing
Load test shows memory growth	Cache eviction not working or memory leak	Profile memory during test, check cache size metrics	Implement proper LRU eviction, close HTTP clients, use weak references
Tests pass but manual verification fails	Test mocks too permissive or incomplete	Run test with real components, compare logs	Make mocks more strict, test with actual HTTP traffic

Debugging Guide

> **Milestone(s):** All milestones (1 through 5)

Building a distributed caching system involves numerous moving parts that can fail in subtle ways. This debugging guide provides a practical manual for diagnosing and fixing the most common issues you'll encounter while implementing the Blue Origin CDN. Think of debugging a CDN as **being a traffic investigator at a complex highway interchange**—you need to examine multiple layers (network routes, vehicle behavior, traffic signals) to understand why congestion or accidents occur. This guide helps you systematically trace problems through the caching layers, from client requests to origin responses and everything in between.

Symptom → Cause → Fix Table

The following table catalogs common symptoms, their root causes, diagnostic approaches, and solutions. Each row represents a complete debugging workflow for a specific issue you might encounter.

Symptom	Likely Cause	How to Diagnose	Fix
Origin server receives duplicate identical requests (Thundering herd)	Missing request collapsing at the origin shield layer, or shield bypassed	1. Check shield logs for concurrent requests with same cache key 2. Verify `RequestCoalescingMap` is tracking in-flight requests 3. Monitor `ShieldMetrics.collapsed_requests` counter	Implement proper request deduplication in `ShieldRequestHandler.handle_request()` using `RequestCoalescingMap.get_or_create_future()`
Cache hit ratio remains near 0% despite repeated requests	Cache keys not matching, TTLs set to zero, or responses marked uncacheable	1. Add debug headers (`X-Cache-Key` , `X-Cache-Status`) to responses 2. Check `is_response_cacheable()` logic for false negatives 3. Inspect `Cache-Control` headers from origin	Ensure cache key generation includes all `Vary` header dimensions and respects `CacheDirectives.public` vs `private`
Purge commands don't remove content from all edge nodes	Invalidation propagation delay or failed broadcast	1. Check `InvalidationMessage` delivery via pub/sub broker 2. Verify all edge nodes subscribe to the same channel 3. Monitor control plane logs for delivery failures	Implement acknowledgment mechanism in pub/sub or switch to more reliable propagation (e.g., control plane polling)
Clients receive stale content after origin update	Cache entries not invalidated, TTL too long, or revalidation failing	1. Check `CacheEntry.expires_at` vs current time 2. Verify purge commands actually delete entries 3. Test `CacheEntry.is_fresh()` logic with edge cases	Implement proper invalidation propagation and ensure `stale-while-revalidate` doesn't serve outdated data indefinitely
Edge node crashes under high load	Memory exhaustion from unbounded cache growth or ban rule accumulation	1. Monitor `EdgeMetrics.current_cache_size_bytes` 2. Check `BanRule` list length and TTLs 3. Profile memory usage during load tests	Implement LRU eviction with strict capacity limits and add TTL-based cleanup for ban rules via `InvalidationHandler._gc_ban_rules()`
Geo-routing directs clients to distant edge nodes	GeoIP database inaccuracy or health status misreporting	1. Verify `GeoIPLookup.lookup()` returns correct location for test IPs 2. Check `EdgeNode.health_status` reporting 3. Test consistent hashing ring assignment	Use more accurate GeoIP database, implement weighted health checks, and add fallback routing based on latency measurements
Range requests return incorrect byte ranges	Cache fragmentation or incorrect `Range` header parsing	1. Test `RangeRequestHandler.parse_range_header()` with various inputs 2. Verify cached body length matches `Content-Length` 3. Check for off-by-one errors in `ByteRange.end`	Implement proper boundary checking in `RangeRequestHandler.extract_range_from_body()` and cache full responses, not partials
Compression causes CPU spikes at edge	Compressing all content types or using high compression levels	1. Profile CPU usage by content type 2. Check if already-compressed responses (images, videos) get recompressed 3. Monitor compression ratio vs CPU time trade-off	Skip compression for already-compressed formats and implement configurable compression level per content type
Circuit breaker trips unnecessarily	Failure threshold too low or origin temporary blips treated as failures	1. Check `CircuitBreakerConfig.failure_threshold` value 2. Monitor origin response times vs timeout settings 3. Verify health check logic isn't too sensitive	Adjust circuit breaker parameters based on actual failure patterns and implement exponential backoff for `half_open` state
Negative caching causes 404s to persist	Error response TTL too long or missing invalidation on content creation	1. Check TTL assigned to error responses (should be short) 2. Verify purge commands affect negative cache entries 3. Monitor `ShieldMetrics.negative_cache_hits`	Cache 404/5xx responses with very short TTLs (5-10 seconds) and include them in purge operations
Vary header handling causes cache bloat	Treating each unique header combination as	1. Inspect cache storage for duplicate content with different `Vary` values 2. Check if `Vary: *` is incorrectly cached 3. Monitor memory growth with varied requests	Implement `Vary` header normalization and limit number of variations per URL; reject `Vary: *` from cache

Symptom	Likely Cause	How to Diagnose	Fix
	separate entry without bounds		
Request collapsing causes client timeouts	Collapsing timeout longer than client timeout or deadlock in coalescing map	1. Compare `RequestCoalescingMap._default_timeout` with client timeout 2. Check for stuck futures never fulfilled 3. Monitor queue times in `ShieldMetrics`	Set collapsing timeout shorter than client timeout and implement timeout cleanup in `RequestCoalescingMap._gc_loop()`
Consistent hashing causes hot spots	Uneven virtual node distribution or skewed request patterns	1. Analyze request distribution across edge nodes 2. Check `ConsistentHashRing.replicas_per_node` count 3. Verify hash function distribution uniformity	Increase virtual nodes per physical node and consider weighted consistent hashing based on node capacity
Stale-while-revalidate serves outdated data indefinitely	Background revalidation failing silently or not triggered	1. Check `_revalidate_in_background()` error handling 2. Verify stale entries get revalidation triggered on access 3. Monitor background thread/task completion	Implement proper error recovery in revalidation and fallback to fetch fresh content if revalidation fails repeatedly
Health checks cause cascading failures	Aggressive health checking marks nodes unhealthy during normal load	1. Check health check interval vs node response time under load 2. Verify health metrics thresholds are realistic 3. Monitor `NodeHealth` transitions during traffic spikes	Implement weighted health checks that consider historical performance and add grace periods before marking nodes unhealthy

CDN-Specific Debugging Techniques

Beyond symptom-based diagnosis, effective CDN debugging requires specialized techniques for inspecting the complex interactions between caching layers, routing logic, and distributed state. These methods transform the CDN from a black box into a transparent, observable system where you can trace every request's journey and understand every caching decision.

Adding Diagnostic Headers to HTTP Responses

The simplest yet most powerful debugging technique is augmenting HTTP responses with custom headers that reveal the CDN's internal decision-making. These headers act as **flight data recorders for each request**, documenting its path through the caching infrastructure. Add the following headers to all responses from edge nodes:

- **`X-Cache-Status`**: Indicates whether the response was served from cache (`HIT`), fetched from upstream (`MISS`), revalidated (`REVALIDATED`), or served stale while revalidating (`STALE`)
- **`X-Cache-Key`**: Shows the exact cache key used for lookup, helping identify key generation issues
- **`X-Cache-Age`**: The time in seconds since the cached response was fetched from origin
- **`X-Edge-Node`**: Identifier of the edge node that served the request (crucial for geo-routing debugging)
- **`X-Request-ID`**: Unique identifier propagated through all layers (edge → shield → origin) for request tracing
- **`X-Cache-TTL`**: Remaining time-to-live in seconds for the cached response
- **`X-Vary-Headers`**: Shows which request headers were considered in cache key generation due to `Vary`

To implement these headers, modify the `EdgeRequestHandler.handle_request()` method to append them before returning the response. In production CDNs, these headers are typically enabled via configuration and stripped for external clients, but during development they provide invaluable visibility.

Inspecting Cache State and Metrics

When responses aren't caching as expected, you need to directly examine the cache's internal state. The CDN provides several inspection points:

1. **Cache Storage Dump**: Implement a debug endpoint (e.g., `GET /debug/cache`) that returns a JSON representation of all cache entries, including their keys, URLs, TTLs, and tags. This helps verify what's actually stored versus what you expect to be stored.
2. **Metrics Inspection Endpoint**: Create `/debug/metrics` endpoint exposing `EdgeMetrics`, `ShieldMetrics`, and `TimeWindowedMetrics` in structured format. Monitor:
   - Cache hit ratio trends over time
   - Cache size and eviction rates
   - Request collapsing effectiveness at shield
   - Geo-routing distribution across nodes
3. **Surrogate Key Index Verification**: Add a debug endpoint to inspect `SurrogateKeyIndex` mappings, showing which tags are assigned to which cache keys and vice versa. This is crucial when tag-based purges aren't working.

4. **Ban Rule Audit**: Expose active `BanRule` patterns, their creation times, and match counts to ensure they're being applied correctly and garbage collected when expired.

> **Debugging Insight**: Always compare cache state with actual HTTP behavior. If the cache dump shows an entry exists but requests miss, the issue is likely in cache key generation or request/response header mismatching.

Simulating Geo-Distribution Locally

Testing geo-routing and multi-node behavior typically requires distributed infrastructure, but you can simulate it locally using these techniques:

1. **IP Address Spoofing**: Use tools like `curl --interface` or programming libraries to bind requests to different source IPs. Combine with a test `GeoIP` database that maps specific test IPs to different regions (e.g., `203.0.113.1` → "US-West", `203.0.113.2` → "EU-Central").
2. **Local Edge Node Cluster**: Run multiple edge node instances on different ports, each configured with a different `region` in `EdgeConfig`. Use a simple local DNS resolver or HTTP proxy to route requests based on simulated client IPs.
3. **Consistent Hashing Visualization**: Implement a debug visualization of the `ConsistentHashRing` showing positions of nodes and sample cache keys. This helps identify hot spots and understand key redistribution during node addition/removal.
4. **Network Condition Simulation**: Use tools like `tc` (traffic control on Linux) or `comcast` to simulate latency, packet loss, and bandwidth constraints between nodes, reproducing real-world network partitions.
5. **Control Plane Mock**: Create a mock control plane that coordinates multiple local edge nodes, allowing you to test invalidation propagation, health check aggregation, and configuration distribution without cloud deployment.

Request Tracing and Log Correlation

For complex issues involving multiple hops (client → edge → shield → origin), implement distributed tracing:

1. **Generate Unique Request ID**: At the first CDN entry point, generate a UUID and pass it through all layers via the `X-Request-ID` header.
2. **Structured Logging**: Use JSON-structured logs at each component that includes the request ID, component name, cache key, decision points, and timing information.
3. **Log Aggregation**: Collect logs from all edge nodes, shields, and control plane into a central location (even a simple file during development) where you can filter by request ID to see the complete journey.
4. **Timing Measurements**: Log key timing events: cache lookup duration, upstream fetch time, shield queuing delay, and geo-routing decision time. This helps identify performance bottlenecks.

Testing Cache Invalidation Propagation

Invalidation bugs often manifest as stale content served from some nodes but not others. Test propagation systematically:

1. **Propagation Delay Test**: Issue a purge command, then immediately request the same resource from multiple edge nodes while logging `X-Cache-Status` and `X-Edge-Node`. Measure time until all nodes reflect the purge.
2. **Network Partition Simulation**: Disconnect one edge node from the pub/sub network, issue purges, then reconnect and verify it receives queued invalidation messages.
3. **Sequence Testing**: Send rapid successive purge commands and verify they're processed in order, especially important for soft-purge-then-hard-purge scenarios.
4. **Tag-Based Purge Coverage**: Create resources with multiple surrogate keys, then purge by one tag and verify only appropriate resources are invalidated.

> **Critical Debugging Principle**: When debugging distributed caching systems, **assume eventual consistency by default**. Many issues resolve themselves given time. The question is whether the inconsistency window matches your design expectations.

Common HTTP Edge Case Testing

Several HTTP caching edge cases frequently cause subtle bugs. Create test cases for:

1. `Vary: *`: Ensure responses with this header are never cached, despite other cache directives suggesting they could be.
2. `Cache-Control: no-cache` **vs** `no-store`: Verify `no-cache` allows storage but requires revalidation, while `no-store` prevents any storage.
3. **Multiple** `Cache-Control` **headers**: Test handling of multiple `Cache-Control` headers (should be concatenated per RFC).
4. `Expires` **with invalid date format**: Ensure graceful fallback when `Expires` header contains malformed dates.
5. **Chunked encoding with cache**: Verify chunked responses can be cached (must be reassembled into complete body).
6. `Content-Encoding` **with** `Vary: Accept-Encoding`: Test that compressed and uncompressed versions are stored separately.

By combining these debugging techniques with the symptom-cause-fix table, you'll develop systematic approach to troubleshooting the CDN. Remember that caching bugs often involve **timing, distribution, and state**—reproduce issues under realistic load and network conditions, and always verify behavior at multiple

points in the request flow.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Diagnostic Headers	Custom middleware in request handler	OpenTelemetry tracing with baggage propagation
Cache Inspection	Debug HTTP endpoints with JSON output	Integrated admin UI with real-time metrics
Geo Simulation	Static IP-to-region mapping file	Docker containers with virtual networks and IP routing
Request Tracing	Manual `X-Request-ID` propagation	Distributed tracing (Jaeger/Zipkin) with spans
Logging	Structured JSON to stdout	Centralized log aggregation (Loki/ELK)

B. Recommended File/Module Structure

```
blue-origin-cdn/
├── src/
│   ├── debug/                    # Debugging utilities
│   │   ├── __init__.py
│   │   ├── headers.py            # Diagnostic header injection
│   │   ├── inspector.py          # Cache state inspection
│   │   ├── tracing.py            # Request tracing utilities
│   │   └── simulation.py         # Geo-distribution simulation tools
│   ├── metrics/                  # Metrics collection
│   │   ├── __init__.py
│   │   ├── collectors.py         # Metrics collection logic
│   │   └── exporters.py          # Export to Prometheus/JSON
│   └── tests/
│       ├── __init__.py
│       ├── test_debug_headers.py
│       ├── test_cache_inspection.py
│       └── simulation/           # Simulation tests
│           ├── test_geo_routing.py
│           └── test_invalidation_propagation.py
```

C. Infrastructure Starter Code

Here's a complete, ready-to-use diagnostic header middleware for Python:

PYTHON

```python
# src/debug/headers.py
import time
import uuid
from typing import Dict, Tuple, Optional

class DiagnosticHeaders:
    """Middleware for adding diagnostic headers to HTTP responses."""

    def __init__(self, node_id: str = "local"):
        self.node_id = node_id

    def add_cache_headers(self,
                          status: str,
                          cache_key: str,
                          cache_entry: Optional['CacheEntry'],
                          response_headers: Dict[str, str]) -> Dict[str, str]:
        """Add cache-related diagnostic headers to response."""
        headers = response_headers.copy()

        # Always add cache status and node ID
        headers['X-Cache-Status'] = status
        headers['X-Edge-Node'] = self.node_id

        # Add cache key for debugging
        headers['X-Cache-Key'] = cache_key

        # Add cache age and TTL if we have a cache entry
        if cache_entry:
            current_time = time.time()
            age = int(current_time - cache_entry.fetched_at)
            ttl = int(cache_entry.expires_at - current_time) if cache_entry.expires_at > current_time else 0

            headers['X-Cache-Age'] = str(age)
            headers['X-Cache-TTL'] = str(ttl)

            # Add vary header info if present
            if cache_entry.vary_headers:
                vary_list = ','.join(f"{k}:{v}" for k, v in cache_entry.vary_headers.items())
                headers['X-Vary-Headers'] = vary_list

        return headers
```

```
def generate_request_id(self, existing_id: Optional[str] = None) -> str:
    """Generate or propagate request ID for tracing."""
    if existing_id and existing_id.strip():
        return existing_id
    return f"req_{uuid.uuid4().hex[:16]}"

def add_tracing_headers(self,
                        request_headers: Dict[str, str],
                        response_headers: Dict[str, str]) -> Dict[str, str]:
    """Add distributed tracing headers to response."""
    headers = response_headers.copy()

    # Get or generate request ID
    request_id = self.generate_request_id(
        request_headers.get('X-Request-ID') or request_headers.get('x-request-id')
    )
    headers['X-Request-ID'] = request_id

    # Add timing information
    headers['X-Processed-At'] = str(time.time())

    return headers
```

D. Core Logic Skeleton Code

Here's skeleton code for the cache inspection debug endpoint:

PYTHON

```python
# src/debug/inspector.py

from typing import Dict, Any, List

import json

import time


class CacheInspector:

    """Debug endpoint for inspecting cache state."""


    def __init__(self, cache_storage: 'CacheStorage',

                 key_index: Optional['SurrogateKeyIndex'] = None,

                 ban_rules: Optional[List['BanRule']] = None):

        self.cache = cache_storage

        self.key_index = key_index

        self.ban_rules = ban_rules or []


    def handle_debug_request(self, path: str,

                             query_params: Dict[str, str]) -> Tuple[int, Dict[str, str], bytes]:

        """Handle GET /debug/cache[/subpath] requests."""

        # TODO 1: Parse path to determine what to inspect:

        #   - /debug/cache/entries: List all cache entries

        #   - /debug/cache/keys: List just cache keys

        #   - /debug/cache/tags: Show tag-to-key mappings

        #   - /debug/cache/bans: Show active ban rules


        # TODO 2: For /debug/cache/entries, iterate through cache

        #   and collect entry metadata without full bodies


        # TODO 3: For /debug/cache/tags, use key_index.get_keys_for_tag

        #   to show all tag associations


        # TODO 4: For /debug/cache/bans, list all active ban rules

        #   with their patterns, creation time, and TTL


        # TODO 5: Apply query filters:

        #   - ?url=pattern: Filter by URL pattern

        #   - ?tag=name: Filter by surrogate key

        #   - ?limit=N: Limit number of results


        # TODO 6: Format results as JSON with appropriate structure


        # TODO 7: Return HTTP 200 with application/json content-type
```

```
        # TODO 8: Handle errors gracefully (e.g., missing key_index)

        pass

    def _get_cache_entries_summary(self, limit: int = 100) -> List[Dict[str, Any]]:
        """Get summary of cache entries without full bodies."""
        # TODO 1: Get list of all cache keys from storage

        # TODO 2: For each key, retrieve cache entry metadata

        # TODO 3: Extract key information: url, fetched_at, expires_at,
        #   size, surrogate_keys, hit_count

        # TODO 4: Apply limit and return list of summaries

        pass

    def _get_tag_mappings(self) -> Dict[str, Any]:
        """Get surrogate key index mappings."""
        # TODO 1: Check if key_index is available

        # TODO 2: Return tag_to_keys and key_to_tags structures

        # TODO 3: Include statistics: total tags, avg keys per tag

        pass
```

E. Language-Specific Hints

- **Python's `asyncio` debugging**: Use `asyncio.create_task()` for background revalidation but capture exceptions with `task.add_done_callback()` to avoid silent failures.
- **Memory profiling**: Use `tracemalloc` to track cache memory usage and identify leaks in cache storage implementation.
- **Timing measurements**: Use `time.perf_counter()` for high-resolution timing in critical paths (cache lookups, upstream fetches).
- **GeoIP database**: Use `maxminddb-geolite2` package for development GeoIP database, but remember it's not updated in real-time.

F. Milestone Checkpoint

After implementing debugging features, verify them with:

```bash
# Start edge node with debug endpoints enabled
python src/edge_server.py --port 8080 --debug

# Test diagnostic headers
curl -v http://localhost:8080/image.jpg

# Look for headers:
# X-Cache-Status: MISS (first time) or HIT (subsequent)
# X-Cache-Key: /image.jpg (or with vary dimensions)
# X-Edge-Node: local
# X-Request-ID: req_abcdef1234567890

# Test cache inspection endpoint
curl http://localhost:8080/debug/cache/entries?limit=5

# Expected: JSON array of cache entries with metadata
# [{"key": "...", "url": "...", "fetched_at": 123.45, ...}]

# Test invalidation and verify via debug endpoint
curl -X PURGE http://localhost:8080/image.jpg
curl http://localhost:8080/debug/cache/entries | grep image.jpg
# Should show no entries or stale status
```

G. Debugging Tips

Symptom	How to Diagnose	Fix
Diagnostic headers missing	Check middleware order in request handler	Ensure `DiagnosticHeaders` processes responses before sending
Cache inspection endpoint slow	Profile `_get_cache_entries_summary` with large cache	Implement pagination and limit full scan frequency
Request ID not propagating	Trace headers through edge → shield → origin	Ensure each component reads and forwards `X-Request-ID`
Simulation not reflecting real behavior	Compare latency measurements with production-like conditions	Add jitter, packet loss, and bandwidth constraints to simulation

Future Extensions

> **Milestone(s):** This section explores potential enhancements beyond the foundational five milestones, showing how the current architecture accommodates future growth and more advanced CDN features.

The "Blue Origin" CDN you've built represents a solid foundation for content delivery, but like any production system, it will naturally evolve to meet new requirements and leverage emerging technologies. This section explores several **natural extensions** that build upon the existing architecture without requiring fundamental redesigns. Each extension demonstrates how the modular, layered design enables incremental enhancement—a key principle of sustainable system architecture.

Mental Model: The Highway System Evolution

Imagine the current CDN as a basic interstate highway system connecting cities (origin) to suburbs (users) with well-designed on-ramps (edge nodes), rest stops (shields), and traffic control systems (routing). The foundational system efficiently moves standard vehicles (HTTP content) between points. Future extensions are analogous to:

1. **Express lanes for predictable commuters** → **Prefetching** anticipates regular traffic patterns
2. **Vehicle modification stations** → **Image optimization** transforms content for different vehicles
3. **Real-time convoy coordination** → **WebSocket proxying** maintains persistent connections

4. **Integration with shipping warehouses** → **Object storage** connects to cloud-native origins

Each extension builds upon the existing infrastructure rather than replacing it, demonstrating the power of well-defined interfaces and separation of concerns.

Potential Enhancements

1. Predictive Prefetching and Cache Warming

What it is: Automatically fetching content before users request it, based on access patterns, link analysis, or explicit hints.

Why it's valuable: Dramatically improves cache hit ratios for predictable traffic patterns (daily content updates, scheduled releases, trending content). Reduces perceived latency to near-zero for prefetched resources.

How it integrates with current design:

Integration Point	Extension Required
Edge Cache Storage	New `prefetch_queue` field in `EdgeConfig` to manage background fetch jobs
Control Plane	New prefetch scheduler analyzing access logs and content relationships
Edge Node Routing	Enhanced `EdgeRequestHandler` with predictive logic before cache check
Analytics	New `PrefetchMetrics` tracking hit rates on prefetched content

Design considerations:

> **Decision: Client-Driven vs. Server-Driven Prefetching**
>
> - **Context**: We need to decide who initiates prefetch operations—the edge node (server-driven) or the client (via hints like Link headers).
> - **Options Considered**:
>   1. **Server-driven predictive prefetching**: Edge nodes analyze access patterns and automatically fetch likely-next resources
>   2. **Client-driven hint-based prefetching**: Use HTTP `Link` headers with `rel="prefetch"` or `rel="preload"`
>   3. **Hybrid approach**: Combine server predictions with client hints for highest accuracy
> - **Decision**: Start with client-driven prefetching using standard HTTP headers
> - **Rationale**: Client hints are standardized (RFC 5988), deterministic, and don't require complex machine learning. Server-driven prefetching risks wasteful bandwidth usage if predictions are wrong.
> - **Consequences**: Simpler implementation, respects origin bandwidth, but misses opportunities where clients don't send hints.

Implementation approach:

1. Extend `EdgeRequestHandler.handle_request()` to parse `Link` headers with `rel="prefetch"` or `rel="preload"`
2. Add background worker pool to asynchronously fetch linked resources
3. Store prefetched content with lower priority (evicted first under memory pressure)
4. Add `X-CDN-Prefetch: hit/miss` diagnostic headers to track effectiveness

Challenges:

- **Cache pollution**: Prefetched content that's never used wastes storage and bandwidth
- **Origin load spikes**: Aggressive prefetching could overwhelm origins
- **Priority inversion**: Prefetching shouldn't block real user requests

Solution patterns:

- Implement prefetch budget limiting (max concurrent prefetches per edge)
- Use TTL-based decay for prefetch priority (older predictions less valuable)
- Add `prefetch` flag to `CacheEntry` to track prefetched vs. demand-fetched content

2. Image Optimization at the Edge

What it is: On-the-fly resizing, format conversion (WebP/AVIF), compression, and quality adjustment for images based on client device and network conditions.

Why it's valuable: Images typically constitute 60-80% of page weight. Optimizing them at the edge reduces bandwidth costs, improves page load times, and provides better user experience across diverse devices.

How it integrates with current design:

Component	Enhancement Required
Edge Cache	New `ImageOptimizer` component with format detection and conversion
Cache Key Generation	Include image transformation parameters in cache key
Content Negotiation	Enhanced `Accept` header parsing for image formats
Response Headers	Add `Content-DPR`, `Content-Width` headers for responsive images

Key data structures:

Structure Name	Fields	Purpose
`ImageTransformation`	`format: str`, `width: int`, `height: int`, `quality: int`, `crop: bool`, `dpr: float`	Defines transformation parameters for an image
`DeviceProfile`	`dpi: int`, `max_width: int`, `supported_formats: List[str]`, `network_type: str`	Client device capabilities for adaptive optimization
`OptimizedImageCacheKey`	`original_key: str`, `transformation: ImageTransformation`, `variant_hash: str`	Composite key for transformed image variants

Workflow:

1. Client request includes `Accept` header with preferred image formats and `DPR` (Device Pixel Ratio) header
2. `EdgeRequestHandler` checks for existing optimized variant in cache
3. On miss, fetch original from upstream (or from cache if original exists)
4. Apply transformations using `ImageOptimizer.transform(image_bytes, transformation)`
5. Store transformed variant with separate `CacheEntry` and TTL
6. Serve optimized image with appropriate `Content-Type` and `Vary: Accept, DPR`

Performance optimization strategies:

Strategy	Implementation	Benefit
Lazy transformation	Only transform on demand, not preemptively	Reduces CPU and storage waste
Progressive rendering	Serve low-quality placeholder first, then enhance	Improves perceived performance
Format fallback chain	Try WebP → JPEG → PNG based on client support	Maximizes compression benefits
Dimension-aware caching	Cache common sizes (320w, 640w, 1024w, etc.)	Increases cache hit ratio across devices

Challenges and solutions:

⚠️ **Pitfall: CPU exhaustion at edge**

- **Problem**: Image transformation is CPU-intensive; under heavy load, edge nodes could become compute-bound
- **Solution**: Implement request queuing with priority (user requests > optimization), and limit concurrent transformations per node

⚠️ **Pitfall: Storage explosion**

- **Problem**: Each image could have dozens of variants (formats × sizes × qualities)
- **Solution**: Use LRU with lower TTL for variants, implement variant garbage collection based on access patterns

3. WebSocket Proxying and Edge Compute

What it is: Extending the CDN to proxy WebSocket connections and execute lightweight compute functions at the edge.

Why it's valuable: Modern applications increasingly use real-time bidirectional communication (chat, gaming, collaborative editing). Edge compute enables custom logic execution close to users (A/B testing, personalization, authentication).

How it integrates with current design:

Current Component	WebSocket Extension	Edge Compute Extension
Edge Node	WebSocket handshake handling, connection pooling	Python/JavaScript runtime sandbox
Cache Storage	Session state storage (limited TTL)	Function code cache with invalidation
Routing	Connection-aware sticky routing	Function registry and versioning
Metrics	Connection count, message rates, uptime	Execution time, memory usage, error rates

WebSocket proxying architecture:

```
Client ──WS──> Edge Node ──WS──> Origin
          │                    │
          ├── Message Cache ────┘ (for pub/sub patterns)
          └── Connection Pool ──── (reuse origin connections)
```

Key considerations for WebSocket support:

1. **Connection lifecycle management**: WebSocket connections are long-lived (hours/days vs HTTP's seconds)
2. **Message caching**: Some pub/sub patterns benefit from edge caching of broadcast messages
3. **Sticky routing**: Once a WebSocket connects to an edge node, subsequent HTTP requests should route to same node (for session consistency)
4. **Protocol upgrade handling**: Proper HTTP/1.1 Upgrade header processing with fallback

Edge compute implementation pattern:

```python
# Pseudo-structure (not Layer 1 code, but conceptual)

class EdgeFunction:

    name: str

    version: str

    code_hash: str  # For cache invalidation

    runtime: str    # "python", "javascript"

    memory_limit_mb: int

    timeout_ms: int

    env_vars: Dict[str, str]



class EdgeFunctionRuntime:

    async def execute(function: EdgeFunction,

                      request: HttpRequest,

                      context: ExecutionContext) -> HttpResponse:

        # Isolate in sandbox, enforce limits, collect metrics
```

Security considerations critical for edge compute:

Concern	Mitigation Strategy
Code injection	Strict sandboxing (seccomp, namespaces, cgroups)
Resource exhaustion	Hard limits on memory, CPU time, execution duration
Data leakage	Encryption at rest, process isolation between tenants
DoS amplification	Rate limiting per function, global resource quotas

Integration with cache invalidation: Edge functions can trigger cache purges programmatically, enabling dynamic content generation with automatic invalidation.

4. Object Storage Integration as Origin

What it is: Treating cloud object storage (AWS S3, Google Cloud Storage, Azure Blob Storage) as a first-class origin, with optimized protocols and caching semantics.

Why it's valuable: Modern applications increasingly store static assets in object storage. Direct integration reduces latency (no intermediary web server) and leverages storage-native features (versioning, lifecycle policies).

How it integrates with current design:

CDN Component	Object Storage Integration
Origin Shield	S3 API client with signature v4 authentication
Cache Invalidation	S3 event notification integration (purge on object update)
Range Requests	Native support for S3's multipart and range requests
TTL Management	Respect S3 metadata (Cache-Control headers on objects)

Optimization opportunities specific to object storage:

1. **Parallel range fetching**: For large files, fetch chunks in parallel from S3
2. **Prefix-based caching**: Special handling for directory-like listings
3. **Version-aware caching**: Cache multiple versions of same object key
4. **Lifecycle synchronization**: Automatically purge when S3 lifecycle policy deletes objects

Data flow for S3-backed origin:

```
Edge Node → Origin Shield → S3 API Endpoint
    ↑           ↑               ↑
    Cache     Transform      Authentication
    Layer    HTTP to S3       (SigV4)
              Protocol
```

Implementation details:

- Extend `_fetch_from_upstream()` in `ShieldRequestHandler` to handle S3-specific HTTP headers (`x-amz-*`)
- Implement S3 authentication (AWS Signature Version 4) for private buckets
- Parse S3 error responses and convert to appropriate HTTP status codes
- Handle S3's multipart upload references for large objects

Challenges and solutions:

⚠️ **Pitfall: Authentication token management**

- **Problem**: S3 requires frequent signature rotation; hardcoding credentials is insecure
- **Solution**: Implement IAM role assumption with temporary credentials, refreshed automatically

⚠️ **Pitfall: S3 eventual consistency**

- **Problem**: S3 offers eventual consistency for overwrite PUTs and DELETEs
- **Solution**: Implement read-after-write consistency verification or use S3 Strong Consistency where available

5. Additional Extension Ideas

Real-time Analytics Pipeline: Extend the `EdgeMetrics` collection to stream to a time-series database (InfluxDB, TimescaleDB) for real-time dashboards and anomaly detection.

Advanced Compression: Add Zstandard (zstd) support alongside Brotli and gzip, with adaptive compression level based on content type and client capabilities.

Video Streaming Optimizations:

- HLS/DASH manifest manipulation at edge
- Per-title encoding ladder optimization
- Buffer management for adaptive bitrate streaming
- Prefetching of video segments based on playback rate

Security Enhancements:

- DDoS protection with rate limiting and challenge-response
- WAF (Web Application Firewall) rules execution at edge
- Bot detection and mitigation
- TLS 1.3 with early data support

Multi-CDN Failover: Integrate with commercial CDNs as fallback origins, implementing health-based routing between your CDN and third-party providers.

Architectural Principles for Extensibility

The current "Blue Origin" CDN design follows several principles that make these extensions feasible:

1. **Interface-based design**: Clear boundaries between components (cache storage, request handlers, routing) allow swapping implementations

2. **Pluggable middleware**: The request processing pipeline can be extended with additional handlers (compression, optimization, authentication)
3. **Configuration-driven behavior**: `EdgeConfig` provides hooks for enabling/disabling features without code changes
4. **Observability first**: Built-in metrics collection makes it easy to track the impact of new features
5. **Graceful degradation**: Circuit breakers and fallback mechanisms ensure new features don't break core functionality

Prioritization Framework

When deciding which extensions to implement first, consider this decision matrix:

Extension	User Impact	Implementation Complexity	Operational Overhead	Alignment with Trends
Image Optimization	High (60%+ bandwidth savings)	Medium (requires image libraries)	Medium (CPU intensive)	High (Core Web Vitals)
WebSocket Proxying	Medium (real-time apps)	High (stateful connections)	High (connection management)	Medium (growing but not universal)
Object Storage Integration	High (cloud-native apps)	Low (protocol translation)	Low (stateless)	High (cloud migration)
Predictive Prefetching	Medium (reduced latency)	High (ML/pattern analysis)	Medium (extra origin load)	Medium (established technique)

> **Design Insight**: Start with object storage integration—it provides immediate value for modern applications with relatively low complexity. Follow with image optimization, which delivers tangible performance improvements users can feel.

Testing Extended Functionality

Each extension requires specific testing strategies:

Extension	Test Category	Specific Considerations
Image Optimization	Visual regression	Ensure transformations don't corrupt images
WebSocket Proxying	Load/stress testing	Handle thousands of concurrent connections
Object Storage	Integration testing	Mock S3 API with moto or localstack
Edge Compute	Security testing	Fuzz testing for sandbox escape vulnerabilities

Migration Path from Foundation to Extended CDN

The modular design allows incremental adoption:

Phase 1: Foundation (Milestones 1-5) → Basic caching, routing, shielding **Phase 2: Optimization** (+Image optimization, +Advanced compression) → Better performance **Phase 3: Modernization** (+Object storage, +Prefetching) → Cloud-native compatibility
Phase 4: Real-time (+WebSocket, +Edge compute) → Application platform

Each phase delivers value independently while building toward a comprehensive edge delivery platform.

Implementation Guidance

While full implementation of all extensions is beyond this foundational guide, here's how to approach the first natural extension: **Object Storage Integration**.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
S3 Client	`boto3` (Python AWS SDK)	Custom HTTP client with connection pooling
Authentication	IAM instance profiles	AssumeRole with temporary credentials
Error Handling	Basic retry with exponential backoff	Circuit breaker with fallback to HTTP origin
Testing	`moto` library for mocking	LocalStack for full S3 API simulation

B. Recommended Module Structure

```
blue_origin/
├── origins/                  # New package for origin implementations
│   ├── __init__.py
│   ├── base.py               # Abstract OriginClient
│   ├── http_origin.py        # Existing HTTP origin client
│   ├── s3_origin.py          # New S3 origin client
│   └── multi_origin.py       # Fallback across multiple origins
├── shield/
│   └── handler.py            # Updated to use OriginClient interface
└── config/
    └── origin_config.py      # Origin-specific configuration
```

C. Infrastructure Starter Code

PYTHON

```python
# origins/base.py - Abstract interface for origin implementations

from abc import ABC, abstractmethod

from typing import Dict, Tuple, Optional

import asyncio


class OriginClient(ABC):

    """Abstract base class for origin implementations."""



    @abstractmethod

    async def fetch(

        self,

        path: str,

        method: str = "GET",

        headers: Optional[Dict[str, str]] = None,

        body: Optional[bytes] = None

    ) -> Tuple[int, Dict[str, str], bytes]:

        """Fetch resource from origin."""

        pass



    @abstractmethod

    def should_cache_response(self, status_code: int, headers: Dict[str, str]) -> bool:

        """Determine if response from this origin should be cached."""

        pass



    @abstractmethod

    def get_cache_key_prefix(self) -> str:

        """Get prefix for cache keys from this origin (for isolation)."""

        pass


# origins/s3_origin.py - S3 implementation starter

import hashlib

import hmac

from datetime import datetime

from urllib.parse import urlparse, quote

from typing import Dict, Tuple, Optional

import aiohttp


class S3OriginClient(OriginClient):

    """Origin client for Amazon S3 and compatible object storage."""



    def __init__(
```

```
        self,
        endpoint: str,
        bucket: str,
        region: str = "us-east-1",
        access_key: Optional[str] = None,
        secret_key: Optional[str] = None,
        session_token: Optional[str] = None,
        use_iam_role: bool = False
    ):
        self.endpoint = endpoint.rstrip('/')
        self.bucket = bucket
        self.region = region
        self.access_key = access_key
        self.secret_key = secret_key
        self.session_token = session_token
        self.use_iam_role = use_iam_role

        # Connection pool for S3 requests
        self.session = aiohttp.ClientSession()

    def _sign_request(self, method: str, path: str, headers: Dict[str, str]) -> Dict[str, str]:
        """Generate AWS Signature Version 4 for the request."""
        # TODO: Implement AWS SigV4 signing
        # This is complex but follows a standard algorithm:
        # 1. Create canonical request
        # 2. Create string to sign
        # 3. Calculate signature
        # 4. Add authorization header
        pass

    async def fetch(
        self,
        path: str,
        method: str = "GET",
        headers: Optional[Dict[str, str]] = None,
        body: Optional[bytes] = None
    ) -> Tuple[int, Dict[str, str], bytes]:
        """Fetch object from S3."""
        if headers is None:
            headers = {}
```

```
    # Build S3 URL
    s3_path = quote(path.lstrip('/'))
    url = f"{self.endpoint}/{self.bucket}/{s3_path}"

    # Add S3-specific headers
    s3_headers = headers.copy()
    if body:
        s3_headers['Content-Length'] = str(len(body))

    # Sign the request if using credentials
    if self.access_key and self.secret_key:
        s3_headers = self._sign_request(method, path, s3_headers)

    # Make request to S3
    async with self.session.request(
        method=method,
        url=url,
        headers=s3_headers,
        data=body
    ) as response:
        status = response.status
        response_headers = dict(response.headers)
        response_body = await response.read()

        # Convert S3 errors to appropriate HTTP status
        if status == 404 and 'Error' in response_headers.get('x-amz-error-code', ''):
            # S3 returns 404 for missing objects, which is correct
            pass
        elif status == 403:
            # Permission denied - might want to log this differently
            pass

        return status, response_headers, response_body

def should_cache_response(self, status_code: int, headers: Dict[str, str]) -> bool:
    """S3-specific caching logic."""
    # Cache successful GETs and 404s (negative caching)
    if status_code == 200:
        return True
```

```
        elif status_code == 404:
            # Negative cache 404s for shorter TTL
            return True
        return False

    def get_cache_key_prefix(self) -> str:
        """Isolate S3 cache entries from HTTP origin entries."""
        return f"s3:{self.bucket}:"

    async def close(self):
        """Cleanup connection pool."""
        await self.session.close()

# config/origin_config.py - Configuration for multiple origin types
from enum import Enum
from typing import Union, Dict, Any
from pydantic import BaseModel

class OriginType(str, Enum):
    HTTP = "http"
    S3 = "s3"
    GOOGLE_STORAGE = "google_storage"
    AZURE_BLOB = "azure_blob"

class HttpOriginConfig(BaseModel):
    url: str
    health_check_path: str = "/health"
    timeout_seconds: int = 30

class S3OriginConfig(BaseModel):
    endpoint: str
    bucket: str
    region: str = "us-east-1"
    access_key: Optional[str] = None
    secret_key: Optional[str] = None
    use_iam_role: bool = False
    force_path_style: bool = False

class OriginConfig(BaseModel):
    type: OriginType
    config: Union[HttpOriginConfig, S3OriginConfig, Dict[str, Any]]
    fallback_to: Optional[List[str]] = None  # List of other origin IDs to try
```

D. Core Logic Skeleton for Shield Integration

```python
# shield/handler.py - Updated to support multiple origin types

import asyncio

from typing import Dict, Optional, Tuple

from origins.base import OriginClient

from origins.s3_origin import S3OriginClient

from origins.http_origin import HttpOriginClient


class ShieldRequestHandler:

    def __init__(

        self,

        cache: CacheStorage,

        origin_configs: Dict[str, OriginConfig],  # Multiple origins by ID

        default_origin: str,  # ID of default origin

        coalescing_map: RequestCoalescingMap,

        max_concurrent_requests: int = 100

    ):

        self.cache = cache

        self.origin_configs = origin_configs

        self.default_origin = default_origin

        self.coalescing_map = coalescing_map

        self._origin_semaphore = asyncio.Semaphore(max_concurrent_requests)



        # Initialize origin clients

        self.origin_clients: Dict[str, OriginClient] = {}

        self._init_origin_clients()



    def _init_origin_clients(self):

        """Initialize origin clients from configuration."""

        for origin_id, config in self.origin_configs.items():

            if config.type == OriginType.HTTP:

                self.origin_clients[origin_id] = HttpOriginClient(

                    base_url=config.config.url,

                    timeout=config.config.timeout_seconds

                )

            elif config.type == OriginType.S3:

                s3_config = config.config

                self.origin_clients[origin_id] = S3OriginClient(

                    endpoint=s3_config.endpoint,

                    bucket=s3_config.bucket,

                    region=s3_config.region,
```

```
                access_key=s3_config.access_key,
                secret_key=s3_config.secret_key,
                use_iam_role=s3_config.use_iam_role
            )
        # TODO: Add other origin types

def _determine_origin(self, request_headers: Dict[str, str]) -> str:
    """Determine which origin to use for this request."""
    # TODO 1: Check for X-Origin-Override header for explicit origin selection
    # TODO 2: Parse URL path to determine bucket/namespace for routing
    # TODO 3: Check origin health status (circuit breakers)
    # TODO 4: Return origin ID or raise exception if no healthy origin
    return self.default_origin

async def handle_request(
    self,
    request_headers: Dict[str, str],
    request_body: Optional[bytes]
) -> Tuple[int, Dict[str, str], bytes]:
    # ... existing cache logic ...

    # On cache miss or need to revalidate:
    origin_id = self._determine_origin(request_headers)
    origin_client = self.origin_clients[origin_id]

    # TODO: Add origin-specific cache key prefix
    cache_key = self._generate_cache_key(request_headers, origin_id)

    # Use the generic origin client interface
    async with self._origin_semaphore:
        status, headers, body = await origin_client.fetch(
            path=request_path,
            method=request_method,
            headers=request_headers,
            body=request_body
        )

    # TODO: Store with origin-specific TTL logic
    if origin_client.should_cache_response(status, headers):
        cache_entry = CacheEntry.from_upstream_response(
```

```
                key=cache_key,
                url=request_url,
                vary_headers=vary_dict,
                status_code=status,
                headers=headers,
                body=body,
                surrogate_keys=self._extract_surrogate_keys(headers)
            )
            await self.cache.store(cache_entry)

        return status, headers, body

    async def _revalidate_in_background(
        self,
        cache_key: str,
        stale_entry: CacheEntry,
        request_headers: Dict[str, str]
    ):
        """Revalidate with appropriate origin based on cache entry metadata."""
        # TODO 1: Extract origin ID from cache key or entry metadata
        # TODO 2: Get origin client for that origin
        # TODO 3: Perform conditional revalidation (If-None-Match, If-Modified-Since)
        # TODO 4: Update cache if changed, respecting origin-specific caching rules
        pass
```

E. Language-Specific Hints for Python

- Use `aioboto3` for async S3 operations instead of synchronous `boto3`
- Implement connection pooling with `aiohttp.ClientSession` for S3 requests
- Use `pydantic` for configuration validation with discriminated unions
- Consider `smart_open` library for streaming large files from S3 efficiently
- For image optimization, `Pillow` is the standard library, but consider `wand` (ImageMagick bindings) for advanced transformations

F. Testing Extended Features

```python
# tests/test_s3_origin.py - Example test with moto

import pytest

import boto3

from moto import mock_s3

from origins.s3_origin import S3OriginClient


@mock_s3

@pytest.mark.asyncio

async def test_s3_origin_fetch():

    """Test fetching object from mocked S3."""

    # Setup mock S3

    s3 = boto3.client('s3', region_name='us-east-1')

    s3.create_bucket(Bucket='test-bucket')

    s3.put_object(

        Bucket='test-bucket',

        Key='test/image.jpg',

        Body=b'test image content',

        ContentType='image/jpeg'

    )



    # Create S3 origin client

    client = S3OriginClient(

        endpoint='http://localhost:4566',  # LocalStack default

        bucket='test-bucket',

        region='us-east-1',

        access_key='test',

        secret_key='test'

    )



    # Fetch object

    status, headers, body = await client.fetch('test/image.jpg')



    assert status == 200

    assert headers['Content-Type'] == 'image/jpeg'

    assert body == b'test image content'



    await client.close()
```

G. Migration Strategy

1. **Phase 1**: Add S3 origin support alongside existing HTTP origin
2. **Phase 2**: Implement origin routing based on URL prefixes (`/s3/*` → S3, `/*` → HTTP)

3. **Phase 3**: Add health checking and failover between origins
4. **Phase 4**: Implement S3 event notifications for cache invalidation

Each phase can be tested and deployed independently, demonstrating the value of incremental extension to a well-architected foundation.

Glossary

> **Milestone(s):** All milestones (1 through 5)

This glossary provides definitions for key terms, acronyms, and domain-specific vocabulary used throughout the Blue Origin CDN design document. Understanding these terms is essential for navigating the complex distributed systems concepts involved in building a content delivery network.

Term Definitions

Term	Definition
Anycast	A network addressing and routing method where the same IP address is announced from multiple locations. When a client sends a request to an anycast IP, network routers direct it to the nearest (in terms of routing hops) location announcing that IP. This is commonly used in CDNs for global load balancing without DNS.
Background Revalidation	The process of asynchronously fetching updated content from the origin server while continuing to serve stale cached content to clients. This enables `stale-while-revalidate` behavior where users get fast responses while the cache is refreshed in the background.
Ban	A pattern-based invalidation rule that matches URLs using wildcards or regular expressions. When a URL matches a ban rule, the corresponding cache entry is either immediately removed (hard ban) or marked stale (soft ban). Bans are useful for invalidating entire categories of content like `/api/v1/users/*`.
Ban Rule	A data structure representing a pattern-based invalidation rule with fields including `pattern` (string pattern to match URLs), `created_at` (timestamp), and `is_soft` (boolean indicating whether it's a soft or hard ban). Ban rules have TTLs and are garbage collected when expired.
Byte Range	A specification of start and end byte positions for partial content delivery, represented as a tuple `(start, end)` where `end` is optional. Used in HTTP Range requests to request specific portions of large files like videos.
Cache Entry	The core data structure holding a cached HTTP response, including fields like `key` (unique cache identifier), `url` (original request URL), `vary_headers` (dictionary of Vary header values), `status_code`, `headers`, `body`, `fetched_at` (timestamp when fetched), `expires_at` (timestamp when entry expires), `last_used_at` (timestamp of last access), `use_count` (access count for LFU), `surrogate_keys` (list of tags), `etag`, and `last_modified`.
Cache Hit	A request that is successfully served from the edge cache without contacting the origin server. This represents optimal performance as the response is delivered from a nearby location with minimal latency.
Cache Hit Ratio	The percentage of requests served from cache versus total requests, calculated as `hits / (hits + misses)`. A key performance metric for CDNs indicating caching efficiency. Higher ratios mean less origin load and faster responses.
Cache Key	A unique identifier for a cached response constructed from the URL and values of headers specified in the Vary header. For example, if Vary includes "Accept-Encoding", then requests for `/image.jpg` with "gzip" and "br" Accept-Encoding values generate different cache keys.
Cache Miss	A request that cannot be served from cache and must be fetched from the upstream source (origin shield or origin server). This occurs when content is not cached, has expired, or has been invalidated.
Cache Miss Storm	A sudden surge of cache misses that overwhelms the origin server, often occurring when popular content expires simultaneously or after a cache invalidation. Origin shielding and request collapsing mitigate this problem.
Cache Poisoning	A security issue where malicious or incorrect content is cached and served to users. Can occur due to bugs in cache key generation, missing validation, or origin server compromise. Defensive measures include careful cache key construction and origin authentication.
Cache State Dump	A JSON representation of all cache entries for inspection and debugging purposes, typically exposed via a `/debug/cache` endpoint. Includes metadata but not response bodies to avoid exposing sensitive data.
Cache Storage	An abstract interface for cache storage implementations (in-memory, disk-based, distributed). Provides methods for storing, retrieving, and deleting cache entries. The CDN's caching logic works with this interface regardless of the underlying storage.
Circuit Breaker	A resilience pattern that prevents cascading failures by failing fast when upstream services are unhealthy. The circuit has three states: CLOSED (normal operation), OPEN (fail fast, no requests sent), and HALF_OPEN (testing if upstream has recovered). Implemented via the `CircuitBreaker` class with configurable thresholds.
CircuitBreakerConfig	Configuration parameters for a circuit breaker including `failure_threshold` (number of failures before opening), `reset_timeout` (time before attempting recovery), `half_open_max_requests` (max requests in half-open state), `half_open_success_threshold` (successes needed to close), and `max_failure_history` (how many failures to track).
Cold Start	The performance degradation that occurs when a new edge node starts with an empty cache. Until the cache warms up with frequently requested content, most requests will be cache misses, increasing latency and origin load. Mitigated by cache warming strategies and content prefetching.
Consistent Hashing	A distributed hashing technique that minimizes the number of keys that need to be remapped when nodes are added or removed from the system. Keys are mapped to a virtual ring (hash ring), and each node is assigned multiple positions on the ring. When looking up a key, you find the next node clockwise from the key's position.
Content Delivery Network (CDN)	A geographically distributed network of proxy servers (edge nodes) that caches content close to users to reduce latency, bandwidth usage, and origin server load. The CDN sits between users and origin servers, accelerating delivery of static and dynamic content.
Control Plane	The management layer of the CDN responsible for configuration distribution, analytics collection, health monitoring, and coordination between edge nodes. Handles administrative tasks like cache invalidation propagation but does not process user traffic.

Term	Definition
Data Plane	The layer that handles actual user traffic (requests and responses) flowing through edge nodes. This is the performance-critical path where caching, compression, and routing decisions happen in real-time.
Device Pixel Ratio (DPR)	The ratio between physical pixels and CSS pixels on a device's display. High-DPR devices (like Retina displays) require higher resolution images. The CDN can use this value in image optimization to serve appropriately sized images.
DNS TTL	The Time-To-Live value in DNS records that controls how long DNS resolvers cache the response. In geo-routing, short DNS TTLs (e.g., 60 seconds) allow faster failover when edge nodes become unhealthy, at the cost of increased DNS query load.
Edge Compute	The capability to execute lightweight compute functions at edge nodes rather than at the origin. Enables personalization, A/B testing, security checks, and dynamic content generation close to users. Represented by the `EdgeFunction` and `EdgeFunctionRuntime` concepts.
Edge Node/Edge Server/PoP	A geographically distributed server in a CDN that caches and serves content to nearby users. Also called a Point of Presence (PoP). These nodes form the "edge" of the network closest to end users. Each edge node runs the caching logic described in Milestone 1.
Eventual Consistency	The property where updates (like cache invalidations) propagate to all nodes eventually, but not immediately. During the propagation window, some users may see stale content while others see fresh content. This trade-off is often accepted for scalability in distributed systems.
Eventual Invalidation	A lazy invalidation strategy where a banned cache entry is only removed when it is next accessed, not immediately. This reduces the immediate performance impact of sweeping invalidations but means stale content may remain in cache longer.
Flight Data Recorders	Diagnostic headers (like `X-Cache-Status`, `X-Cache-Key`, `X-Node-ID`) that document a request's path through the caching infrastructure. Analogous to aircraft black boxes, they provide visibility for debugging without impacting performance.
GeoIP Database	A database that maps IP addresses to geographic locations (country, region, city, coordinates). Used by the CDN's geo-routing system to direct clients to the nearest edge node. Requires regular updates as IP allocations change.
Geo-Routing	Directing user requests to the nearest optimal edge node based on geographic location. Uses client IP address lookup in a GeoIP database to determine approximate location, then selects the edge node with lowest network latency or geographic distance.
Geo Simulation	Techniques to simulate geographic distribution locally during development and testing, such as using IP address prefixes to represent regions or manually setting location headers. Helps test routing logic without deploying to multiple physical locations.
Graceful Degradation	A resilience strategy of serving stale content, reduced functionality, or cached error pages when upstream services are impaired. For example, serving stale content via `stale-if-error` when the origin server is down maintains availability at the cost of freshness.
Hard Purge	Immediate deletion of a cache entry from storage. The next request for that content will be a cache miss and fetch fresh content from origin. Provides strong consistency but may cause cache miss storms if popular content is purged.
Hash Ring	A virtual circle representing the hash space in consistent hashing, typically using a 32-bit or 64-bit integer range. Nodes are assigned positions on the ring via hash functions, and keys are mapped to the next node clockwise from their hash position.
Haversine Formula	An equation used in calculating great-circle distances between two points on a sphere given their longitudes and latitudes. Used in geo-routing to find the geographically closest edge node to a client's location.
Health Checking	The process of monitoring node availability and performance to ensure reliable routing. Edge nodes periodically report health metrics to the control plane, which removes unhealthy nodes from rotation. Health checks verify connectivity, latency, and resource utilization.
Image Optimization	On-the-fly resizing, format conversion (WebP, AVIF), compression, and quality adjustment of images at the edge based on device capabilities and network conditions. Reduces bandwidth usage and improves page load times. Represented by the `ImageTransformation` and `OptimizedImageCacheKey` structures.
Invalidation Propagation	The process of disseminating an invalidation command (purge, ban, tag purge) to all edge nodes in the CDN. Can use pub/sub messaging, control plane broadcasts, or gossip protocols to achieve eventual consistency across nodes.
Load Testing	Testing system behavior under production-like load to identify performance bottlenecks, capacity limits, and failure modes. For a CDN, load testing simulates many concurrent users requesting various content types to verify the system can handle peak traffic.
Negative Caching	Caching error responses (like 404 Not Found or 503 Service Unavailable) with a short TTL to protect the origin from repeated requests for non-existent or temporarily unavailable resources. Prevents "cache miss storms" for error conditions.
Node Failover	The process of redirecting traffic from a failed or unhealthy edge node to a backup node. When health checks detect a node failure, the routing system (DNS or anycast) updates to direct new clients to the next-nearest healthy node.
Object Storage Integration	Treating cloud object storage services (AWS S3, Google Cloud Storage, Azure Blob Storage) as first-class origins for the CDN. Requires handling authentication, bucket policies, and storage-specific APIs while maintaining caching semantics.
Origin Client	An abstract base class for fetching content from different types of origins (HTTP servers, S3 buckets, etc.). Subclasses like `S3OriginClient` implement origin-specific authentication and request handling while presenting a unified interface to the caching layer.
Origin Routing	Directing requests to different origins based on rules like path prefixes, hostnames, or geographic location. Enables multi-origin setups where different content comes from different backend systems while appearing as a unified CDN.

Term	Definition
Origin Server	The ultimate source of truth for web content, also called the "backend" or "source" server. The CDN fetches content from origin servers when cache misses occur and serves cached copies to users. Origins should implement proper cache control headers.
Origin Shield/Mid-Tier Cache	An intermediate caching layer between edge nodes and the origin server that reduces origin load by acting as a shared cache for multiple edge nodes. The shield collapses duplicate requests and queues others, preventing thundering herd problems at the origin.
Predictive Prefetching	Automatically fetching content before users request it based on access patterns, link analysis, or machine learning predictions. For example, prefetching linked CSS and JavaScript files when a user requests an HTML page.
Propagation Delay	The time it takes for invalidation commands to reach all edge nodes in a distributed CDN. During this window, some users may receive stale content from nodes that haven't yet received the invalidation notice. Systems aim to minimize this delay.
Property-Based Testing	A testing methodology that verifies properties hold for all possible inputs rather than specific examples. For caching, properties might include "cache hit should always return same response as original fetch" or "purge should always remove the entry".
Pub/Sub Broker	A message broker that implements the publish-subscribe pattern, allowing components to broadcast messages to multiple subscribers. The `InMemoryPubSubBroker` handles invalidation propagation within a single process, while production systems use distributed brokers like Redis or Kafka.
Pub/Sub Channel	A named message bus topic for publish-subscribe messaging. In the CDN, channels like `invalidation` or `health` allow edge nodes to subscribe to relevant updates without knowing about other subscribers.
Race Condition	When system behavior depends on the sequence or timing of uncontrollable events. In caching, race conditions can occur when multiple threads simultaneously check cache, miss, and fetch from origin, causing duplicate origin requests or cache corruption.
Range Request	An HTTP request for partial content using the Range header (e.g., `Range: bytes=0-999`). Used for video streaming, large file downloads, and resumable transfers. The CDN must handle range requests efficiently without fetching entire files.
Request Coalescing Map	An in-memory data structure that tracks in-flight requests for deduplication. When multiple clients request the same uncached resource concurrently, the map ensures only one origin fetch occurs, and all clients wait for that single result.
Request Collapsing	Deduplicating concurrent identical requests to the origin. When multiple edge nodes or clients request the same uncached content simultaneously, the shield collapses them into a single origin fetch and shares the response with all requesters.
Request Tracing	Tracking a request through all CDN layers using unique IDs (like `X-Request-ID`) that are propagated across service boundaries. Enables distributed debugging by correlating logs and timing information across edge nodes, shields, and origins.
Soft Purge	Marking a cache entry as stale while continuing to serve it to clients, then triggering background revalidation from the origin. Provides better user experience than hard purge (no cache miss penalty) but may serve stale content briefly.
Stale-if-error	A Cache-Control directive that allows serving stale content when the origin server returns an error (5xx status). This provides graceful degradation during origin outages, maintaining availability at the cost of freshness.
Stale-while-revalidate	A Cache-Control directive that allows serving stale content while asynchronously revalidating it in the background. Users get fast responses from cache while the system ensures content freshness for subsequent requests.
Surrogate Key	A tag (string identifier) assigned to cache entries for group-based invalidation. Multiple resources (like all images in a product catalog) can share a surrogate key, allowing invalidation of all related content with a single purge command.
Test Fixture	Reusable setup and teardown code for tests that creates a known state before tests run and cleans up afterward. For CDN testing, fixtures might start a test origin server, create cache entries, or simulate network conditions.
Thundering Herd Problem	A stampede of concurrent requests overwhelming a resource, typically occurring when cached content expires simultaneously or after a cache invalidation. The sudden spike of cache misses can overload origin servers. Mitigated by request collapsing, staggered TTLs, and cache warming.
Time-to-Live (TTL)	The duration a cached item is considered fresh before it needs revalidation. Determined by cache control headers with hierarchy: `s-maxage` (for shared caches) > `max-age` > `Expires`. The CDN calculates `expires_at = fetched_at + TTL` for each cache entry.
TTL Hierarchy	The precedence order for TTL determination: `s-maxage` (for shared caches like CDNs) takes precedence over `max-age`, which takes precedence over `Expires` header. The CDN uses `s-maxage` when present since it's specifically for intermediary caches.
Vary Header	An HTTP response header that indicates which request headers the response varies on. For example, `Vary: Accept-Encoding` means responses differ based on the client's Accept-Encoding header value, requiring separate cache entries for gzip, br, and identity encodings.
Vary: *	A special value of the Vary header indicating the response varies on all request headers. According to RFC 9111, this means the response should never be stored in a shared cache, as it's essentially uncacheable for different clients.
Virtual Node	Multiple hash positions on a consistent hash ring representing a single physical node. Using multiple virtual nodes per physical node (e.g., 100-200) provides better load distribution and minimizes hotspotting when nodes are added or removed.

Term	Definition
WebSocket Proxying	Extending the CDN to proxy WebSocket connections between clients and origins, maintaining the bidirectional communication channel while providing caching for initial handshake and potential message inspection/transformation.