High-Performance Server with io_uring: Design Document

Overview

This document outlines the design for building a high-performance server leveraging Linux's io_uring interface for asynchronous I/O. It solves the architectural challenge of managing thousands of concurrent I/O operations efficiently, moving beyond the limitations of traditional event-loop models like epoll. The key focus is on mastering core concepts such as submission/completion queues, zero-copy operations, and linked requests to achieve superior throughput for I/O-bound workloads.

> 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 foundational section establishes the core architectural problem that the entire project addresses, providing context for all four milestones.

High-performance servers face a fundamental architectural challenge: efficiently managing thousands of concurrent input/output (I/O) operations without wasting precious CPU cycles waiting for slow devices like disks, networks, or remote services. This I/O bottleneck determines the scalability ceiling for web servers, databases, file servers, and proxy systems. The evolution of Linux I/O models—from blocking system calls to event-driven polling to today's io_uring—represents a continuous quest to reduce overhead and maximize hardware utilization. This document outlines a journey to master **io_uring**, Linux's revolutionary asynchronous I/O interface that fundamentally changes how applications interact with the kernel, offering unprecedented performance for I/O-heavy workloads.

The I/O Bottleneck and The Evolution of Async

Imagine a busy restaurant kitchen. When an order arrives, the traditional approach (**blocking I/O**) is like a chef who stops everything, goes to the pantry to fetch ingredients, waits at the refrigerator, and only returns to the stove when they have everything in hand. The entire kitchen grinds to a halt for each order. This is exactly what happens when a server uses blocking `read()` or `write()` system calls—the entire process (or thread) is suspended, consuming memory and context switch overhead while waiting for data.

The first evolution introduced **non-blocking I/O with polling mechanisms** (`select()` / `poll()`). In our kitchen analogy, this is like having a chef who periodically checks a shared whiteboard (the file descriptor set) to see which orders have ingredients ready. The chef can work on other tasks between checks, but they waste time constantly looking at the whiteboard (CPU cycles polling), and as the restaurant gets busier (thousands of file descriptors), checking the entire whiteboard becomes painfully slow—an **O(n)** operation.

The **epoll** model brought a significant improvement: an intelligent notification system. Now, the kitchen installs a bell that rings only when specific orders have ingredients ready. The chef can focus on cooking until the bell rings, then check exactly which orders are ready. This **O(1)** event notification scales to tens of thousands of concurrent connections and powers modern servers like Nginx and Node.js. However, epoll still has inherent overhead: every I/O operation requires at least two system calls (submit + wait/notify), and data must be copied between kernel and user space buffers, consuming CPU cycles.

The **io_uring** model represents a paradigm shift: it's a **ticket system** with two-sided queues. The kitchen now has two physical rails:

1. **Submission Queue (SQ) Rail**: Chefs place order tickets (operations) on this rail. They can place multiple tickets at once in a batch.
2. **Completion Queue (CQ) Rail**: The kitchen staff (kernel) processes tickets and places completed orders on this pickup rail.

The critical innovation is that both queues are **shared memory rings** mapped between the application and kernel. Submitting an I/O operation no longer requires a system call—the application just writes a ticket (Submission Queue Entry, or SQE) to the shared ring and optionally rings a doorbell (a lightweight system call) to notify the kernel. The kernel processes requests asynchronously and writes completion tickets (Completion Queue Entries, or CQEs) to the other ring. The application harvests completions by reading from this shared memory, again without system calls in the happy path. This architecture enables:

- **True zero-system-call I/O** in optimal cases (submission and completion both handled in userspace)
- **Batched operations** where dozens of requests can be submitted with a single notification
- **Advanced features** like linked operations, zero-copy networking, and kernel-side request chaining

The mental model shift is profound: instead of repeatedly asking the kernel "is my I/O ready?" (poll/epoll), you give the kernel a todo list (SQ) and it gives you a done list (CQ)—all through shared memory with minimal kernel involvement.

Existing Approaches: A Comparison

To understand why io_uring represents a fundamental advance, we must compare it with historical approaches. Each model represents a trade-off between complexity, scalability, and overhead. The table below captures the essential characteristics:

I/O Model	Mental Model	Syscall Overhead per I/O Op	Scalability (Concurrent FDs)	Batching Capability	Kernel-User Copies	Primary Limitation
Synchronous (Blocking)	Chef waits at pantry	1:1 (blocking call per op)	Poor (one thread per FD)	None	One copy per read/write	Thread/process explosion, wasted resources
Non-blocking with `select()` / `poll()`	Chef checks whiteboard periodically	2+ (poll + I/O op)	Limited (~1024 FDs for select)	None (per FD readiness)	One copy per read/write	O(n) polling overhead, thundering herd
Event-driven with `epoll`	Bell rings when order ready	2+ (epoll_wait + I/O op)	Excellent (10k+ FDs)	Limited (ready FDs only)	One copy per read/write	At least two syscalls per I/O, copy overhead
AIO (POSIX/linux)	Async order slips (broken)	2+ (submit + wait)	Good in theory	Yes, but flawed	One copy	Complex, limited file/network support, completion pitfalls
io_uring	Ticket system with shared rails	**0-1** (shared memory + optional doorbell)	Excellent (10k+ FDs)	**Full batch submission**	**Zero-copy possible**	Steeper learning curve, manual buffer management

The comparison reveals three critical dimensions where io_uring excels:

1. **Syscall Overhead Reduction**: Traditional models require at least one system call to initiate I/O and another to wait for completion (or check readiness). io_uring allows both submission and completion to happen entirely in userspace via shared memory rings. The `io_uring_enter()` syscall becomes optional for submission (only needed when the SQ is full or explicit kernel notification is desired) and for completion harvesting (only when waiting for events).
2. **Batching Efficiency**: While epoll can return multiple ready file descriptors in one call, each still requires separate I/O system calls. io_uring allows submitting dozens—even hundreds—of I/O operations with a **single** `io_uring_enter()` call. This dramatically reduces context switch overhead for I/O-intensive workloads like database queries or file servers handling concurrent requests.
3. **Zero-Copy Potential**: Traditional I/O requires data to be copied between kernel buffers and application buffers. io_uring supports **fixed buffers** (registered once, reused) and **zero-copy operations** like `IORING_OP_SEND_ZC` where network data can be sent directly from application buffers without intermediate copies. This eliminates a major CPU cost for high-throughput networking.

> **Design Insight:** The fundamental architectural shift in io_uring is moving from a **solicited notification model** (application asks kernel about readiness) to an **unsolicited completion model** (kernel tells application when operations finish via shared memory). This inversion of control reduces latency and syscall overhead, but requires careful buffer lifecycle management since operations proceed asynchronously.

Why This Matters for Server Performance: Modern servers are often I/O-bound, not CPU-bound. The time spent waiting for disk seeks, network packets, or remote API responses dominates execution time. Each system call—while fast in absolute terms—adds overhead: context switches between user and kernel mode, CPU pipeline flushes, and TLB misses. By minimizing these, io_uring allows servers to achieve near-linear scaling with core count for I/O workloads, efficiently saturating high-speed storage (NVMe SSDs) and network interfaces (100GbE). For educational purposes, mastering io_uring provides deep insight into modern high-performance systems architecture and prepares developers for the next generation of Linux server infrastructure.

The Learning Journey Ahead: This project progresses through four milestones that build competency incrementally:

1. **Basic SQ/CQ Operations**: Understanding the shared ring mechanics—the foundation of all io_uring usage.
2. **File I/O Server**: Applying io_uring to disk operations with buffer management.

3. **Network Server**: Building a full TCP server with connection lifecycle management.
4. **Advanced Features**: Implementing zero-copy, linked operations, and benchmarking against epoll.

Each milestone addresses specific pitfalls and design decisions that distinguish proficient io_uring usage from naive implementation. The following sections detail these decisions with architecture decision records, data models, and implementation guidance.

Goals and Non-Goals

> **Milestone(s):** This section defines the scope for all four milestones, establishing the educational boundaries of the project.

This section defines the precise scope of our educational journey with io_uring. Building a production-ready server involves countless considerations—security, protocol compliance, monitoring, and more. To maintain focus on mastering the novel architecture and performance characteristics of io_uring itself, we deliberately constrain what we will build. The goals are organized as concrete, verifiable outcomes for each milestone, while the non-goals explicitly exclude complexities that, while important in real systems, would distract from the core learning objectives.

Goals (What we must do)

The primary educational goals are structured as four sequential milestones, each building upon the last. Success is measured by achieving the specific acceptance criteria listed below, which are derived directly from the project requirements.

Milestone 1: Basic SQ/CQ Operations This milestone establishes foundational competence with the io_uring machinery.

1. **Initialize an io_uring instance:** Successfully call `io_uring_setup` and map the shared submission and completion queue rings into the application's address space using `mmap`.
2. **Submit basic operations:** Prepare and submit `IORING_OP_READ` and `IORING_OP_WRITE` Submission Queue Entries (SQEs) for file descriptors.
3. **Harvest and process completions:** Implement a loop that retrieves Completion Queue Entries (CQEs) from the completion queue, correctly interprets the `res` field (handling both success and error codes), and correlates them to their original request via `user_data`.
4. **Demonstrate batched syscalls:** Submit multiple prepared SQEs (e.g., 4-8) to the kernel with a single `io_uring_enter` syscall, verifying that all corresponding completions are later harvested.

Milestone 2: File I/O Server This milestone applies io_uring to a concrete problem: high-throughput file serving.

1. **Serve file reads asynchronously:** Build a server that responds to file read requests by submitting `IORING_OP_READ` SQEs with appropriate file descriptors, offsets, and buffer pointers, instead of using synchronous `pread` syscalls.
2. **Manage concurrent operations:** Handle multiple overlapping file read requests without blocking, using a pool of buffers and tracking the state of each in-flight operation.
3. **Implement fixed buffer registration:** Register a set of application buffers with the kernel using `io_uring_register` with the `IORING_REGISTER_BUFFERS` opcode, and subsequently submit read operations that reference these buffers by index (using `IORING_OP_READ_FIXED`), eliminating per-operation kernel mapping overhead.
4. **Benchmark performance:** Produce a benchmark (e.g., using `fio` or a custom client) that demonstrates a measurable throughput improvement for parallel file reads compared to a synchronous or `epoll`-based server under the same workload.

Milestone 3: Network Server This milestone extends io_uring to network I/O, building a fully async TCP server.

1. **Accept connections asynchronously:** Use `IORING_OP_ACCEPT` to asynchronously accept incoming TCP connections on a listening socket, without blocking the main thread.
2. **Implement operation chaining:** Design a connection lifecycle where an accept completion automatically triggers the submission of a read SQE for that new socket, and a read completion triggers a write SQE to echo back the data, forming a logical chain of events.
3. **Utilize multishot operations:** Where supported by the kernel, use `IORING_ACCEPT_MULTISHOT` to automatically re-arm the accept operation, and explore `IORING_RECV_MULTISHOT` for reads, reducing submission queue pressure.
4. **Achieve scalability:** Demonstrate that the server can maintain stable performance (in requests per second and latency) while handling thousands of concurrent idle and active connections, managed entirely through io_uring without an `epoll` fallback.

Milestone 4: Zero-copy, Linked Ops, and Benchmarks This milestone explores advanced optimizations and provides definitive performance comparisons.

1. **Implement zero-copy network send:** Use `IORING_OP_SEND_ZC` to transmit network data without the kernel copying the payload from user space to kernel buffers. Correctly manage the lifecycle of the sent buffer until the kernel signals completion via a dedicated CQE.

2. **Create linked SQE chains:** Use the `IOSQE_IO_LINK` flag to create atomic sequences of operations. For example, link a read, a processing step (potentially a no-op or simple transformation), and a write SQE so they execute in order and fail as a unit.
3. **Apply ordering guarantees:** Use the `IOSQE_IO_DRAIN` flag to enforce that a specific SQE does not start executing until all previously submitted SQEs have completed, understanding the performance trade-off involved.
4. **Execute comprehensive benchmarks:** Design and run a benchmark suite that compares the io_uring server against a functionally equivalent server built using the traditional `epoll` event loop. Test across varied workloads: small/large file I/O, high-connection concurrency, and mixed request patterns. The benchmark must show clear scenarios where io_uring's advantages manifest.

Non-Goals (What we won't do)

To maintain a sharp educational focus on io_uring's core mechanics, we explicitly exclude the following production-grade features and complexities. This is not a judgment on their importance, but a recognition that adding them would dilute the primary learning objectives.

Non-Goal Category	Specific Examples	Rationale for Exclusion
Protocol Implementation	Implementing HTTP/1.1, HTTP/2, WebSocket, or gRPC parsing.	Protocol parsing is a substantial topic orthogonal to async I/O mechanics. Our servers will use simple, custom request/response formats (e.g., "READ /path/to/file") to keep I/O logic clear.
Security & Encryption	Adding TLS/SSL termination (via OpenSSL or RustTLS).	TLS introduces complex buffering, handshake state management, and potential blocking operations that would obscure the io_uring flow.
Advanced Architecture	Multi-threading with shared io_uring instances (`IORING_SETUP_SQPOLL`, `IORING_SETUP_COOP_TASKRUN`), multi-process load balancing.	Managing concurrency *within* the ring is a complex, advanced topic. Our design uses a single thread per ring, which is sufficient to demonstrate io_uring's performance benefits for I/O-bound work.
Production Operations	Dynamic configuration reloading, comprehensive logging (beyond debug prints), metrics exposition (Prometheus), health checks, graceful shutdown with drain phases.	These are essential for deployable services but are "busy work" that doesn't advance understanding of io_uring's submission/completion model.
Resource Management	Sophisticated connection pooling, adaptive buffer sizing, memory-mapped file I/O (`IORING_OP_MMAP`).	We will use simple, fixed-size pools. Advanced resource optimization is a follow-on topic once the fundamentals are solid.
Filesystem & Storage	Supporting directory listings, file writes/uploads, file metadata operations, or database integration.	Our file server is read-only to simplify the example. The principles learned apply equally to write operations (`IORING_OP_WRITE`, `IORING_OP_FSYNC`).
Network Topology	IPv6 support, UDP server implementation, socket option tuning (`SO_REUSEPORT`), or kernel bypass techniques (e.g., DPDK, XDP).	These are valuable extensions but would shift focus from mastering basic io_uring networking to network stack intricacies.
Cross-Platform Compatibility	Making the code run on non-Linux systems (e.g., via fallbacks to `kqueue` or `IOCP`).	io_uring is a Linux-specific interface. The project's goal is deep mastery of this specific technology.

> **Key Design Insight:** The most common failure mode in educational projects is **scope creep**. By ruthlessly defining these non-goals, we create guardrails that keep the learner's effort and attention directed at the novel architectural concepts of io_uring: the shared ring buffers, the separation of submission and completion, and the efficient kernel-user space communication. Mastery of these fundamentals is the prerequisite for later, more complex integrations.

Common Pitfalls: Scope Management

Even with clear non-goals, learners often inadvertently expand scope. Below are common missteps and how to avoid them.

⚠️ **Pitfall: Implementing a "real" protocol like HTTP**

- **What happens:** A learner decides their file server should respond with proper HTTP headers (`HTTP/1.1 200 OK`, `Content-Length`, etc.) to be testable with a browser or `curl`.

- **Why it's problematic:** Suddenly, the project is no longer about managing I/O completions; it's about parsing request lines, handling headers, URL encoding, and status codes. This can easily double the codebase with logic that doesn't advance the core learning goal.
- **How to avoid:** Stick to a trivial application-layer protocol. For example, a request is a single line: `GET /path/to/file\n`. The response is the raw file data followed by a `\n`. This is trivial to parse and keeps all complexity in the I/O layer.

⚠️ Pitfall: Premature multi-threading

- **What happens:** Concerned about "using all CPU cores," a learner attempts to create multiple `io_uring` instances or share one instance across threads early in the project.
- **Why it's problematic:** This introduces immense complexity: thread synchronization for ring access, managing `user_data` collisions, and handling `IORING_ENTER_GETEVENTS` across threads. The single-threaded io_uring model is already capable of saturating disk or network I/O, which is the bottleneck.
- **How to avoid:** Complete all milestones with a single-threaded, single-ring design first. Only consider multi-threading (Milestone 4+) after the core flow is flawless and you are specifically exploring CPU-bound processing alongside I/O.

⚠️ Pitfall: Building extensive CLI or configuration frameworks

- **What happens:** Significant time is invested in parsing command-line arguments, reading configuration files (YAML/TOML), or building admin consoles.
- **Why it's problematic:** This is "meta-work" that yields no insight into async I/O. It delays engagement with the actual subject matter.
- **How to avoid:** Use hardcoded parameters (port numbers, file directories) or simple environment variables. The focus should be on the event loop, not on configuration plumbing.

Implementation Guidance

This section provides concrete starting points for organizing the codebase and tracking progress against the defined goals.

A. Technology Recommendations Table

Component	Simple Option (Recommended for Learning)	Advanced Option (For Further Exploration)
Language & Toolchain	C11 with GCC/Clang, `make` build system.	Rust with the `tokio-uring` or `ringbahn` crate; Zig with its built-in `io_uring` support.
Kernel Version	Linux 5.10+ (supports core features).	Linux 6.0+ (for latest features like `IORING_OP_MSG_RING`).
Debugging Tools	`strace -e io_uring_enter`, `perf` for syscall counts, custom debug logging.	`bpftrace` / `BCC` for tracing kernel-side io_uring events.
Benchmarking Tools	Custom load-test client, `fio` (for file I/O), `wrk` / `ab` (for HTTP-like loads).	`perf` for CPU utilization and cycle analysis, kernel `ftrace` for I/O latency histograms.

B. Recommended File/Module Structure

Organize the code from the start to support all four milestones cleanly. This structure separates concerns and makes it easy to focus on one component at a time.

```
io_uring-server-project/
├── Makefile                    # Build definitions
├── src/
│   ├── core/                   # Milestone 1: Core engine (shared by all servers)
│   │   ├── uring.c             # Core setup/teardown, submission/completion loops
│   │   ├── uring.h             # Core types and function declarations
│   │   └── buffer_pool.c       # Milestone 2: Buffer management logic
│   ├── file_server/            # Milestone 2: File I/O server
│   │   ├── main.c              # Entry point for file server
│   │   ├── handler.c           # Logic for processing file read requests
│   │   └── handler.h
│   ├── network_server/         # Milestone 3 & 4: Network server
│   │   ├── main.c              # Entry point for network server
│   │   ├── connection.c        # Connection state machine and management
│   │   ├── connection.h
│   │   └── protocols/
│   │       └── echo.c          # Simple echo protocol handler
│   ├── benchmarks/             # Milestone 4: Performance tests
│   │   ├── benchmark.c         # Generic benchmarking harness
│   │   ├── file_bench.c        # File server benchmark
│   │   ├── network_bench.c     # Network server benchmark
│   │   └── epoll_baseline.c    # Epoll-based server for comparison
│   └── utils/
│       ├── logging.c           # Simple debug logging macro/function
│       └── utils.c             # Common helpers (e.g., error handling)
└── tools/                      # Test clients, scripts
    ├── test_client.c           # Generic test client
    └── run_benchmark.sh        # Script to automate benchmark runs
```

C. Infrastructure Starter Code: Logging and Error Handling

To avoid rewriting basic utilities, here is a complete, reusable logging module. Place this in `src/utils/logging.c` and `src/utils/logging.h`.

File: `src/utils/logging.h`

```c
#ifndef LOGGING_H
#define LOGGING_H

#include <stdio.h>
#include <errno.h>
#include <string.h>

// Log levels
#define LOG_FATAL   0
#define LOG_ERROR   1
#define LOG_WARN    2
#define LOG_INFO    3
#define LOG_DEBUG   4

#ifndef LOG_LEVEL
#define LOG_LEVEL LOG_INFO  // Default log level
#endif

// Helper macro to get errno string
#define CLEAN_ERRNO (errno == 0 ? "None" : strerror(errno))

// Log macros that include file and line
#define LOG(level, fmt, ...) do { \
    if (level <= LOG_LEVEL) { \
        fprintf(stderr, "[%s:%d] " fmt "\n", __FILE__, __LINE__, ##__VA_ARGS__); \
    } \
    if (level == LOG_FATAL) exit(1); \
} while(0)

#define FATAL(fmt, ...) LOG(LOG_FATAL, fmt, ##__VA_ARGS__)
#define ERROR(fmt, ...) LOG(LOG_ERROR, fmt, ##__VA_ARGS__)
#define WARN(fmt, ...)  LOG(LOG_WARN, fmt, ##__VA_ARGS__)
#define INFO(fmt, ...)  LOG(LOG_INFO, fmt, ##__VA_ARGS__)
#define DEBUG(fmt, ...) LOG(LOG_DEBUG, fmt, ##__VA_ARGS__)

// Log with errno context
#define ERRNO_ERROR(fmt, ...) ERROR(fmt ": (errno: %s)", ##__VA_ARGS__, CLEAN_ERRNO)
#define ERRNO_FATAL(fmt, ...) FATAL(fmt ": (errno: %s)", ##__VA_ARGS__, CLEAN_ERRNO)

#endif // LOGGING_H
```

File: `src/utils/logging.c`

```c
// This file exists to compile the logging module if needed.

// The header is primarily macro-based, so this may be empty.

#include "logging.h"
```

D. Core Logic Skeleton: Main Event Loop

The central pattern for all servers will be a main event loop. Below is a skeleton for the core engine's loop, to be placed in `src/core/uring.c`. This provides the structural TODOs that map directly to the procedural steps learned in Milestone 1.

```c
#include "uring.h"

#include "../utils/logging.h"


// TODO: Define struct io_uring_ctx to hold ring state, buffer pools, etc.


int uring_loop_run(struct io_uring_ctx *ctx) {

    int ret;

    unsigned head;

    unsigned count = 0;


    INFO("Starting io_uring event loop");


    while (1) {

        // TODO 1: Check for any external events or signals (e.g., to stop the loop)


        // TODO 2: Prepare Submission Queue Entries (SQEs)

        // - Check for new network connections to accept

        // - Check for new file requests to read

        // - Check for pending data to write on sockets

        // For each operation, prepare an SQE via io_uring_get_sqe()

        // Fill opcode, fd, addr, len, offset, user_data, flags


        // TODO 3: Submit all prepared SQEs to the kernel

        // Call io_uring_submit() (or io_uring_enter with appropriate flags)

        // This is where batching happens: multiple SQEs go with one syscall.


        // TODO 4: Harvest Completion Queue Entries (CQEs)

        // Use io_uring_peek_batch_cqe or io_uring_wait_cqe to get CQEs

        // Process each CQE:

        //   - Read cqe->res (negative is error, positive is bytes transferred)

        //   - Use cqe->user_data to find the associated operation context

        //   - Handle the completion based on the original opcode

        //     * For READ: data is now in buffer, send it or process it

        //     * For ACCEPT: new socket fd, submit a READ for it

        //     * For WRITE: data sent, can clean up buffer

        //   - Advance the CQ ring tail (io_uring_cqe_seen)


        // TODO 5: Perform any housekeeping

        // - Clean up closed connections

        // - Return buffers to the pool

        // - Resubmit multishot operations if they stopped
```

```
    }

    INFO("Event loop exiting");
    return 0;
}
```

E. Language-Specific Hints: C

1. **Compiler Flags:** Use `-std=c11 -D_GNU_SOURCE -O2 -g` to enable the necessary GNU extensions (like `io_uring_setup`), modern C features, optimizations, and debug symbols.
2. **Linking:** No special libraries are required for io_uring syscalls; they are provided by the kernel. Use `-pthread` if you eventually add threading.
3. **Kernel Headers:** Include `<linux/io_uring.h>` for structure and constant definitions. The `liburing` library is not used in this project (to understand the raw syscalls), but it's an excellent production wrapper.
4. **Memory Barriers:** When directly manipulating the ring buffer head/tail indexes (as opposed to using helper functions from `liburing`), you must use compiler barriers (`asm volatile("" ::: "memory")`) or atomic operations to prevent reordering. This is critical for correctness.

F. Milestone Checkpoint: Overall Progress Tracker

After completing the implementation for each milestone, run these verification steps before proceeding to the next.

Milestone	Verification Command & Expected Output	What Success Looks Like
1	`./build/milestone1_test` (a custom test that writes/reads a file)	Program runs without hanging or crashing. Output shows SQEs submitted in batches and CQEs harvested with correct `res` values. `strace` shows few `io_uring_enter` calls relative to I/O ops.
2	`./src/file_server/file_server &` then `./tools/test_client file read /large/file.bin`	Server handles concurrent client requests. Kernel logs (with `dmesg`) show registered buffers. Benchmark shows higher IOPS than a synchronous version under `fio --numjobs=16`.
3	`./src/network_server/network_server &` then `./tools/load_test --connections=5000`	Server maintains all connections without OOM or file descriptor leaks. Accept and read completions are processed continuously. No `EMFILE` (too many open files) errors.
4	`./src/benchmarks/run_all.sh` (script running the benchmark suite)	Benchmarks complete, generating a report. The io_uring server shows superior throughput in file I/O and high-connection scenarios. Zero-copy sends are verified by reduced CPU% in `perf stat`.

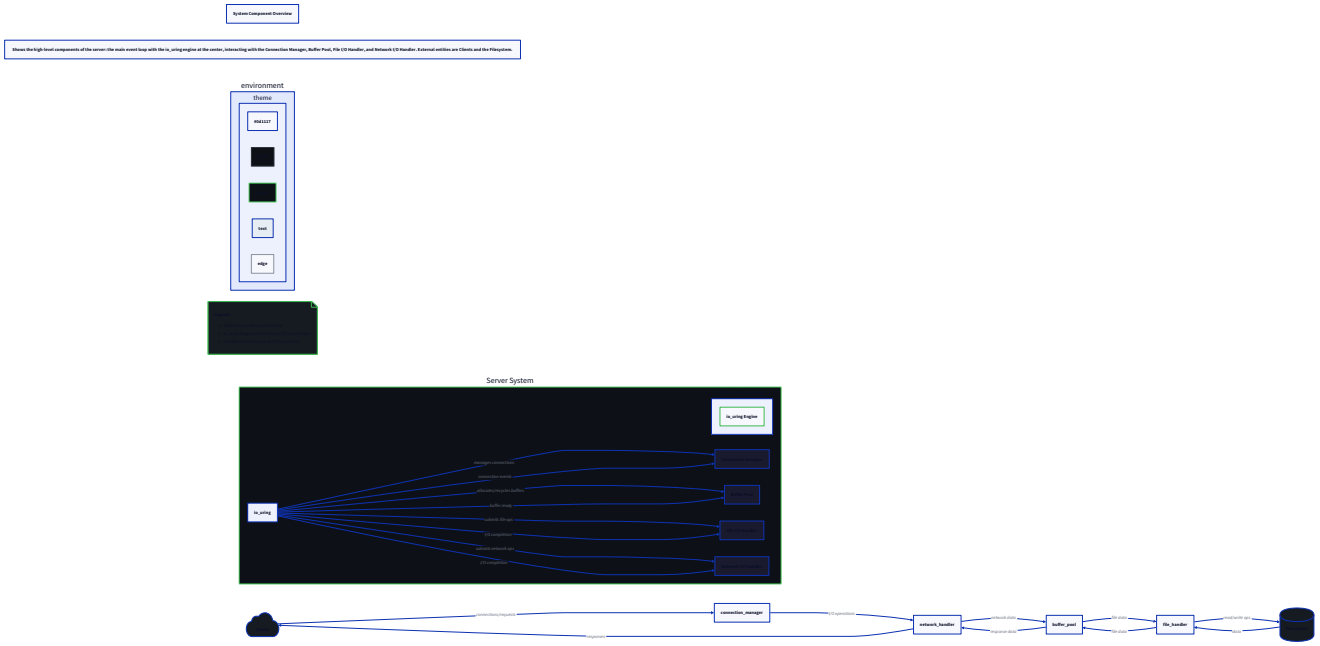
G. Debugging Tips: Early-Stage Issues

Symptom	Likely Cause	How to Diagnose	Fix
Program crashes on first `io_uring_setup`	Kernel too old (< 5.1).	Run `uname -r`. Check `dmesg | grep io_uring`.	Upgrade kernel or use a VM with a supported kernel (5.10+ recommended).
`io_uring_enter` returns `-1` with `errno=EINVAL`	Invalid parameters (e.g., flags, ring fd).	Check all parameters passed to `io_uring_enter`. Ensure the `fd` is from `io_uring_setup`.	Consult kernel docs for valid flag combinations for your kernel version.
Completions never arrive	SQEs not properly submitted; CQ tail not advanced; `user_data` mismatch.	Use a debugger to inspect the SQ ring state (head, tail, array). Check if `io_uring_enter` is being called with the correct `to_submit` count. Add logging to track SQE submission.	Ensure `io_uring_enter` is called *after* filling SQEs. Verify the CQ tail is advanced with `io_uring_cqe_seen`.
High CPU usage even at idle	Busy-waiting on completions (e.g., looping on `io_uring_peek_cqe` without pause).	Use `perf top` to see if the process is spinning in userspace.	Use `io_uring_wait_cqe` or `io_uring_enter` with `IORING_ENTER_GETEVENTS` to block the thread until completions are available.

> **Milestone(s):** This section provides the architectural blueprint that underpins all four milestones, defining the core components and their interactions for the final integrated server.

High-Level Architecture

At the highest level, our server follows a **single-threaded event-driven architecture** centered around an `io_uring` instance that serves as the unified I/O engine. Unlike traditional servers that might separate network I/O (using `epoll`) from file I/O (using thread pools or blocking calls), this design channels *all* I/O operations—network accepts, socket reads/writes, and file reads—through a single submission/completion queue pair. This centralized approach eliminates context switches between different I/O subsystems and enables true asynchronous operation for all I/O types.



Component Overview

The architecture decomposes into three main logical components that collaborate through the `io_uring` engine:

1. The io_uring Engine Core

Purpose: Manages the Linux `io_uring` instance itself—the central "engine room" where all I/O operations are submitted and their completions harvested. This component abstracts the low-level kernel interface (`io_uring_setup`, `io_uring_enter`, ring memory management) and provides a clean API for the rest of the system to submit operations and process results.

Key Responsibilities:

- **Ring Initialization & Teardown:** Creating the `io_uring` instance via `io_uring_setup`, memory-mapping the shared Submission Queue (SQ) and Completion Queue (CQ) rings, and cleaning up resources on shutdown.
- **SQE Lifecycle Management:** Allocating free SQEs from the submission queue (`io_uring_get_sqe`), populating them with operation parameters (opcode, fd, buffer, offset), and submitting batches to the kernel (`io_uring_submit` / `io_uring_enter`).
- **CQE Processing:** Harvesting completed entries from the completion queue (`io_uring_wait_cqe`, `io_uring_peek_batch_cqe`), marking them as consumed (`io_uring_cqe_seen`), and routing them to appropriate handlers based on the embedded `user_data` token.
- **Kernel Interaction Coordination:** Deciding when to call `io_uring_enter`—whether to submit immediately, wait for completions, or both—based on the system's load and batching strategy.

Mental Model: Think of this as the **central dispatch and control tower** at a major airport. All flight requests (I/O operations) are submitted here as flight plans (SQEs). The control tower sends these plans to the runway (kernel) for execution. When flights complete (operations finish), the tower receives landing confirmations (CQEs) and notifies the appropriate gates (connection handlers) that their passengers (data) have arrived. The tower manages the entire flow without ever leaving its command center.

2. Connection & Session Manager

Purpose: Manages the lifecycle of client TCP connections, maintaining state for each connection and orchestrating the sequence of I/O operations needed to serve requests. This component translates high-level server logic (accept connections, read requests, send responses) into specific `io_uring` operations.

Key Responsibilities:

- **Connection Acceptance:** Configuring and submitting `IORING_OP_ACCEPT` operations (potentially with `IORING_ACCEPT_MULTISHOT`) to asynchronously accept incoming TCP connections without blocking the main thread.
- **State Tracking:** Maintaining a `struct connection_state` for each active connection, tracking its current phase (accepting, reading, writing, closing), associated file descriptors, pending buffers, and request context.
- **Operation Sequencing:** Determining which I/O operation should come next for a given connection—for example, after an accept completes, submitting a read; after a read completes, perhaps submitting a write response. This may involve creating chains of linked SQEs for atomic request/response cycles.
- **Connection Cleanup:** Properly closing sockets and releasing associated resources when connections terminate, including canceling any in-flight operations for that connection.

Mental Model: This component acts as the **concierge and waitstaff team** in a hotel restaurant. The concierge (accept handler) continuously greets new guests and assigns them to tables (connections). Waitstaff (read/write handlers) take orders (read requests) from assigned tables, relay them to the kitchen (file I/O subsystem), and deliver meals (write responses) back. Each waiter remembers which table they're serving and what stage of the meal they're at, ensuring smooth service from seating to payment.

3. Buffer Manager

Purpose: Manages the allocation, registration, and lifecycle of data buffers used for I/O operations. This critical component ensures buffers remain valid while kernel I/O is in flight and enables performance optimizations like fixed buffer registration and zero-copy transfers.

Key Responsibilities:

- **Buffer Pool Management:** Maintaining a pool of pre-allocated buffers (e.g., using `malloc` or `mmap`) that can be assigned to incoming I/O operations, avoiding per-request allocation overhead.
- **Fixed Buffer Registration:** Registering buffers with the kernel via `io_uring_register` with `IORING_REGISTER_BUFFERS`, allowing subsequent SQEs to reference buffers by index rather than pointer, eliminating kernel mapping overhead per operation.
- **Lifecycle Coordination:** Ensuring buffers remain "pinned" (not reused or freed) from the moment they're attached to an SQE until the corresponding CQE is harvested and processed. This is especially critical for zero-copy operations where the kernel retains buffer references beyond completion notification.
- **Zero-Copy Buffer Handling:** Special handling for buffers used with `IORING_OP_SEND_ZC`, where the kernel takes ownership and returns them via a separate notification mechanism rather than standard CQE.

Mental Model: Imagine this as a **library book checkout system**. The library (buffer manager) maintains shelves of books (buffers). When a patron (I/O operation) requests a book, the librarian checks it out—the book cannot be given to another patron until it's returned. For special exhibitions (zero-copy), the library might lend a book directly to a museum (kernel), which returns it via a special courier (notification CQE) rather than through normal returns. The librarian tracks all loans to ensure no book is missing or double-lent.

Component Interaction Flow:

1. The **Connection Manager** determines it needs to read data from a socket.
2. It requests a buffer from the **Buffer Manager**, which provides a registered buffer from its pool.
3. The Connection Manager calls the **io_uring Engine** to submit an `IORING_OP_READ` SQE, attaching the buffer and a `user_data` token identifying the connection.
4. The Engine submits the SQE batch to the kernel.
5. When the read completes, the Engine harvests the CQE and uses the `user_data` to route it back to the Connection Manager.
6. The Connection Manager processes the data, then returns the buffer to the Buffer Manager for reuse.

Recommended File/Module Structure

A well-organized codebase is essential for managing the complexity of an `io_uring` server. Below is the recommended directory and file structure for the C implementation, designed to separate concerns and facilitate incremental development across milestones.

```
io_uring-server/
├── CMakeLists.txt                    # Build configuration (or Makefile)
├── include/                          # Public header files
│   └── iouring_server.h             # Main API declarations
├── src/                              # Source code root
│   ├── core/                         # Milestone 1: io_uring Engine Core
│   │   ├── iouring_engine.c          # Ring setup/teardown, SQE/CQE handling
│   │   ├── iouring_engine.h          # Engine API (internal)
│   │   ├── event_loop.c              # Main event loop implementation
│   │   └── event_loop.h              # Event loop interface
│   ├── buffer/                       # Milestone 2 & 4: Buffer Manager
│   │   ├── buffer_pool.c             # Fixed buffer registration and pool management
│   │   └── buffer_pool.h             # Buffer pool API
│   ├── fileio/                       # Milestone 2: Asynchronous File I/O
│   │   ├── file_server.c             # File request handling logic
│   │   ├── file_server.h             # File server API
│   │   └── file_ops.c                # Async read/write operations for files
│   ├── network/                      # Milestone 3 & 4: Network Server
│   │   ├── connection.c              # Connection state management
│   │   ├── connection.h              # Connection state structures
│   │   ├── network_server.c          # TCP server setup and accept handling
│   │   ├── network_server.h          # Network server API
│   │   ├── socket_ops.c              # Async accept/read/write for sockets
│   │   └── protocols/                # Optional: simple protocol handlers
│   │       └── echo.c                # Example: echo protocol
│   ├── advanced/                     # Milestone 4: Advanced Features
│   │   ├── zero_copy.c               # Zero-copy send implementation
│   │   ├── linked_ops.c              # Linked SQE chain construction
│   │   └── ordering.c                # IO_DRAIN and ordering control
│   ├── benchmarks/                   # Milestone 4: Performance testing
│   │   ├── benchmark.c               # Benchmark orchestration
│   │   ├── file_bench.c              # File I/O throughput tests
│   │   ├── network_bench.c           # Connection scalability tests
│   │   └── epoll_comparison.c        # io_uring vs epoll comparison
│   ├── utils/                        # Shared utilities
│   │   ├── logging.c                 # Logging facility
│   │   ├── errors.c                  # Error handling helpers
│   │   └── timers.c                  # Simple timer utilities
│   └── main.c                        # Program entry point, component wiring
└── tests/                            # Unit and integration tests
    ├── test_core.c                   # Tests for engine core
    ├── test_buffer.c                 # Tests for buffer pool
    ├── test_fileio.c                 # Tests for file I/O
    └── test_network.c                # Tests for network operations
```

Module Dependencies and Build Progression:

- **Milestone 1:** Focus on `src/core/` and `src/utils/`. The `main.c` creates the engine and runs the basic event loop.
- **Milestone 2:** Add `src/buffer/` and `src/fileio/`. The file server uses the engine and buffer pool.
- **Milestone 3:** Add `src/network/`. The network server integrates connection management with the existing engine.
- **Milestone 4:** Add `src/advanced/` and `src/benchmarks/`. Advanced features build upon all previous components.

Key Header File Relationships:

```
main.c → includes iouring_engine.h, buffer_pool.h, file_server.h, network_server.h
iouring_engine.h → includes <liburing.h> or kernel uapi headers
buffer_pool.h → self-contained, used by file_server.c and network_server.c
file_server.h → includes buffer_pool.h, iouring_engine.h
network_server.h → includes connection.h, buffer_pool.h, iouring_engine.h
connection.h → includes buffer_pool.h (for buffer references)
```
PLAINTEXT

This structure ensures clean separation: the core engine knows nothing about files or networks, the file and network components don't know about each other, and all share the buffer manager. The `main.c` file acts as the composition root that wires everything together based on compile-time configuration or command-line arguments.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option (Learning Focus)	Advanced Option (Further Exploration)
Build System	Single `Makefile` with explicit rules	CMake with `find_package(liburing)` for portability
io_uring Library	Direct syscalls (`syscall(SYS_io_uring_*)`) for maximum transparency	`liburing` helper library for convenience and stability
Buffer Management	Pre-allocated array of fixed-size buffers	Buddy allocator or slab allocator for variable-sized buffers
Connection Tracking	Array of `struct connection_state` with FD as index	Hash table keyed by FD for sparse connection sets
Benchmarking	Custom microbenchmarks with `clock_gettime()`	Integration with `perf` and `bpftrace` for kernel-level analysis

B. Recommended Starter Files

File: `src/core/iouring_engine.h` (Complete starter)

```c
#ifndef IOURING_ENGINE_H
#define IOURING_ENGINE_H

#include <stdbool.h>
#include <stdint.h>

// Forward declaration - implementation details hidden
struct io_uring_engine;
struct io_uring_sqe;
struct io_uring_cqe;

// Callback type for completion handlers
typedef void (*io_completion_cb)(struct io_uring_engine *eng,
                                 struct io_uring_cqe *cqe,
                                 void *user_ctx);

// Engine configuration
struct engine_config {
    unsigned int sq_entries;      // Submission queue size
    unsigned int cq_entries;      // Completion queue size (often 2x SQ)
    unsigned int flags;           // io_uring_setup flags
    bool enable_sqpoll;           // Enable submission queue polling (kernel thread)
    unsigned int sqpoll_cpu;      // CPU to pin sqpoll thread to
};

// Engine statistics
struct engine_stats {
    uint64_t sqe_submitted;
    uint64_t cqe_processed;
    uint64_t io_enter_calls;
    uint64_t batches_submitted;
};

// Engine API
struct io_uring_engine *engine_create(const struct engine_config *cfg);
void engine_destroy(struct io_uring_engine *eng);

// Get a free SQE for configuration
struct io_uring_sqe *engine_get_sqe(struct io_uring_engine *eng);

// Submit prepared SQEs (non-blocking)
```

```c
int engine_submit(struct io_uring_engine *eng);

// Submit and wait for at least 'min_complete' completions
int engine_submit_and_wait(struct io_uring_engine *eng, unsigned int min_complete);

// Process available completions (non-blocking)
int engine_process_completions(struct io_uring_engine *eng);

// Block until completions are available, then process them
int engine_wait_and_process(struct io_uring_engine *eng);

// Get engine statistics
void engine_get_stats(struct io_uring_engine *eng, struct engine_stats *out);

// Get the underlying io_uring instance (for advanced use)
struct io_uring *engine_get_ring(struct io_uring_engine *eng);

#endif // IOURING_ENGINE_H
```

File: `src/core/event_loop.h` (Complete starter)

```c
#ifndef EVENT_LOOP_H

#define EVENT_LOOP_H


#include <stdbool.h>


struct event_loop;

struct io_uring_engine;


// Event loop callbacks

struct event_callbacks {

    // Called before entering the main loop (setup)

    int (*on_setup)(struct event_loop *loop, void *user_data);



    // Called on each iteration before checking for completions

    int (*on_pre_submit)(struct event_loop *loop, void *user_data);



    // Called when the loop should stop (return true to stop)

    bool (*should_stop)(struct event_loop *loop, void *user_data);



    // Called after loop exits (cleanup)

    void (*on_cleanup)(struct event_loop *loop, void *user_data);

};


// Event loop configuration

struct loop_config {

    int max_batch_size;           // Max SQEs to submit per io_uring_enter

    int min_complete;             // Min CQEs to wait for (0 = don't wait)

    unsigned int flags;           // IORING_ENTER_* flags

    bool busy_wait;               // Busy-wait for completions (high CPU, low latency)

    unsigned int busy_wait_usec;  // Microseconds to busy-wait before falling back to syscall

};


// Create and run event loop

struct event_loop *event_loop_create(struct io_uring_engine *eng,

                                     const struct loop_config *config,

                                     const struct event_callbacks *cbs,

                                     void *user_data);


int event_loop_run(struct event_loop *loop);

void event_loop_destroy(struct event_loop *loop);
```

```c
// Signal the event loop to stop (thread-safe if needed)
void event_loop_request_stop(struct event_loop *loop);

#endif // EVENT_LOOP_H
```

File: `src/core/event_loop.c` (Skeleton with TODOs)

```c
#include "event_loop.h"
#include "iouring_engine.h"
#include "logging.h"
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

struct event_loop {
    struct io_uring_engine *engine;
    struct loop_config config;
    struct event_callbacks cbs;
    void *user_data;
    volatile bool stop_requested;
    // TODO: Add any additional state needed (timers, signal handlers, etc.)
};

struct event_loop *event_loop_create(struct io_uring_engine *eng,
                                     const struct loop_config *config,
                                     const struct event_callbacks *cbs,
                                     void *user_data) {
    struct event_loop *loop = malloc(sizeof(*loop));
    if (!loop) return NULL;

    memset(loop, 0, sizeof(*loop));
    loop->engine = eng;

    // TODO 1: Copy configuration if provided, otherwise use defaults
    // TODO 2: Copy callbacks if provided, otherwise use empty stubs
    // TODO 3: Initialize any additional state (stop_requested = false, etc.)

    return loop;
}

int event_loop_run(struct event_loop *loop) {
    if (!loop) return -1;

    // Setup phase
    if (loop->cbs.on_setup) {
```

```
    int ret = loop->cbs.on_setup(loop, loop->user_data);
    if (ret < 0) {
        LOG_ERROR("Setup callback failed: %d", ret);
        return ret;
    }
}

// Main event loop
while (1) {
    // TODO 4: Check if stop requested or should_stop callback returns true
    //         Break loop if so

    // Pre-submit phase: prepare new operations
    if (loop->cbs.on_pre_submit) {
        int ret = loop->cbs.on_pre_submit(loop, loop->user_data);
        if (ret < 0) {
            LOG_ERROR("Pre-submit callback failed: %d", ret);
            break;
        }
    }

    // TODO 5: Decide submission strategy based on config:
    //         - If busy_wait enabled, use non-blocking check first
    //         - If min_complete > 0, use engine_submit_and_wait
    //         - Otherwise, use engine_submit then engine_process_completions

    // TODO 6: Submit operations to kernel (call appropriate engine function)

    // TODO 7: Process completions (call engine_process_completions if not done above)
    //         Note: Completion processing triggers callbacks registered via user_data
}

// Cleanup phase
if (loop->cbs.on_cleanup) {
    loop->cbs.on_cleanup(loop, loop->user_data);
}
```

```
    return 0;
}

void event_loop_destroy(struct event_loop *loop) {
    if (!loop) return;
    // TODO 8: Free any allocated resources
    free(loop);
}

void event_loop_request_stop(struct event_loop *loop) {
    if (loop) {
        loop->stop_requested = true;
    }
}
```

C. Language-Specific Hints for C

1. **Memory Barriers:** When implementing the engine directly with syscalls (not liburing), use `__atomic_load_n` and `__atomic_store_n` with `__ATOMIC_ACQUIRE` and `__ATOMIC_RELEASE` for SQ/CQ head/tail updates to ensure proper synchronization between user space and kernel.
2. **Error Handling:** Check all syscall returns. For `io_uring_enter`, a negative return indicates error, but positive returns indicate number of submitted/completed entries. Use `errno` for detailed error codes.
3. **Buffer Alignment:** When using direct I/O or fixed buffers, ensure buffers are aligned to the block size (typically 512 bytes). Use `posix_memalign` for allocation.
4. **Signal Safety:** Avoid calling non-async-signal-safe functions in signal handlers. Use `signalfd` with `io_uring` for safe signal handling within the event loop.
5. **Debugging Aids:** Store operation type and connection ID in the upper bits of `user_data` for easy decoding in completion handlers. Implement a debug mode that logs every SQE submission and CQE completion.

Data Model

> **Milestone(s):** This section defines the foundational data structures that underpin all four milestones, with core types for io_uring operations (Milestone 1), buffer management (Milestone 2), connection state (Milestone 3), and advanced operation chaining (Milestone 4).

At the heart of any io_uring-based server lies a carefully designed data model that must efficiently track in-flight operations, manage connection state, and coordinate buffer ownership. Unlike traditional synchronous I/O where operations complete immediately, io_uring introduces inherent asynchrony: you submit a request, continue working, and later learn about its completion. This decoupling requires a robust system of **correlation tokens** (to match completions to their originating context) and **state machines** (to track the lifecycle of connections and operations).

Think of this data model as a **warehouse inventory system** for asynchronous I/O operations. When you submit an SQE, it's like placing a package on a conveyor belt with a tracking number (`user_data`). You record in your inventory system what that package contains (which connection, which buffer, what operation type). When the package returns on the completion belt (CQE) with the same tracking number, you consult your inventory to understand what to do with it—deliver it to a specific client, return the buffer to the pool, or transition the connection to the next state.

Core Types and Structures

The data model consists of four interrelated categories: (1) **io_uring core structures** provided by the kernel API, (2) **engine wrapper structures** that abstract the raw API, (3) **operation context structures** that track in-flight I/O, and (4) **resource management structures** for buffers and connections.

io_uring Core Structures (Kernel API)

These structures are defined by the Linux kernel and represent the fundamental building blocks of the io_uring interface. They are typically accessed via the liburing helper library but can be manipulated directly for advanced use cases.

Structure Name	Field Name	Type	Description
`struct io_uring`	`sq_ring`	`struct io_sq_ring`	Submission queue ring metadata (head, tail, ring mask, entries array)
	`cq_ring`	`struct io_cq_ring`	Completion queue ring metadata (head, tail, ring mask, overflow count, CQE array)
	`fd`	`int`	File descriptor for the io_uring instance, used for `io_uring_enter` syscalls
`struct io_uring_sqe`	`opcode`	`uint8_t`	Operation type (e.g., `IORING_OP_READ`, `IORING_OP_WRITE`)
	`fd`	`int`	Target file descriptor (socket, file, etc.) for the operation
	`addr`	`uint64_t`	Buffer address (for read/write) or pointer to operation-specific data
	`len`	`uint32_t`	Buffer length in bytes
	`offset`	`uint64_t`	File offset for file operations (or `IOSQE_OFF_INLINE` for inline data)
	`user_data`	`uint64_t`	User-defined token passed through to the corresponding CQE
	`flags`	`uint8_t`	SQE flags (`IOSQE_IO_LINK`, `IOSQE_IO_DRAIN`, `IOSQE_ASYNC`, etc.)
	`ioprio`	`uint16_t`	I/O priority hint (0 for default)
	`buf_index`	`uint16_t`	Index into registered buffers (for fixed buffer operations)
	`personality`	`uint16_t`	Credentials index for restricted operations
	`splice_fd_in`	`int32_t`	File descriptor for splice operations
	`__pad2[2]`	`uint64_t`	Padding/reserved fields for future expansion
`struct io_uring_cqe`	`user_data`	`uint64_t`	Copy of the `user_data` token from the corresponding SQE
	`res`	`int32_t`	Result of the operation: bytes transferred for read/write, negative error code on failure
	`flags`	`uint32_t`	Completion flags (e.g., `IORING_CQE_F_MORE` for multishot)

Engine Wrapper Structures (Abstraction Layer)

These structures wrap the raw io_uring API to provide a cleaner, more maintainable interface for the server. They manage configuration, statistics, and the event loop integration.

Structure Name	Field Name	Type	Description
`struct engine_config`	`sq_entries`	`unsigned int`	Number of entries in the submission queue (power of two recommended)
	`cq_entries`	`unsigned int`	Number of entries in the completion queue (typically 2× SQ size)
	`flags`	`unsigned int`	io_uring setup flags (`IORING_SETUP_IOPOLL` , `IORING_SETUP_SQPOLL` , etc.)
	`enable_sqpoll`	`bool`	Whether to enable kernel-side submission queue polling (reduces syscalls)
	`sqpoll_cpu`	`unsigned int`	CPU affinity for SQPOLL thread (if enabled)
`struct engine_stats`	`sqe_submitted`	`uint64_t`	Total number of SQEs submitted since engine creation
	`cqe_processed`	`uint64_t`	Total number of CQEs processed since engine creation
	`io_enter_calls`	`uint64_t`	Number of `io_uring_enter` syscalls made
	`batches_submitted`	`uint64_t`	Number of times multiple SQEs were submitted in a single syscall
`struct io_uring_ctx`	`ring`	`struct io_uring`	The underlying io_uring instance
	`config`	`struct engine_config`	Configuration used to create this instance
	`stats`	`struct engine_stats`	Running statistics for monitoring and debugging
	`fixed_buffers`	`struct iovec*`	Array of registered fixed buffers (if any)
	`fixed_buffer_count`	`unsigned int`	Number of registered fixed buffers
`struct io_uring_engine`	`ctx`	`struct io_uring_ctx*`	Opaque pointer to the engine context (hides implementation details)
`struct event_callbacks`	`on_setup`	`int (*)(struct event_loop*)`	Called once during event loop initialization
	`on_pre_submit`	`int (*)(struct event_loop*)`	Called before each batch submission (prepare new SQEs)
	`should_stop`	`bool (*)(struct event_loop*)`	Called to check if the event loop should terminate
	`on_cleanup`	`void (*)(struct event_loop*)`	Called after the event loop stops (cleanup resources)
`struct loop_config`	`max_batch_size`	`int`	Maximum number of SQEs to submit per `io_uring_enter` call
	`min_complete`	`int`	Minimum number of CQEs to wait for when blocking (0 = non-blocking)
	`flags`	`unsigned int`	Flags for `io_uring_enter` (`IORING_ENTER_GETEVENTS` , etc.)
	`busy_wait`	`bool`	Whether to busy-wait for completions instead of blocking
	`busy_wait_usec`	`unsigned int`	Microseconds to busy-wait before falling back to blocking
`struct event_loop`	`engine`	`struct io_uring_engine*`	The io_uring engine driving the event loop

Structure Name	Field Name	Type	Description
	`config`	`struct loop_config`	Configuration for the event loop behavior
	`cbs`	`struct event_callbacks`	Callback functions for custom event processing
	`user_data`	`void*`	User-provided context passed to callbacks
	`stop_requested`	`volatile bool`	Atomic flag to request graceful shutdown

Operation Context Structures (In-Flight Tracking)

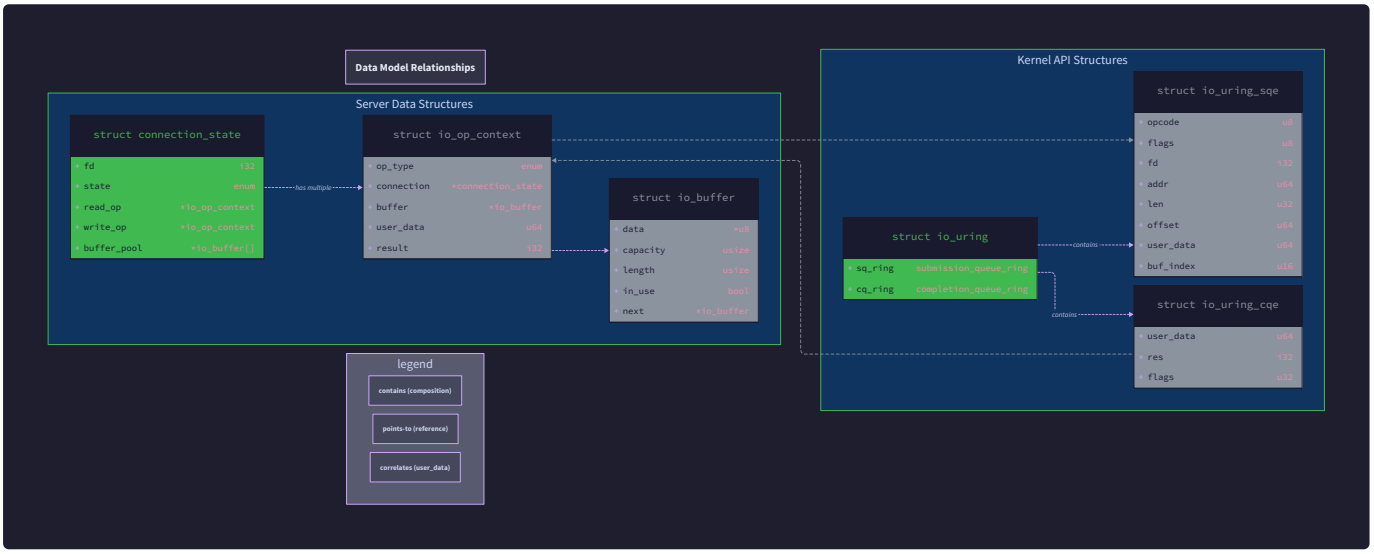
Each submitted SQE requires associated context to be properly handled upon completion. These structures form the "inventory records" that map `user_data` tokens back to their original operation details.

Structure Name	Field Name	Type	Description
`struct io_op_context`	`id`	`uint64_t`	Unique identifier used as the `user_data` token
	`op_type`	`uint8_t`	Operation type (mirroring SQE `opcode`)
	`connection`	`struct connection_state*`	Pointer to the associated connection (NULL for file ops)
	`buffer`	`struct io_buffer*`	Pointer to the buffer used for this operation
	`file`	`struct file_context*`	Pointer to file context (for file I/O, NULL otherwise)
	`chain_next`	`struct io_op_context*`	Next operation in a linked chain (for `IOSQE_IO_LINK`)
	`aux_data`	`union`	Operation-specific auxiliary data (offsets, sizes, flags)
`struct file_context`	`fd`	`int`	File descriptor for the opened file
	`path`	`char*`	Path to the file (for debugging/reopening)
	`file_size`	`off_t`	Size of the file in bytes (cached)
	`ref_count`	`int`	Reference count (multiple in-flight ops may share this)

Resource Management Structures (Buffers and Connections)

These structures manage the lifecycle of two critical resources: memory buffers for data transfer and network connection state.

Structure Name	Field Name	Type	Description
`struct io_buffer`	`data`	`void*`	Pointer to the allocated memory region
	`capacity`	`size_t`	Total size of the allocated region
	`in_use`	`bool`	Whether this buffer is currently referenced by an in-flight operation
	`is_fixed`	`bool`	Whether this buffer is part of the registered fixed buffer set
	`fixed_index`	`uint16_t`	Index within the fixed buffer array (if `is_fixed` is true)
	`zc_refcount`	`int`	Reference count for zero-copy operations (must reach 0 before reuse)
`struct connection_state`	`fd`	`int`	Socket file descriptor for this connection
	`remote_addr`	`struct sockaddr_in`	Remote client address (IPv4 example)
	`state`	`enum conn_state`	Current state (`ACCEPTING` , `READING` , `WRITING` , `CLOSING` , `CLOSED`)
	`read_buffer`	`struct io_buffer*`	Buffer currently being used for read operations
	`write_buffer`	`struct io_buffer*`	Buffer currently being used for write operations
	`pending_ops`	`int`	Count of in-flight I/O operations for this connection
	`last_active`	`time_t`	Timestamp of last I/O activity (for timeouts)
	`user_data`	`uint64_t`	`user_data` token for the most recently submitted operation



Relationships and Lifecycles

The power of this data model lies in how these structures reference each other to maintain coherence across asynchronous operations. Let's examine the key relationships and when each structure is created, used, and destroyed.

The Correlation Chain: CQE → Operation Context → Resources

When a completion queue entry (`struct io_uring_cqe`) is harvested, its `user_data` field contains a token that must be mapped back to the original operation context. This is typically implemented as a lookup table (array or hash table) indexed by `user_data` . The `struct io_op_context` found provides all necessary information to handle the completion:

1. **Operation Type**: Determines what action to take (e.g., for `IORING_OP_READ` , process the data in the buffer; for `IORING_OP_ACCEPT` , create a new connection state).
2. **Connection Pointer**: If non-NULL, directs the completion to the appropriate connection's state machine.

3. **Buffer Pointer**: Indicates which buffer contains the data (for reads) or can be released (for writes).
4. **Chain Pointer**: If operations were linked, points to the next operation to submit.

This chain of references enables the server to remain completely stateless in its main event loop—all necessary context travels with the operation via the `user_data` token.

> **Design Insight:** The `user_data` token serves as a "claim ticket" for your I/O operations. Just as a dry cleaner gives you a numbered ticket when you drop off clothes, you get back the same number when they're ready. Your job is to maintain the mapping from ticket numbers to garment descriptions (operation context).

Connection Lifecycle State Machine

Each `struct connection_state` progresses through a well-defined state machine, with transitions triggered by I/O completions:

Current State	Triggering Event	Next State	Actions Taken
`ACCEPTING`	`IORING_OP_ACCEPT` completes successfully	`READING`	Allocate read buffer, submit `IORING_OP_READ` SQE
`READING`	`IORING_OP_READ` completes (bytes > 0)	`WRITING`	Process data, prepare response, submit `IORING_OP_WRITE` SQE
`READING`	`IORING_OP_READ` completes (0 bytes = EOF)	`CLOSING`	Initiate graceful shutdown or immediate close
`WRITING`	`IORING_OP_WRITE` completes successfully	`READING`	Release write buffer, submit new `IORING_OP_READ` SQE
Any state	Any operation completes with error (e.g., ECONNRESET)	`CLOSING`	Cancel pending operations, release buffers
`CLOSING`	All pending operations completed	`CLOSED`	Close socket fd, free connection structure

The `pending_ops` counter is crucial: it tracks how many in-flight I/O operations reference this connection. When transitioning to `CLOSING`, we must wait for this counter to reach zero before freeing the connection memory, as kernel may still write to buffers referenced by pending operations.

Buffer Ownership and Reference Counting

Buffer management follows strict ownership rules to prevent use-after-free errors:

1. **Allocation**: Buffers are allocated from a pool (`struct io_buffer_pool`) at startup. Fixed buffers are registered once via `io_uring_register` .
2. **Acquisition**: When preparing an SQE, a buffer is marked `in_use = true` . For zero-copy sends, `zc_refcount` is incremented.
3. **Submission**: The SQE's `addr` field points to `buffer->data` , and `user_data` references the operation context that holds the buffer pointer.
4. **Completion**: When the CQE arrives, the buffer remains valid until processed. For reads, data is copied out; for writes, buffer can be released.
5. **Release**: Buffer is marked `in_use = false` (and `zc_refcount` decremented if zero-copy). It returns to the available pool.

The critical rule: **A buffer must not be reused while any SQE references it, even if that SQE hasn't been submitted yet.** This is because SQEs are batched—you might prepare several SQEs with different buffers, then submit them together. If you reused a buffer between preparation and submission, you'd create a race condition.

Creation and Destruction Points

Understanding when each structure is created and destroyed prevents resource leaks:

Structure	Creation Point	Destruction Point	Notes
`struct io_uring_ctx`	`engine_create()` called with configuration	`engine_destroy()` called	Must unregister fixed buffers before destruction
`struct io_op_context`	When preparing a new I/O operation	After processing its CQE and any chained operations	Use object pooling to avoid allocation overhead
`struct connection_state`	`IORING_OP_ACCEPT` completion handler	`pending_ops` reaches zero in `CLOSING` state	Must cancel any pending operations before freeing
`struct io_buffer`	Buffer pool initialization or dynamic allocation	Buffer pool destruction or when pool shrinks	Fixed buffers must persist until after unregistration
`struct file_context`	First request for a file path	Reference count reaches zero	Can be cached for repeated access

Memory Barrier Requirements

The io_uring rings use shared memory between user space and kernel space. Proper synchronization requires memory barriers when updating ring head/tail pointers:

1. **Submission Side**: After writing SQEs into the ring array, you must write a memory barrier before updating the SQ tail to make SQEs visible to the kernel.
2. **Completion Side**: After reading CQEs from the ring array, you must write a memory barrier before updating the CQ head to signal consumption to the kernel.

While liburing helpers handle these barriers internally, when implementing advanced optimizations or writing your own ring management, you must ensure:

- `io_uring_smp_store_release()` for tail updates
- `io_uring_smp_load_acquire()` for head reads
- Compiler barriers (`asm volatile("" ::: "memory")`) to prevent reordering

> **Common Pitfall:** Forgetting memory barriers can cause "lost completions" where the kernel doesn't see your submitted SQEs, or your application doesn't see completed CQEs. Symptoms include hangs where operations appear to never complete.

Zero-Copy Buffer Lifecycle Extensions

For `IORING_OP_SEND_ZC` operations, buffer lifecycle extends beyond the immediate completion. The kernel may retain the buffer for network stack processing. The `struct io_buffer` includes a `zc_refcount` to track these extended references:

1. On submission: `zc_refcount++`
2. On notification callback (optional): `zc_refcount--`
3. On final completion: `zc_refcount--`
4. Buffer can be reused only when `in_use == false` AND `zc_refcount == 0`

This extended lifecycle requires careful coordination—if the server needs to shut down while zero-copy operations are in flight, it must wait for all `zc_refcount`s to reach zero or explicitly cancel the operations.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Operation Context Storage	Array indexed by `user_data` modulo table size	Hash table with linear probing for better distribution
Buffer Pool Management	Static array of pre-allocated buffers	Dynamic pool with buddy allocator for variable sizes
Connection Lookup	Direct array indexed by connection ID	Red-black tree keyed by socket fd for O(log n) lookup
Memory Barriers	Use liburing helper functions (`io_uring_submit` , `io_uring_cqe_seen`)	Manual barriers with compiler intrinsics for micro-optimization

Recommended File/Module Structure

```
project-root/
├── include/
│   ├── data_model.h          # All structure definitions and types
│   ├── engine.h              # Engine wrapper API
│   └── event_loop.h          # Event loop API
├── src/
│   ├── core/
│   │   ├── data_model.c      # Structure initialization/cleanup helpers
│   │   ├── engine.c          # Engine implementation (Milestone 1)
│   │   └── event_loop.c      # Event loop implementation
│   ├── network/
│   │   ├── connection.c      # Connection state management (Milestone 3)
│   │   └── protocol.c        # Protocol handlers
│   ├── file/
│   │   ├── file_io.c         # File operation contexts (Milestone 2)
│   │   └── buffer_pool.c     # Buffer management
│   └── advanced/
│       ├── zero_copy.c       # Zero-copy buffer tracking (Milestone 4)
│       └── linked_ops.c      # Linked operation chains
└── tests/
    ├── test_data_model.c     # Unit tests for data structures
    └── integration/
        └── test_lifecycle.c  # Integration tests for full lifecycles
```

Infrastructure Starter Code

data_model.h (complete header):

```c
#ifndef DATA_MODEL_H

#define DATA_MODEL_H


#include <stdint.h>

#include <stdbool.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <time.h>


// Connection state machine

typedef enum {

    CONN_ACCEPTING,   // Waiting for accept to complete

    CONN_READING,     // Waiting for data to read

    CONN_WRITING,     // Waiting for write to complete

    CONN_CLOSING,     // Cleaning up, waiting for pending ops

    CONN_CLOSED       // Fully cleaned up, can be freed

} conn_state_t;


// Core operation context

typedef struct io_op_context {

    uint64_t id;                    // Unique ID used as user_data

    uint8_t op_type;                // IORING_OP_* value

    void* connection;               // struct connection_state*

    void* buffer;                   // struct io_buffer*

    void* file;                     // struct file_context*

    struct io_op_context* chain_next; // Next in linked chain

    union {

        struct {

            off_t file_offset;

            size_t expected_len;

        } file_op;

        struct {

            int listen_fd;

            socklen_t addrlen;

        } accept_op;

        struct {

            uint32_t flags;

            uint64_t aux1;
```

```
            uint64_t aux2;
        } generic;
    } aux_data;
} io_op_context_t;

// Buffer management
typedef struct io_buffer {
    void* data;
    size_t capacity;
    bool in_use;
    bool is_fixed;
    uint16_t fixed_index;
    int zc_refcount;                // For zero-copy operations
    struct io_buffer* next;         // For pool free list
} io_buffer_t;

// Connection state
typedef struct connection_state {
    int fd;                         // Socket file descriptor
    struct sockaddr_in remote_addr; // Client address
    conn_state_t state;             // Current state
    io_buffer_t* read_buffer;       // Current read buffer
    io_buffer_t* write_buffer;      // Current write buffer
    int pending_ops;                // In-flight operations count
    time_t last_active;             // Last I/O activity timestamp
    uint64_t user_data;             // user_data of last submitted op
    void* user_ctx;                 // Protocol-specific context
} connection_state_t;

// File context
typedef struct file_context {
    int fd;
    char* path;
    off_t file_size;
    int ref_count;
    time_t last_access;
} file_context_t;

// Context table for user_data lookup
```

```c
typedef struct context_table {
    io_op_context_t** slots;
    uint64_t mask;                  // Table size - 1 (power of two)
    uint64_t count;                 // Number of active contexts
} context_table_t;

// Function prototypes
context_table_t* create_context_table(size_t size);
void destroy_context_table(context_table_t* table);
io_op_context_t* allocate_op_context(context_table_t* table);
void free_op_context(context_table_t* table, io_op_context_t* ctx);
io_op_context_t* lookup_op_context(context_table_t* table, uint64_t user_data);

#endif // DATA_MODEL_H
```

data_model.c (starter implementation):

```c
#include "data_model.h"
#include <stdlib.h>
#include <string.h>
#include <assert.h>

// Create a context table with given size (must be power of two)
context_table_t* create_context_table(size_t size) {
    context_table_t* table = calloc(1, sizeof(context_table_t));
    if (!table) return NULL;

    table->slots = calloc(size, sizeof(io_op_context_t*));
    if (!table->slots) {
        free(table);
        return NULL;
    }

    table->mask = size - 1;
    table->count = 0;
    return table;
}

// Destroy context table and all contained contexts
void destroy_context_table(context_table_t* table) {
    if (!table) return;

    // Free all contexts still in the table
    for (size_t i = 0; i <= table->mask; i++) {
        if (table->slots[i]) {
            free(table->slots[i]);
            table->count--;
        }
    }

    assert(table->count == 0); // All contexts should have been freed
    free(table->slots);
    free(table);
}
```

```c
// Allocate and initialize a new operation context
io_op_context_t* allocate_op_context(context_table_t* table) {
    // TODO 1: Find an unused slot in the table (linear probing)
    // TODO 2: Allocate memory for io_op_context_t
    // TODO 3: Generate a unique ID (e.g., mix slot index with counter)
    // TODO 4: Initialize all fields to safe defaults
    // TODO 5: Insert into table using ID as key
    // TODO 6: Increment table->count
    // TODO 7: Return the allocated context

    // HINT: For the unique ID, you can combine: (slot_index << 32) | sequence_number
    // This ensures IDs are unique even if contexts are reused

    return NULL; // Replace with actual implementation
}

// Look up context by user_data token
io_op_context_t* lookup_op_context(context_table_t* table, uint64_t user_data) {
    // TODO 1: Compute hash index: user_data & table->mask
    // TODO 2: Linear probe until finding matching ID or empty slot
    // TODO 3: Return found context or NULL if not found
    // TODO 4: Handle the case where the slot is occupied but ID doesn't match (collision)

    // WARNING: Don't modify the context during lookup - this may be called from
    // the completion processing path while other threads are allocating contexts

    return NULL; // Replace with actual implementation
}

// Free a context and remove it from the table
void free_op_context(context_table_t* table, io_op_context_t* ctx) {
    if (!ctx || !table) return;

    // TODO 1: Find the context in the table (use lookup to get exact position)
    // TODO 2: Clear the table slot (set to NULL)
    // TODO 3: Decrement table->count
    // TODO 4: Free any resources owned by the context (buffers, etc.)
    // TODO 5: Free the context structure itself
```

```
    // IMPORTANT: For fixed buffers, just mark as available - don't free the memory
    // For zero-copy buffers, ensure zc_refcount is 0 before marking available
}
```

Core Logic Skeleton Code

buffer_pool.c (skeleton for Milestone 2):

```c
#include "data_model.h"
#include <sys/uio.h>

typedef struct buffer_pool {
    io_buffer_t* buffers;
    size_t count;
    size_t fixed_count;
    struct iovec* iovs;          // For io_uring_register
    io_buffer_t* free_list;      // Linked list of available buffers
    pthread_mutex_t lock;        // For thread safety (if needed)
} buffer_pool_t;

// Initialize buffer pool with fixed registration
buffer_pool_t* create_buffer_pool(size_t buffer_size, size_t buffer_count,
                                  bool register_fixed, struct io_uring* ring) {
    buffer_pool_t* pool = calloc(1, sizeof(buffer_pool_t));
    if (!pool) return NULL;

    // TODO 1: Allocate array of io_buffer_t structures
    // TODO 2: For each buffer: allocate aligned memory, set fields
    // TODO 3: Build free_list linked list
    // TODO 4: If register_fixed is true: prepare iovec array, call io_uring_register
    // TODO 5: Set fixed_index on each buffer if registered
    // TODO 6: Initialize mutex if using threading

    return pool;
}

// Get an available buffer from the pool
io_buffer_t* acquire_buffer(buffer_pool_t* pool) {
    // TODO 1: Lock mutex if threading
    // TODO 2: Check free_list - if empty, return NULL (or allocate new)
    // TODO 3: Remove buffer from free_list, mark in_use = true
    // TODO 4: Unlock mutex
    // TODO 5: Return buffer

    // NOTE: For zero-copy operations, you'll need to check zc_refcount == 0
```

```
    return NULL;
}

// Return buffer to pool after completion
void release_buffer(buffer_pool_t* pool, io_buffer_t* buffer) {
    if (!buffer) return;

    // TODO 1: Wait for zc_refcount to reach 0 if it was a zero-copy buffer
    // TODO 2: Lock mutex if threading
    // TODO 3: Mark in_use = false, reset zc_refcount = 0
    // TODO 4: Add buffer back to free_list
    // TODO 5: Unlock mutex

    // WARNING: Never release a buffer while an SQE still references it!
    // Ensure all completions are processed before releasing.
}
```

Language-Specific Hints (C)

- **Memory Alignment**: Use `posix_memalign()` to allocate buffers aligned to page boundaries (4096 bytes) for optimal performance with direct I/O.
- **Atomic Operations**: For `pending_ops` and `zc_refcount`, use GCC builtins like `__atomic_add_fetch()` and `__atomic_load_n()` for thread-safe updates without locks.
- **Zero-Copy Notification**: When using `IORING_OP_SEND_ZC`, you may receive a separate notification CQE. Use `IORING_CQE_F_NOTIF` flag in the CQE to distinguish it from the main completion.
- **Linked Operations**: When setting up linked SQEs, ensure all SQEs in the chain have `IOSQE_IO_LINK` flag set except the last one. The chain executes sequentially and stops on first error.

Milestone Checkpoint: Data Model Validation

After implementing the core data structures, verify they work correctly:

```bash
# Compile and run the data model unit tests
gcc -o test_data_model tests/test_data_model.c src/core/data_model.c -lpthread
./test_data_model

# Expected output:
# [PASS] Context table creation and destruction
# [PASS] Operation context allocation and lookup
# [PASS] Buffer pool acquisition and release
# [PASS] Connection state transitions
# [PASS] Zero-copy reference counting
# All tests passed!
```

Manual verification steps:

1. Create a context table with 1024 slots, allocate 500 contexts, verify all can be looked up by their `user_data`.
2. Simulate buffer lifecycle: acquire buffer, mark in-use, release, verify it can be reacquired.
3. Test connection state machine by calling transition functions and verifying `pending_ops` is updated correctly.
4. Verify that attempting to acquire a buffer with `zc_refcount > 0` fails (returns NULL).

Signs of problems:

- **Memory leaks**: Run with `valgrind --leak-check=full` and ensure all allocations are freed.
- **Race conditions**: Run stress tests with multiple threads allocating/freeing contexts.
- **Incorrect lookups**: Add assertion that `lookup_op_context(table, ctx->id) == ctx`.

Component Design: The io_uring Engine Core

> **Milestone(s):** Milestone 1: Basic SQ/CQ Operations

This component represents the heart of the entire system—the asynchronous I/O engine built around Linux's io_uring interface. It manages the **submission queue (SQ)** and **completion queue (CQ)** that form the core communication channel between user space and the kernel. This section covers the foundational design decisions and implementation patterns for initializing, operating, and managing the io_uring instance, directly addressing the acceptance criteria for Milestone 1.

Mental Model: The Kitchen Ticket System

Before diving into technical details, imagine a high-end restaurant kitchen during peak dinner service. This mental model will help you understand io_uring's core mechanics intuitively:

- **Submission Queue (SQ) as the Kitchen Order Ticket Rail**: When a waiter takes an order from a customer, they write it on a ticket and place it on a spinning order rail that faces the kitchen. Each ticket represents a single dish that needs preparation (I/O operation). The kitchen staff (kernel) continuously takes tickets from this rail. In io_uring, the SQ is this rail—a ring buffer where your application places **Submission Queue Entries (SQEs)** describing I/O operations (read file X at offset Y into buffer Z).
- **Completion Queue (CQ) as the Finished Order Pickup Counter**: Once the kitchen completes a dish, they place it on a heated pickup counter with the original ticket attached. Waiters periodically check this counter to retrieve completed dishes and deliver them to customers. The CQ is this counter—a ring buffer where the kernel places **Completion Queue Entries (CQEs)** indicating finished operations, preserving the original "ticket" (`user_data`) for correlation.
- **Batched Submissions as Ticket Batches**: Instead of walking to the kitchen after every single order, an efficient waiter collects multiple orders and places them on the rail in one trip. This reduces unnecessary walking (syscall overhead). Similarly, io_uring allows batching multiple SQEs and submitting them with a single `io_uring_enter` syscall.
- **`user_data` as the Claim Ticket**: Each order ticket has a unique number. When the dish is ready, the kitchen shouts "Order #42 is up!" The waiter knows exactly which table gets that dish. The `user_data` field serves this exact purpose—a user-defined token passed from SQE to corresponding CQE, allowing your application to correlate completions with their original requests without expensive lookups.

This model reveals io_uring's efficiency: the kitchen (kernel) can work ahead on multiple orders while waiters (application) continue taking new orders, with minimal coordination overhead. Unlike traditional models where waiters must constantly check if each individual order is ready (polling) or wait at the kitchen door until something is ready (blocking), io_uring provides a dedicated, shared space for efficient bidirectional communication.

ADR: Initialization and Ring Sizing

Decision: Use liburing Helper Functions for Initialization with Manual Ring Sizing

- **Context**: Setting up an io_uring instance involves multiple steps: creating the instance via `io_uring_setup`, mmap'ing the shared memory rings, and optionally registering resources. We must decide between using the low-level system calls directly versus the higher-level `liburing` helper library. Additionally, we must determine appropriate sizes for the submission and completion queues based on expected workload.
- **Options Considered**:
  1. **Pure system calls**: Manually call `io_uring_setup`, mmap rings with correct offsets, manage barriers and synchronization entirely ourselves.
  2. **liburing helpers**: Use `io_uring_queue_init` and related functions from the `liburing` library, which wrap setup and teardown logic.
  3. **Hybrid approach**: Use `liburing` for setup but manually manage ring operations for maximum control.

- **Decision**: Use `io_uring_queue_init` from `liburing` for initialization and teardown, while manually specifying ring sizes based on configuration. We will use `liburing`'s helper functions for common operations (`io_uring_get_sqe`, `io_uring_submit`, etc.) but understand their internal workings.
- **Rationale**:
  - **Educational balance**: `liburing` reduces boilerplate and common mistakes in setup/teardown (incorrect mmap sizes, missing barriers) while still exposing the core ring operations we need to learn.
  - **Production readiness**: `liburing` is widely used in production systems and maintained alongside kernel io_uring development, ensuring compatibility.
  - **Focus on core concepts**: Manual ring manipulation for submissions and completions remains visible, keeping the educational focus on SQ/CQ mechanics rather than setup minutiae.
  - **Safety**: `liburing` handles edge cases like `IORING_SETUP_SQPOLL` configuration and feature detection.
- **Consequences**:
  - **Slightly abstracted**: Some low-level details of ring mmap layout are hidden, but we can still inspect the resulting `struct io_uring` fields.
  - **Dependency**: Requires linking against `liburing` (or including its header-only version).
  - **Consistent patterns**: Aligns with most real-world io_uring codebases and documentation.

Ring Sizing Decision:

Option	Pros	Cons	Chosen?
Fixed small sizes (e.g., SQ=4096, CQ=8192)	Simple, predictable memory usage. Good for learning.	May limit peak performance under high load if queues fill.	Yes for Milestone 1
Dynamically sized based on workload	Adapts to load, optimal memory usage.	Complex to implement, requires monitoring and resizing logic.	No for initial implementation
Kernel default sizes (use 0 in params)	No sizing decisions needed.	May be suboptimal for specific workloads, less predictable.	No
CQ = 2× SQ size (common pattern)	Prevents CQ overflow when many completions arrive while processing. Matches kernel recommendation.	Wastes memory if SQ size is already generous.	**Yes** – we adopt this heuristic

The sizing configuration will be captured in `struct engine_config`:

- `sq_entries`: Number of SQEs in the submission ring (power of two, e.g., 4096)
- `cq_entries`: Number of CQEs in the completion ring (typically `2 * sq_entries`)
- `flags`: IORING_SETUP_* flags (e.g., `IORING_SETUP_SQPOLL` for advanced use)

> **Key Insight**: The completion queue should be larger than the submission queue because multiple completions can arrive for a single submission (e.g., multishot operations), and completions may accumulate while your application is busy processing previous ones. A full CQ causes operations to fail with `-EBUSY`.

ADR: Submission and Completion Strategies

Decision: Batched Submission with Blocking Wait for Completions, with Optional Busy-Wait Tuning

- **Context**: Once the io_uring instance is initialized, we need a strategy for how and when to submit SQEs to the kernel and harvest CQEs. Key decisions include: batching granularity, when to call `io_uring_enter`, and whether to block waiting for completions or actively poll.
- **Options Considered**:
  1. **Submit immediately after each SQE preparation**: Call `io_uring_enter` after every `io_uring_get_sqe` and `io_uring_sqe_set_data`. Simplest but highest syscall overhead.
  2. **Batched submission on fixed intervals**: Accumulate SQEs until a timer fires or fixed count reached, then submit. Reduces syscalls but may increase latency.
  3. **Batched submission with opportunistic harvesting**: Submit when either (a) SQ is full, or (b) we need to wait for completions. Balances throughput and latency.
  4. **Busy-wait polling**: Use `IORING_SETUP_SQPULL` or spin on CQ without syscalls. Maximum performance but wastes CPU.

- **Decision**: Implement an **opportunistic batching strategy** with configurable behavior. By default, we will:
  - Prepare multiple SQEs without immediately submitting.
  - Submit when either: (1) We've prepared a configurable batch size (`max_batch_size`), or (2) We need to wait for completions because we're out of free SQEs or want to process results.
  - Use blocking `io_uring_enter` with `IORING_ENTER_GETEVENTS` when waiting for completions.
  - Provide a configuration flag `busy_wait` for experimental polling mode.
- **Rationale**:
  - **Syscall reduction**: Batching multiple SQEs in one `io_uring_enter` call dramatically reduces syscall overhead, which is a primary benefit of io_uring.
  - **Latency control**: The `max_batch_size` parameter allows tuning—smaller batches reduce latency at cost of more syscalls.
  - **Resource efficiency**: Blocking waits free CPU for other processes when no I/O is ready, unlike busy-waiting.
  - **Flexibility**: Configurable busy-wait allows experimentation and adaptation to different workloads (e.g., ultra-low latency vs. energy-efficient).
- **Consequences**:
  - **Implementation complexity**: Need to track prepared-but-unsubmitted SQEs and decide when to flush.
  - **Tuning required**: Optimal `max_batch_size` depends on workload characteristics.
  - **Blocking behavior**: Default blocking may not suit real-time applications; advanced users can enable busy-wait or SQPOLL.

Completion Harvesting Strategy:

Approach	When to Use	Implementation
Peek and process available	Always; non-blocking check for completions	`io_uring_peek_batch_cqe` to get available CQEs without waiting
Wait for minimum completions	When we need results to proceed or want to limit latency	`io_uring_enter` with `min_complete > 0` and `IORING_ENTER_GETEVENTS`
Wait for any completion	When event loop has nothing else to do	`io_uring_wait_cqe` (blocks until at least one CQE)
Busy-wait polling	Extreme low-latency requirements, dedicated CPU	Check CQ head/tail directly in a tight loop

Our design incorporates all four strategies through different code paths in the event loop, controlled by `struct loop_config` parameters:

- `max_batch_size` : Maximum SQEs to accumulate before submission
- `min_complete` : Minimum CQEs to wait for when calling `io_uring_enter`
- `busy_wait` : Boolean to enable active polling instead of blocking
- `busy_wait_usec` : Microseconds to spin before falling back to blocking

The main event loop logic follows this flowchart:

Common Pitfalls: Engine Core

⚠️ Pitfall: Incorrect Memory Barrier Usage Leading to Data Corruption

- **Description**: Directly accessing the SQ/CQ ring head and tail pointers without proper memory barriers can cause the kernel and application to see inconsistent states, leading to lost operations or corrupted data.
- **Why it's wrong**: The kernel and user space run on different CPUs with different caches. Without barriers, writes may not be visible in the expected order, causing the kernel to see stale SQ tail values (missing new SQEs) or the application to see stale CQ head values (re-processing old CQEs).
- **How to fix**: Always use the `liburing` helper functions (`io_uring_smp_store_release` , `io_uring_smp_load_acquire`) or the built-in `io_uring_sqe` / `io_uring_cqe` array accessors provided by `liburing` . If implementing manual ring access, follow the kernel documentation's exact barrier requirements.

⚠️ Pitfall: Forgetting to Advance CQ Tail After Processing CQE

- **Description**: After harvesting a CQE from the completion queue, failing to mark it as consumed by advancing the CQ tail pointer.
- **Why it's wrong**: The kernel will see the same CQE slot as still occupied and won't reuse it for new completions. Eventually, the CQ fills up, causing subsequent operation submissions to fail with `-EBUSY` .
- **How to fix**: Always call `io_uring_cqe_seen()` after processing a CQE. In batch processing, use `io_uring_cq_advance()` to mark multiple CQEs as consumed at once.

⚠️ Pitfall: Mishandling CQE Overflow (CQ Full Condition)

- **Description**: Not checking for the `IORING_CQ_F_OVERFLOW` flag in the CQ ring flags, which indicates that some completions were lost because the CQ was full.
- **Why it's wrong**: Lost completions mean your application doesn't know about finished I/O operations, causing hangs, resource leaks, or incorrect state.
- **How to fix**: Regularly check `io_uring_cq_has_overflow()` and handle overflow by: (1) increasing CQ size, (2) processing completions more aggressively, or (3) falling back to error handling for affected operations.

⚠️ Pitfall: Assuming CQE Order Matches SQE Submission Order

- **Description**: Writing code that expects CQEs to arrive in the same order as SQEs were submitted.

- **Why it's wrong**: io_uring completions are inherently unordered unless explicitly chained with `IOSQE_IO_LINK`. The kernel may complete operations out-of-order based on device readiness.
- **How to fix**: Always use the `user_data` field to correlate CQEs with their original context. Implement a lookup system (like our `context_table`) to find operation context from the `user_data` token.

⚠️ **Pitfall: Not Checking `res` Field for Negative Error Codes**

- **Description**: Assuming a CQE indicates success without checking the `res` field.
- **Why it's wrong**: I/O operations can fail due to various reasons (file not found, connection reset, permission denied). A negative `res` indicates an error, with `-errno` value.
- **How to fix**: Always check `cqe->res`. If negative, handle the error appropriately (clean up resources, retry if `-EAGAIN`, log, etc.). Positive values indicate bytes transferred for read/write operations.

⚠️ **Pitfall: Leaking SQEs When Submission Fails**

- **Description**: Preparing SQEs but not handling the case where `io_uring_submit()` fails (returns negative error or fewer SQEs submitted than expected).
- **Why it's wrong**: Prepared SQEs consume ring slots but may never be submitted, causing ring to eventually fill with stale entries.
- **How to fix**: Check return value of `io_uring_submit()`. On error, either retry or explicitly clear the affected SQEs (by advancing SQ tail without submission). Implement backpressure to stop preparing new SQEs when the ring is full.

Implementation Guidance

This section provides practical starting points for implementing the io_uring engine core. We'll use the C language with `liburing` for helper functions.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
io_uring Setup	`liburing` helper functions (`io_uring_queue_init`)	Manual `io_uring_setup` + mmap with explicit barrier control
Ring Sizing	Fixed compile-time sizes (SQ=4096, CQ=8192)	Dynamic sizing based on runtime configuration
Submission Strategy	Submit after each batch of prepared SQEs	Opportunistic batching with SQ fullness check
Completion Strategy	Blocking wait for completions	Hybrid busy-wait/blocking with configurable timeout
Memory Management	Standard `malloc`/`free` for operation contexts	Object pooling with reuse of context structures

B. Recommended File/Module Structure

For the engine core component, organize files as follows:

```
io_uring-server/
├── include/
│   ├── engine.h           # Engine public API and data structures
│   └── common.h           # Common constants and utilities
├── src/
│   ├── engine/
│   │   ├── engine.c       # Engine core implementation
│   │   ├── engine_priv.h  # Private engine declarations
│   │   └── CMakeLists.txt # Build file for engine component
│   ├── event_loop.c       # Main event loop implementation
│   └── main.c             # Entry point and configuration
└── tests/
    └── test_engine.c      # Unit tests for engine
```

C. Infrastructure Starter Code

Here's complete, working code for initializing and tearing down the io_uring engine:

```c
/* include/engine.h */

#ifndef ENGINE_H

#define ENGINE_H


#include <stdint.h>

#include <stdbool.h>

#include <liburing.h>


/* Engine configuration structure */

struct engine_config {

    unsigned int sq_entries;      /* Submission queue size (power of two) */

    unsigned int cq_entries;      /* Completion queue size (typically 2x SQ) */

    unsigned int flags;           /* IORING_SETUP_* flags */

    bool enable_sqpoll;           /* Enable kernel-side polling */

    unsigned int sqpoll_cpu;      /* CPU affinity for SQPOLL thread */

};


/* Engine statistics structure */

struct engine_stats {

    uint64_t sqe_submitted;       /* Total SQEs submitted */

    uint64_t cqe_processed;       /* Total CQEs processed */

    uint64_t io_enter_calls;      /* io_uring_enter syscall count */

    uint64_t batches_submitted;   /* Batches submitted (groups of SQEs) */

};


/* Opaque engine handle */

struct io_uring_engine;


/* Public API */

struct io_uring_engine *engine_create(const struct engine_config *cfg);

void engine_destroy(struct io_uring_engine *eng);

int engine_submit(struct io_uring_engine *eng);

int engine_submit_and_wait(struct io_uring_engine *eng, unsigned int min_complete);

int engine_process_completions(struct io_uring_engine *eng);

struct io_uring *engine_get_ring(struct io_uring_engine *eng);

void engine_get_stats(struct io_uring_engine *eng, struct engine_stats *out);


#endif /* ENGINE_H */
```

```c
/* src/engine/engine.c */

#include "engine.h"

#include "engine_priv.h"

#include <stdlib.h>

#include <string.h>

#include <errno.h>


/* Private engine context */

struct io_uring_ctx {

    struct io_uring ring;          /* liburing instance */

    struct engine_config config;   /* Configuration copy */

    struct engine_stats stats;     /* Runtime statistics */

};


struct io_uring_engine {

    struct io_uring_ctx *ctx;      /* Opaque pointer to context */

};


/* Create a new io_uring engine instance */

struct io_uring_engine *engine_create(const struct engine_config *cfg)

{

    struct io_uring_engine *eng = NULL;

    struct io_uring_ctx *ctx = NULL;

    struct io_uring_params params;

    int ret;



    /* Validate configuration */

    if (!cfg || cfg->sq_entries == 0) {

        errno = EINVAL;

        return NULL;

    }



    /* Allocate engine handle */

    eng = malloc(sizeof(*eng));

    if (!eng) {

        goto error;

    }
```

```
    /* Allocate private context */
    ctx = malloc(sizeof(*ctx));
    if (!ctx) {
        goto error;
    }

    memset(ctx, 0, sizeof(*ctx));
    memcpy(&ctx->config, cfg, sizeof(*cfg));

    /* Configure io_uring parameters */
    memset(&params, 0, sizeof(params));
    params.flags = cfg->flags;

    /* Enable SQPOLL if requested */
    if (cfg->enable_sqpoll) {
        params.flags |= IORING_SETUP_SQPOLL;
        if (cfg->sqpoll_cpu != (unsigned int)-1) {
            params.sq_thread_cpu = cfg->sqpoll_cpu;
        }
    }

    /* Initialize io_uring instance */
    ret = io_uring_queue_init_params(cfg->sq_entries, &ctx->ring, &params);
    if (ret < 0) {
        errno = -ret;
        goto error;
    }

    /* Verify actual ring sizes */
    if (io_uring_sq_space_left(&ctx->ring) < (int)cfg->sq_entries) {
        /* Kernel may have adjusted size */
        fprintf(stderr, "Warning: Actual SQ size differs from requested\n");
    }

    eng->ctx = ctx;
    return eng;
```

```c
error:
    free(ctx);
    free(eng);
    return NULL;
}

/* Destroy engine and free all resources */
void engine_destroy(struct io_uring_engine *eng)
{
    if (!eng || !eng->ctx) {
        return;
    }

    /* Clean up io_uring instance */
    io_uring_queue_exit(&eng->ctx->ring);

    /* Free context */
    free(eng->ctx);
    free(eng);
}

/* Get underlying io_uring instance (for advanced use) */
struct io_uring *engine_get_ring(struct io_uring_engine *eng)
{
    if (!eng || !eng->ctx) {
        return NULL;
    }
    return &eng->ctx->ring;
}

/* Submit prepared SQEs to the kernel */
int engine_submit(struct io_uring_engine *eng)
{
    int submitted;

    if (!eng || !eng->ctx) {
        errno = EINVAL;
        return -1;
```

```
    }

    submitted = io_uring_submit(&eng->ctx->ring);
    if (submitted > 0) {
        eng->ctx->stats.sqe_submitted += submitted;
        eng->ctx->stats.io_enter_calls++;
        eng->ctx->stats.batches_submitted++;
    }

    return submitted;
}

/* Submit and wait for completions */
int engine_submit_and_wait(struct io_uring_engine *eng, unsigned int min_complete)
{
    int submitted;

    if (!eng || !eng->ctx) {
        errno = EINVAL;
        return -1;
    }

    submitted = io_uring_submit_and_wait(&eng->ctx->ring, min_complete);
    if (submitted > 0) {
        eng->ctx->stats.sqe_submitted += submitted;
        eng->ctx->stats.io_enter_calls++;
        eng->ctx->stats.batches_submitted++;
    }

    return submitted;
}

/* Process available completions without waiting */
int engine_process_completions(struct io_uring_engine *eng)
{
    struct io_uring_cqe *cqe;
    unsigned int head;
    unsigned int count = 0;
```

```
    if (!eng || !eng->ctx) {
        errno = EINVAL;
        return -1;
    }

    /* Process all available CQEs */
    io_uring_for_each_cqe(&eng->ctx->ring, head, cqe) {
        /* TODO: Actual completion processing */
        /* For now, just count them */
        count++;
    }

    /* Mark all processed CQEs as seen */
    io_uring_cq_advance(&eng->ctx->ring, count);

    eng->ctx->stats.cqe_processed += count;
    return count;
}

/* Retrieve engine statistics */
void engine_get_stats(struct io_uring_engine *eng, struct engine_stats *out)
{
    if (!eng || !eng->ctx || !out) {
        return;
    }
    memcpy(out, &eng->ctx->stats, sizeof(*out));
}
```

D. Core Logic Skeleton Code

Here's skeleton code for the main event loop that drives the engine:

```c
/* src/event_loop.c */

#include "engine.h"

#include <stdlib.h>

#include <unistd.h>

#include <signal.h>

#include <stdio.h>


/* Event loop configuration */

struct loop_config {

    int max_batch_size;          /* Maximum SQEs to batch before submission */

    int min_complete;            /* Minimum CQEs to wait for */

    unsigned int flags;          /* Additional flags */

    bool busy_wait;              /* Enable busy-wait polling */

    unsigned int busy_wait_usec; /* Busy-wait microseconds before blocking */

};


/* Event callbacks for extensibility */

struct event_callbacks {

    int (*on_setup)(struct io_uring_engine *eng, void *user_data);

    int (*on_pre_submit)(struct io_uring_engine *eng, void *user_data);

    int (*should_stop)(void *user_data);

    void (*on_cleanup)(struct io_uring_engine *eng, void *user_data);

};


/* Event loop structure */

struct event_loop {

    struct io_uring_engine *engine;

    struct loop_config config;

    struct event_callbacks cbs;

    void *user_data;

    volatile bool stop_requested;

};


/* Create a new event loop instance */

struct event_loop *event_loop_create(struct io_uring_engine *eng,

                                     const struct loop_config *config,

                                     const struct event_callbacks *cbs,

                                     void *user_data)

{
```

```
    struct event_loop *loop = malloc(sizeof(*loop));
    if (!loop) {
        return NULL;
    }

    loop->engine = eng;
    if (config) {
        memcpy(&loop->config, config, sizeof(*config));
    } else {
        /* Default configuration */
        loop->config.max_batch_size = 32;
        loop->config.min_complete = 1;
        loop->config.flags = 0;
        loop->config.busy_wait = false;
        loop->config.busy_wait_usec = 100;
    }

    if (cbs) {
        memcpy(&loop->cbs, cbs, sizeof(*cbs));
    } else {
        memset(&loop->cbs, 0, sizeof(loop->cbs));
    }

    loop->user_data = user_data;
    loop->stop_requested = false;

    return loop;
}

/* Request the event loop to stop */
void event_loop_request_stop(struct event_loop *loop)
{
    if (loop) {
        loop->stop_requested = true;
    }
}

/* Main event loop execution */
```

```c
int event_loop_run(struct event_loop *loop)

{

    struct io_uring *ring;

    struct io_uring_cqe *cqes[32];  /* Batch of CQEs to process */

    int submitted_this_batch = 0;

    int ret;



    if (!loop || !loop->engine) {

        return -EINVAL;

    }



    ring = engine_get_ring(loop->engine);

    if (!ring) {

        return -ENODEV;

    }



    /* Call setup callback if provided */

    if (loop->cbs.on_setup) {

        ret = loop->cbs.on_setup(loop->engine, loop->user_data);

        if (ret < 0) {

            return ret;

        }

    }



    /* Main event loop */

    while (!loop->stop_requested) {

        /* Step 1: Check if we should stop via callback */

        if (loop->cbs.should_stop && loop->cbs.should_stop(loop->user_data)) {

            break;

        }



        /* Step 2: Call pre-submit callback to prepare new operations */

        if (loop->cbs.on_pre_submit) {

            ret = loop->cbs.on_pre_submit(loop->engine, loop->user_data);

            if (ret < 0) {

                /* Error in preparation */

                break;
```

```
        }

        /* ret may indicate how many SQEs were prepared */
        submitted_this_batch += (ret > 0 ? ret : 0);
    }

    /* Step 3: Submit batch if we've reached max size or need to wait */
    bool should_submit = false;
    if (submitted_this_batch >= loop->config.max_batch_size) {
        should_submit = true;
    } else if (io_uring_sq_space_left(ring) == 0) {
        /* SQ is full, must submit to free space */
        should_submit = true;
    }

    if (should_submit && submitted_this_batch > 0) {
        /* TODO: Decide between submit vs submit_and_wait based on:
           - Are we waiting for completions to proceed?
           - Is busy_wait enabled?
           - Do we need to meet min_complete requirement?
        */
        ret = engine_submit(loop->engine);
        if (ret < 0) {
            /* Submission failed */
            perror("io_uring submission failed");
            break;
        }
        submitted_this_batch = 0;
    }

    /* Step 4: Process completions */
    int cqe_count;
    if (loop->config.busy_wait) {
        /* Busy-wait for completions */
        unsigned int spins = 0;
        while (spins < loop->config.busy_wait_usec) {
            cqe_count = io_uring_peek_batch_cqe(ring, cqes,
                                sizeof(cqes)/sizeof(cqes[0]));
```

```
            if (cqe_count > 0) {
                break;
            }
            /* Tiny pause to reduce CPU usage */
            usleep(1);
            spins++;
        }
        if (cqe_count == 0) {
            /* Fall back to blocking wait */
            ret = io_uring_wait_cqe(ring, &cqes[0]);
            if (ret < 0) {
                break;
            }
            cqe_count = 1;
        }
    } else {
        /* Non-blocking peek for completions */
        cqe_count = io_uring_peek_batch_cqe(ring, cqes,
                            sizeof(cqes)/sizeof(cqes[0]));
        if (cqe_count == 0 && loop->config.min_complete > 0) {
            /* No completions ready, but we want to wait for some */
            ret = engine_submit_and_wait(loop->engine, loop->config.min_complete);
            if (ret < 0) {
                break;
            }
            /* Now peek again */
            cqe_count = io_uring_peek_batch_cqe(ring, cqes,
                                sizeof(cqes)/sizeof(cqes[0]));
        }
    }

    /* Step 5: Handle each completion */
    for (int i = 0; i < cqe_count; i++) {
        struct io_uring_cqe *cqe = cqes[i];

        /* TODO: Extract user_data from cqe->user_data */
        uint64_t user_data = cqe->user_data;
```

```
        /* TODO: Look up operation context using user_data */
        /* struct io_op_context *ctx = lookup_op_context(user_data); */

        /* TODO: Check cqe->res for errors */
        if (cqe->res < 0) {
            /* Error handling */
            fprintf(stderr, "I/O operation failed: %s\n", strerror(-cqe->res));
        } else {
            /* Success - process result */
            /* TODO: Based on operation type (stored in context),
               handle the completion:
               - For reads: process data in buffer
               - For writes: clean up buffer
               - For accepts: create new connection state
            */
        }

        /* TODO: Clean up or reuse operation context */
        /* free_op_context(ctx); */
    }

    /* Step 6: Advance completion queue */
    if (cqe_count > 0) {
        io_uring_cq_advance(ring, cqe_count);
    }

    /* Step 7: Check for overflow condition */
    if (io_uring_cq_has_overflow(ring)) {
        fprintf(stderr, "Warning: Completion queue overflow occurred\n");
        /* TODO: Handle overflow - increase CQ size or process faster */
    }
}

/* Call cleanup callback if provided */
if (loop->cbs.on_cleanup) {
    loop->cbs.on_cleanup(loop->engine, loop->user_data);
```

```
    }


    return 0;
}

/* Destroy event loop */
void event_loop_destroy(struct event_loop *loop)
{
    free(loop);
}
```

E. Language-Specific Hints

- **Memory Barriers**: When using `liburing`, the helper functions handle memory barriers automatically. If you need to implement manual ring access, use:

```c
#include <linux/io_uring.h>

/* For writing to shared ring fields */
WRITE_ONCE(ring->sq.tail, new_tail);

/* For reading from shared ring fields */
tail = READ_ONCE(ring->cq.tail);
```

- **Error Handling**: Always check return values from `io_uring_submit()`. Negative values indicate errors (with `-errno`). Common errors:
  - `-EBUSY`: Ring is full, process completions to free space
  - `-EINVAL`: Invalid parameters or unsupported operation
  - `-ENOMEM`: Kernel couldn't allocate resources
- **Buffer Alignment**: For optimal performance (especially with `O_DIRECT`), ensure buffers are aligned to 512-byte boundaries. Use `posix_memalign()`:

```c
void *buf;
if (posix_memalign(&buf, 512, buffer_size) != 0) {
    /* handle error */
}
```

F. Milestone Checkpoint

To verify your engine core implementation for Milestone 1:

1. **Build and run a simple test**:

```bash
# Create a test file
echo "Hello, io_uring!" > test.txt

# Compile your engine with a simple test program
gcc -o test_engine src/engine/engine.c src/event_loop.c test_engine.c -luring

# Run the test
./test_engine test.txt
```

2. **Expected behavior**:
   - The program should initialize io_uring without errors
   - Submit an asynchronous read operation for `test.txt`
   - Harvest the completion and print "Hello, io_uring!" from the buffer
   - Clean up and exit gracefully
3. **Verify batching works**: Use `strace` to confirm multiple operations are batched:

```bash
strace -e io_uring_enter ./test_engine test.txt 2>&1 | grep io_uring_enter
```

You should see fewer `io_uring_enter` calls than total operations if batching is working.

4. **Check for common mistakes**:
   - Run with `valgrind` to ensure no memory leaks
   - Monitor CPU usage (should be low unless using busy-wait)
   - Check kernel log for io_uring errors: `dmesg | tail -20`
5. **Signs something is wrong**:
   - Program hangs: Likely not harvesting completions or not advancing CQ tail
   - High CPU usage: May be busy-waiting incorrectly or in tight loop
   - "Bad file descriptor" errors: File not opened correctly before submitting SQE
   - "Resource temporarily unavailable": Ring is full, need to process completions faster

---

Component Design: Asynchronous File I/O Server

> **Milestone(s):** Milestone 2: File I/O Server

This component focuses on leveraging `io_uring` for high-performance file operations, moving beyond the basic queue operations of Milestone 1 to build a practical file server. The core challenge shifts from understanding queue mechanics to managing the lifecycle of data buffers across asynchronous operations while minimizing per-request overhead. This design enables serving multiple concurrent file reads efficiently, a common requirement in web servers, content delivery networks, and database systems where file I/O is a primary bottleneck.

Mental Model: The Library Book Retrieval System

Imagine a traditional library with a single librarian who must personally retrieve every book requested by patrons—this represents **synchronous I/O**. The librarian goes to the stacks, finds the book, and returns to the front desk, blocking all other patrons during this search. The library can serve only one patron at a time, leading to long queues.

Now, envision a modern library with a **request slip system**:

1. A patron fills out a slip with the book's call number and section (analogous to file descriptor and offset).
2. The patron hands the slip to the librarian and receives a numbered claim ticket (`user_data`).

3. The librarian places the slip on a conveyor belt (`SQ`) leading to a team of retrieval assistants (the kernel).
4. The patron continues browsing other catalogs (the server handles other requests) instead of waiting idly.
5. When an assistant retrieves the book, they place it on a "ready" shelf (`CQ`) with the claim ticket attached.
6. The librarian periodically checks the ready shelf, matches ticket numbers to patrons, and delivers the books.

This system allows multiple book requests to be in flight simultaneously. **Fixed buffer registration** is like providing the library with a set of standardized book carts (pre-registered buffers) that assistants can use immediately without needing to locate a cart for each request. The library knows each cart by a numbered slot, eliminating the overhead of cart allocation per request.

The key insight is that **asynchronous file I/O transforms sequential waiting into parallel processing**. By decoupling request submission from completion handling, the server can maintain high utilization of both CPU and disk resources, especially when serving multiple clients concurrently or reading from different file offsets.

ADR: Buffer Management Strategy

> **Decision: Hybrid Buffer Pool with Fixed Registration for High-Throughput Reads**
>
> **Context:** The file server must handle numerous concurrent read requests without incurring excessive per-operation overhead for buffer allocation and kernel registration. Each `IORING_OP_READ` requires a buffer for the data, and traditional approaches that allocate buffers per request introduce significant memory management and kernel entry overhead.
>
> **Options Considered:**
>
> 1. **Per-Request Allocation:** Dynamically allocate a buffer for each read operation, freeing it after the completion is processed.
> 2. **Fixed Buffer Pool with Registration:** Pre-allocate a pool of buffers at startup, register them with the kernel via `io_uring_register(..., IORING_REGISTER_BUFFERS, ...)` , and reference them by index in SQEs.
> 3. **Hybrid Approach:** Use fixed registered buffers for standard-sized operations (e.g., 64KB file chunks) and fall back to per-request allocation for irregular sizes or one-off operations.
>
> **Decision:** Implement a hybrid buffer pool (`struct buffer_pool`) that manages a set of pre-allocated, fixed-size buffers registered with the `io_uring` instance. For operations that fit the standard size, use fixed buffers via the `IOSQE_FIXED_FILE` flag and buffer index. For operations requiring different sizes, dynamically allocate buffers but avoid registration overhead.
>
> **Rationale:**
>
> - **Performance:** Fixed buffer registration eliminates kernel-level buffer mapping/unmapping per operation, reducing system call overhead and TLB pressure. This is critical for high-throughput scenarios with small to medium-sized reads.
> - **Predictability:** A buffer pool prevents memory fragmentation and provides bounded memory usage, which is essential for long-running servers.
> - **Flexibility:** The hybrid approach accommodates variable request sizes (like HTTP range requests) without complicating the core high-performance path.
> - **Milestone Requirement:** Milestone 2 explicitly requires fixed buffer registration implementation for benchmarking comparison.
>
> **Consequences:**
>
> - **Increased Startup Complexity:** The engine must register buffers during initialization and manage their lifecycle.
> - **Buffer Lifetime Management:** Buffers must remain registered and untouched until all in-flight operations using them complete. This requires careful reference counting or tracking via `io_op_context` .
> - **Size Trade-offs:** Choosing an inappropriate fixed buffer size (too small leads to multiple reads; too large wastes memory) requires profiling the workload. A size like 64KB or 128KB often balances disk block alignment and memory efficiency.
> - **Fallback Path:** The dynamic allocation path will be slower but necessary for edge cases; it should be instrumented to alert if used excessively in production.

Option	Pros	Cons	Chosen?
Per-Request Allocation	Simple to implement; no upfront memory commitment; handles any buffer size	High per-operation overhead (allocation/free, kernel mapping); memory fragmentation; poor cache locality	No
Fixed Buffer Pool with Registration	Minimal per-operation overhead; predictable memory usage; better cache locality	Upfront memory commitment; fixed buffer size may not fit all operations; complex lifecycle management	**Yes** (primary path)
Hybrid Approach	Balances performance and flexibility; meets milestone requirement	Implementation complexity; two code paths to maintain	**Yes** (with fallback)

The buffer pool manager will be responsible for:

- Initializing a set of `io_buffer` structures with memory aligned to the kernel's preferred alignment (typically 4096 bytes or 1MB for huge pages).
- Registering the buffer array with `io_uring_register(ring_fd, IORING_REGISTER_BUFFERS, iovecs, count)`.
- Providing an `acquire_buffer` function that returns an available buffer (marking it `in_use`) and its fixed index.
- Providing a `release_buffer` function that returns the buffer to the pool after the completion handler confirms the data has been processed and no further kernel references exist.

The following table details the buffer pool structure:

Field	Type	Description
`buffers`	`struct io_buffer*`	Array of buffer descriptors, pre-allocated and managed
`count`	`size_t`	Total number of buffers in the pool
`fixed_count`	`size_t`	Number of buffers registered as fixed (≤ count)
`iovs`	`struct iovec*`	Array of iovec structures pointing to buffer memory, passed to `io_uring_register`
`free_list`	`struct io_buffer*`	Linked list of currently available buffers for quick acquisition
`lock`	`pthread_mutex_t`	Mutex protecting pool state (acquisition/release)

Each `io_buffer` descriptor tracks:

Field	Type	Description
`data`	`void*`	Pointer to the actual memory block
`capacity`	`size_t`	Size of the buffer in bytes (fixed for all buffers in pool)
`in_use`	`bool`	Whether the buffer is currently assigned to an in-flight operation
`is_fixed`	`bool`	Whether this buffer is part of the registered fixed set
`fixed_index`	`uint16_t`	Index within the registered buffers (valid if `is_fixed` is true)
`zc_refcount`	`int`	Reference count for zero-copy operations (advanced feature; initially 0)
`next`	`struct io_buffer*`	Next pointer for free list linking

Common Pitfalls: File I/O

⚠️ Pitfall: Buffer Lifetime Error with Fixed Buffers

- **Description:** Reusing a fixed buffer for a new operation before the kernel has completed the previous read on that buffer.
- **Why it's wrong:** The kernel retains a reference to the buffer memory until the operation completes. Overwriting the buffer while the kernel is reading into it causes data corruption or crashes. This is a use-after-free scenario at the kernel level.
- **Fix:** Never mark a buffer as available in the pool until its corresponding CQE has been processed. The `io_op_context` should hold a reference to the buffer until completion. Use the `in_use` flag and ensure `release_buffer` is called only after the completion handler finishes.

⚠️ Pitfall: Misaligned Offsets for Direct I/O

- **Description:** Submitting a read operation with a file offset or buffer address that is not aligned to the underlying storage device's block size when using `O_DIRECT`.
- **Why it's wrong:** Direct I/O bypasses the kernel page cache and requires alignment to block boundaries (typically 512 bytes or 4KB). Misaligned operations fail with `EINVAL`.
- **Fix:** When opening files with `O_DIRECT`, ensure buffers are aligned to the logical block size (use `posix_memalign` for allocation) and offsets are multiples of that size. For general-purpose servers, consider avoiding `O_DIRECT` unless you specifically need to bypass cache and can control alignment.

⚠️ Pitfall: Handling `EAGAIN` /Short Reads

- **Description:** Assuming that a read operation will always read the full requested length, or that `EAGAIN` cannot occur for file I/O.
- **Why it's wrong:** While `EAGAIN` is rare for regular file reads (unless using non-blocking file descriptors), short reads can occur when reading near the end of a file. If the server logic expects a full buffer, it may mishandle partial data.
- **Fix:** Always check the `res` field in the CQE. For read operations, a positive `res` indicates the number of bytes successfully read. If `res` is less than the requested length, treat it as a short read (possibly end-of-file). If `res` is `-EAGAIN`, the operation should be retried (resubmitted). The completion handler should handle these cases appropriately, possibly by submitting a follow-up read for the remaining bytes.

⚠️ Pitfall: Forgetting to Advance the File Offset

- **Description:** Submitting multiple read operations on the same file descriptor without updating the offset, causing each read to start at the same position.
- **Why it's wrong:** The `offset` field in the SQE is used for the read operation. If not updated, concurrent reads will overwrite each other's buffers with the same data.
- **Fix:** For each independent read request, calculate the appropriate file offset (e.g., based on the client's request range) and set it in the SQE. If using `pread` semantics (offset per operation), ensure the offset is correctly passed; if using `read` semantics (sequential reading), track the current offset in the file context and update it after each completion.

⚠️ Pitfall: Ignoring CQE Overflow

- **Description:** Not checking for the `IORING_CQ_F_OVERFLOW` flag in the completion queue, which indicates that some completions were lost due to the CQ being full.
- **Why it's wrong:** Lost completions mean the server loses track of in-flight operations, leading to buffer leaks and hung requests.
- **Fix:** Monitor the overflow flag via `io_uring_cq_has_overflow(ring)`. If overflow occurs, the server must increase the CQ size (by recreating the ring with larger entries) or implement backpressure by slowing submission until completions are harvested more aggressively.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Buffer Allocation	`malloc` / `free` with alignment	Huge pages (`mmap` with `MAP_HUGETLB`) for reduced TLB misses
Buffer Registration	`io_uring_register` with `IORING_REGISTER_BUFFERS`	Dynamically grow/shrink registered buffer set based on load
File I/O Mode	Standard cached I/O (default open)	Direct I/O (`O_DIRECT`) for bypassing page cache (requires alignment)
Concurrency Control	Single-threaded event loop	Multi-threaded with separate rings per thread (future extension)

B. Recommended File/Module Structure

```
project-root/
├── src/
│   ├── core/
│   │   ├── engine.c             # io_uring engine core (Milestone 1)
│   │   └── engine.h
│   ├── file/
│   │   ├── file_server.c        # File I/O server logic (this milestone)
│   │   ├── file_server.h
│   │   ├── buffer_pool.c        # Buffer pool management
│   │   └── buffer_pool.h
│   ├── network/                 # Network server (Milestone 3)
│   └── main.c                   # Entry point, selects server mode
├── benchmarks/                  # Benchmark scripts
└── Makefile
```

C. Infrastructure Starter Code: Buffer Pool Manager

```c
/* buffer_pool.h */
#ifndef BUFFER_POOL_H
#define BUFFER_POOL_H

#include <stddef.h>
#include <stdbool.h>
#include <pthread.h>
#include <liburing.h>

struct io_buffer {
    void* data;
    size_t capacity;
    bool in_use;
    bool is_fixed;
    uint16_t fixed_index;
    int zc_refcount;              /* For zero-copy operations */
    struct io_buffer* next;       /* For free list */
};

struct buffer_pool {
    struct io_buffer* buffers;
    size_t count;
    size_t fixed_count;
    struct iovec* iovs;
    struct io_buffer* free_list;
    pthread_mutex_t lock;
};

/* Create a buffer pool with 'count' buffers of 'buffer_size' each.
 * If 'register_fixed' is true and 'ring' is not NULL, register the first
 * 'fixed_count' buffers with the io_uring instance as fixed buffers.
 * Returns NULL on failure.
 */
struct buffer_pool* create_buffer_pool(size_t buffer_size,
                                       size_t buffer_count,
                                       bool register_fixed,
                                       size_t fixed_count,
                                       struct io_uring* ring);
```

```
/* Destroy buffer pool and free all resources */
void destroy_buffer_pool(struct buffer_pool* pool);

/* Acquire an available buffer from the pool.
 * Returns NULL if no buffers available.
 * Sets *out_index to the fixed buffer index if applicable.
 */
struct io_buffer* acquire_buffer(struct buffer_pool* pool,
                                 uint16_t* out_index);

/* Release a buffer back to the pool.
 * Buffer must no longer be referenced by any in-flight operation.
 */
void release_buffer(struct buffer_pool* pool, struct io_buffer* buf);

#endif /* BUFFER_POOL_H */
```

```c
/* buffer_pool.c */

#include "buffer_pool.h"

#include <stdlib.h>

#include <string.h>

#include <errno.h>


#define ALIGN_UP(size, alignment) \

    (((size) + (alignment) - 1) & ~((alignment) - 1))


struct buffer_pool* create_buffer_pool(size_t buffer_size,

                                       size_t buffer_count,

                                       bool register_fixed,

                                       size_t fixed_count,

                                       struct io_uring* ring) {

    if (buffer_count == 0 || buffer_size == 0)

        return NULL;

    if (register_fixed && (ring == NULL || fixed_count == 0))

        return NULL;

    if (fixed_count > buffer_count)

        fixed_count = buffer_count;


    struct buffer_pool* pool = calloc(1, sizeof(*pool));

    if (!pool) return NULL;


    pool->count = buffer_count;

    pool->fixed_count = register_fixed ? fixed_count : 0;


    /* Allocate buffer descriptors */

    pool->buffers = calloc(buffer_count, sizeof(struct io_buffer));

    if (!pool->buffers) goto error;


    /* Allocate iovec array for registration (only needed for fixed buffers) */

    if (register_fixed) {

        pool->iovs = calloc(fixed_count, sizeof(struct iovec));

        if (!pool->iovs) goto error;

    }


    /* Kernel requires buffer memory to be page-aligned */

    size_t aligned_size = ALIGN_UP(buffer_size, 4096);


    for (size_t i = 0; i < buffer_count; i++) {
```

```
        struct io_buffer* buf = &pool->buffers[i];
        /* Use aligned_alloc for memory alignment (C11) */
        buf->data = aligned_alloc(4096, aligned_size);
        if (!buf->data) goto error;
        buf->capacity = aligned_size;
        buf->in_use = false;
        buf->is_fixed = (i < fixed_count);
        buf->fixed_index = (uint16_t)i;
        buf->zc_refcount = 0;
        buf->next = NULL;

        /* Add to free list */
        buf->next = pool->free_list;
        pool->free_list = buf;

        /* Fill iovec for fixed buffers */
        if (i < fixed_count) {
            pool->iovs[i].iov_base = buf->data;
            pool->iovs[i].iov_len = aligned_size;
        }
    }

    if (register_fixed) {
        int ret = io_uring_register_buffers(ring, pool->iovs, fixed_count);
        if (ret < 0) {
            /* Registration failed - fall back to non-fixed mode */
            fprintf(stderr, "Warning: Fixed buffer registration failed: %s\n",
                    strerror(-ret));
            pool->fixed_count = 0;
            /* Continue without fixed buffers */
        }
    }

    pthread_mutex_init(&pool->lock, NULL);
    return pool;

error:
    destroy_buffer_pool(pool);
    return NULL;
```

```
}

void destroy_buffer_pool(struct buffer_pool* pool) {
    if (!pool) return;

    pthread_mutex_lock(&pool->lock);
    if (pool->buffers) {
        for (size_t i = 0; i < pool->count; i++) {
            free(pool->buffers[i].data);
        }
        free(pool->buffers);
    }
    free(pool->iovs);
    pthread_mutex_unlock(&pool->lock);
    pthread_mutex_destroy(&pool->lock);
    free(pool);
}

struct io_buffer* acquire_buffer(struct buffer_pool* pool,
                                 uint16_t* out_index) {
    if (!pool) return NULL;

    pthread_mutex_lock(&pool->lock);
    struct io_buffer* buf = pool->free_list;
    if (buf) {
        pool->free_list = buf->next;
        buf->in_use = true;
        if (out_index && buf->is_fixed)
            *out_index = buf->fixed_index;
    }
    pthread_mutex_unlock(&pool->lock);
    return buf;
}

void release_buffer(struct buffer_pool* pool, struct io_buffer* buf) {
    if (!pool || !buf) return;

    pthread_mutex_lock(&pool->lock);
    buf->in_use = false;
    buf->zc_refcount = 0; /* Reset any zero-copy references */
```

```
    /* Add back to free list */
    buf->next = pool->free_list;
    pool->free_list = buf;
    pthread_mutex_unlock(&pool->lock);
}
```

D. Core Logic Skeleton Code: File Server Event Handler

```c
/* file_server.c */
#include "file_server.h"
#include "buffer_pool.h"
#include "engine.h"
#include <fcntl.h>
#include <unistd.h>

/* Context for a file read operation */
struct file_read_ctx {
    struct io_op_context op_ctx;   /* Base context */
    int client_fd;                 /* Client socket to respond to */
    int file_fd;                   /* File being read */
    off_t file_offset;             /* Offset in file */
    size_t bytes_to_read;          /* Total bytes requested */
    size_t bytes_read;             /* Bytes read so far */
    struct io_buffer* buffer;      /* Buffer holding data */
};

/* Submit an asynchronous file read operation */
static int submit_file_read(struct io_uring_engine* eng,
                            struct file_read_ctx* ctx) {
    // TODO 1: Get an available SQE using engine_get_sqe(eng)
    // TODO 2: If no SQE available, return -EBUSY (caller should retry)
    // TODO 3: Acquire a buffer from the pool using acquire_buffer
    //         If no buffer available, return -ENOBUFS
    // TODO 4: Configure the SQE:
    //         - opcode = IORING_OP_READ
    //         - fd = ctx->file_fd
    //         - addr = pointer to buffer memory (or 0 if using fixed buffer)
    //         - len = min(ctx->bytes_to_read - ctx->bytes_read, buffer_capacity)
    //         - offset = ctx->file_offset + ctx->bytes_read
    //         - flags = IOSQE_FIXED_FILE if using fixed buffer
    //         - user_data = ctx->op_ctx.id (correlation token)
    // TODO 5: If using fixed buffer, set buf_index to buffer's fixed_index
    // TODO 6: Store buffer pointer in ctx->buffer
    // TODO 7: Submit the SQE using engine_submit (or batch with others)
    // TODO 8: Return 0 on success, negative error code on failure
```

```c
    return 0;
}

/* Handle completion of a file read operation */
static void handle_file_read_completion(struct io_uring_engine* eng,
                                        struct io_uring_cqe* cqe) {
    // TODO 1: Look up the operation context using lookup_op_context with cqe->user_data
    // TODO 2: Cast to file_read_ctx* (verify op_type matches FILE_READ)
    // TODO 3: Check cqe->res for result:
    //         - If res > 0: successful read of 'res' bytes
    //         - If res == 0: end of file reached
    //         - If res == -EAGAIN: operation should be retried
    //         - If res < 0 (other error): handle error (log, close connection, etc.)
    // TODO 4: For successful read (res > 0):
    //         - Update ctx->bytes_read += res
    //         - If ctx->bytes_read < ctx->bytes_to_read:
    //             * Update file offset
    //             * Resubmit another read for remaining bytes (call submit_file_read)
    //         - Else (all bytes read):
    //             * Send data to client (could be async write via io_uring)
    //             * Release buffer using release_buffer
    //             * Free operation context
    // TODO 5: For -EAGAIN: resubmit the same read operation (no buffer change)
    // TODO 6: For fatal errors: log error, release buffer, free context, close client connection
    // TODO 7: Mark CQE as seen using io_uring_cqe_seen
}

/* Main event loop callback for file server */
void file_server_event_loop(struct event_loop* loop) {
    // TODO 1: Initialize buffer pool with fixed registration (e.g., 64 buffers of 64KB each)
    // TODO 2: Open the file to serve (keep file descriptor open for repeated reads)
    // TODO 3: Set up connection handling (accepting client connections - see network section)
    // TODO 4: For each client request (simplified: assume request contains offset and size):
    //         - Allocate a file_read_ctx via allocate_op_context
    //         - Fill in client_fd, file_fd, offset, bytes_to_read
    //         - Submit the first read via submit_file_read
    // TODO 5: In the main loop (engine_wait_and_process or engine_process_completions):
    //         - Process completions; for each CQE with opcode IORING_OP_READ,
```

```
    //           call handle_file_read_completion
    //         - Accept new client connections and create request contexts
    // TODO 6: On shutdown, clean up: close file, destroy buffer pool, free all contexts
}
```

E. Language-Specific Hints (C)

- **Memory Alignment:** Use `aligned_alloc(alignment, size)` (C11) or `posix_memalign` to ensure buffers meet kernel requirements for direct I/O or efficient fixed buffers.
- **Error Handling:** The `res` field in `struct io_uring_cqe` contains negative error codes on failure (e.g., `-EBADF` for bad file descriptor). Convert to positive for `strerror`.
- **Kernel Version:** Fixed buffers require kernel 5.1+. Check `io_uring_register` support at runtime; fall back to non-fixed buffers if unavailable.
- **File Descriptor Management:** Use `fcntl(fd, F_SETFL, O_NONBLOCK)` even for file descriptors? Not needed for `io_uring` operations—they are inherently asynchronous regardless of file descriptor flags.
- **Concurrency:** The buffer pool uses a mutex for thread safety. In the single-threaded event loop, you could omit the lock, but keeping it allows future multi-threaded extensions.

F. Milestone Checkpoint

Objective: Verify the file server can serve multiple concurrent file reads using `io_uring` with fixed buffers, demonstrating throughput improvement over synchronous I/O.

Verification Steps:

1. **Build and Start Server:**

```bash
make file_server
./file_server --port 8080 --file large_dataset.bin --buffer-size 65536 --buffer-count 64
```

2. **Generate Load:** Use a tool like `wrk` or `ab` to simulate multiple concurrent clients requesting different file chunks:

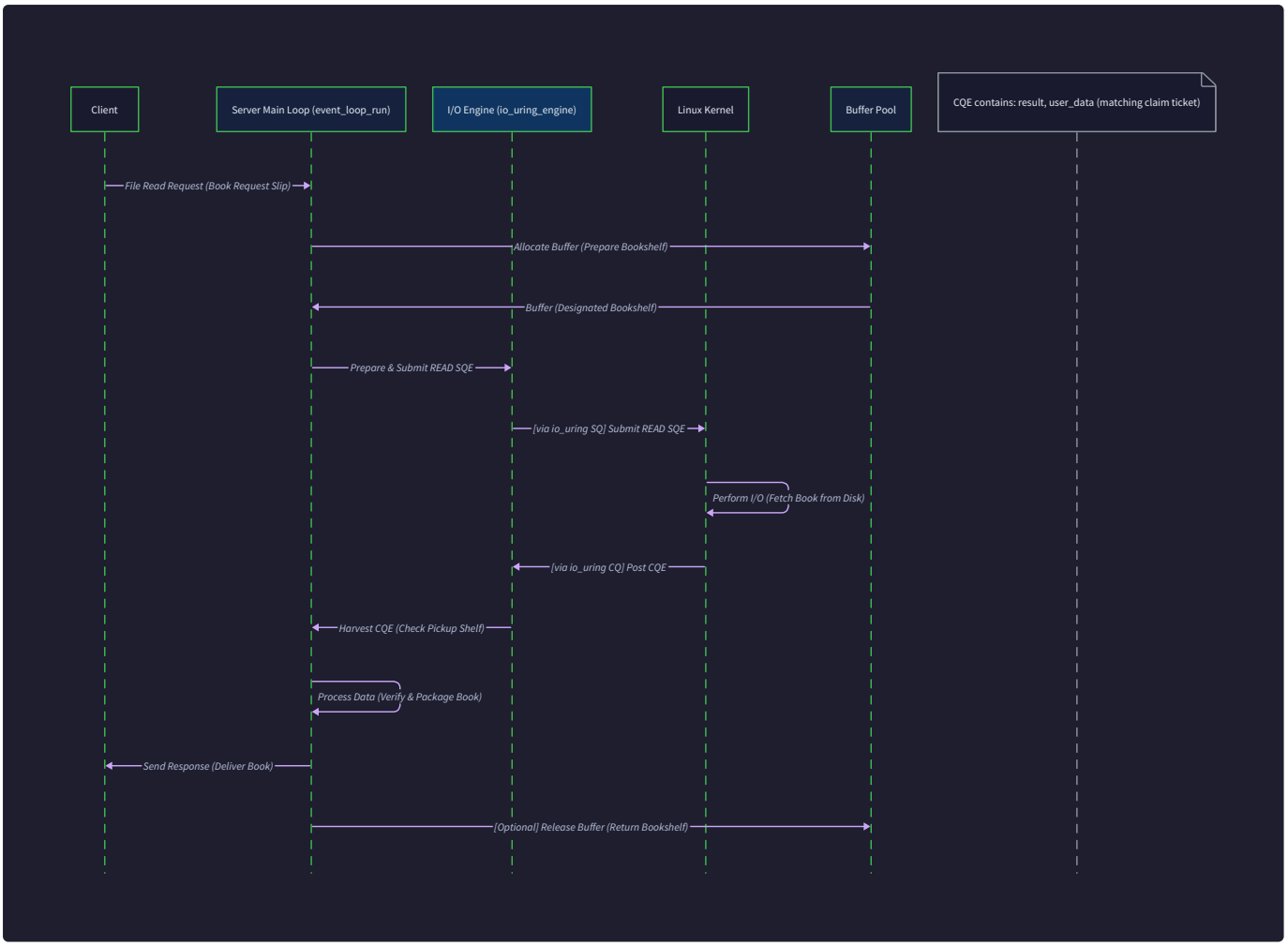
```bash
# Using curl in a loop to request different offsets
for i in {1..100}; do
  curl "http://localhost:8080/read?offset=$((i*8192))&size=65536" -o /dev/null &
done
```

3. **Monitor Performance:**
   - Check server logs for operations per second.
   - Use `strace -c -e io_uring_enter` to confirm batched submissions.
   - Monitor with `perf top` to see CPU usage patterns.
4. **Benchmark Against Synchronous Baseline:** Create a simple synchronous file server using `pread` in a thread pool. Compare throughput under concurrent load (e.g., 100 concurrent clients). Expect `io_uring` version to show:
   - Higher throughput (requests/second)
   - Lower system call count (visible in `strace`)
   - More consistent latency under load
5. **Expected Output Signs of Success:**
   - Server handles multiple concurrent connections without blocking.
   - Completion queue is consistently being processed.
   - No buffer allocation errors or fixed buffer registration failures.
   - Throughput scales with number of concurrent clients up to a point.

6. **Common Failure Modes and Diagnostics:**
   - **Server hangs:** Check if CQEs are being harvested (`engine_process_completions` called). Verify `io_uring_enter` is called with appropriate `min_complete`.
   - **Data corruption:** Verify buffer lifetime—add debugging to ensure buffers aren't reused before completion. Check offset calculations.
   - **Poor performance:** Ensure fixed buffers are actually being used (check `IOSQE_FIXED_FILE` flag). Monitor with `perf` to see if CPU is spent in memory allocation.

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Server crashes on `io_uring_register`	Kernel doesn't support fixed buffers or buffer alignment wrong	Check kernel version (`uname -r`). Verify buffer alignment with `(uintptr_t)buf->data % 4096`.	Use kernel ≥5.1. Ensure 4096-byte aligned buffers.
Read operations return `-EINVAL`	Misaligned offset or buffer for direct I/O	Check if file opened with `O_DIRECT`. Verify offset and buffer address are multiples of block size (typically 512 or 4096).	Avoid `O_DIRECT` or ensure alignment. Use `fstat` to get `st_blksize`.
Memory usage grows indefinitely	Buffers not being released	Add logging to `acquire_buffer`/`release_buffer`. Check that `release_buffer` is called for every buffer after CQE processing.	Ensure each completion handler calls `release_buffer`.
Throughput lower than synchronous version	Too small CQ size or insufficient batching	Check CQ overflow flag. Monitor batch size in `engine_submit`.	Increase CQ entries (2× SQ size). Implement opportunistic batching.
Fixed buffer operations fail with `-EINVAL`	Buffer index out of range	Verify `fixed_index` < registered buffer count. Check that `IOSQE_FIXED_FILE` flag is set.	Validate index before SQE submission. Use `io_uring_register_buffers` return count.



Component Design: Asynchronous Network Server

> **Milestone(s):** Milestone 3: Network Server

This component transforms our server into a high-performance network service capable of handling thousands of concurrent TCP connections entirely through `io_uring`. Unlike traditional event-driven servers using `epoll`, we leverage native asynchronous operations for every network action—accepting connections, reading requests, and writing responses—creating a unified I/O model that eliminates the fundamental performance bottleneck of repeated system calls.

Mental Model: The Concierge Desk and Waitstaff

Imagine a high-end hotel lobby. The **Concierge Desk** (multishot accept) operates continuously: the concierge stands ready with a stack of pre-numbered room keys (client sockets). When a guest arrives, the concierge immediately hands them a key without stopping to fill out paperwork for each arrival. This is `IORING_OP_ACCEPT` with `IORING_ACCEPT_MULTISHOT`—a single setup that automatically re-arms itself, producing a stream of completed connections.

Once guests have their keys, they proceed to the restaurant. **Waitstaff** (read/write operations) handle each table's needs asynchronously. A waiter takes a drink order (read request), submits it to the bar, and immediately moves to another table rather than waiting at the bar. When the drinks are ready (read completion), the waiter is notified and delivers them. Similarly, the waiter submits a food order (write request) to the kitchen and continues serving other tables until the food is ready for pickup (write completion).

This model eliminates idle waiting—the concierge never pauses between guests, and waiters never idle while orders are prepared. The kitchen's **ticket rail** (submission queue) continuously accepts new orders, and the **pickup counter** (completion queue) delivers finished items. The entire system operates on a single, efficient workflow managed by the `io_uring` engine, where every participant remains productive.

ADR: Using Multishot Operations

> **Decision: Use Multishot Accept for Connection Scalability, Defer Multishot Receive for Simplicity**
>
> - **Context**: Network servers must handle connection storms efficiently. Traditional `accept()` requires a system call per connection, creating overhead. io_uring's multishot feature allows a single Submission Queue Entry (SQE) to produce multiple Completion Queue Entries (CQEs) for repeated operations, dramatically reducing submission overhead.
> - **Options Considered**:
>   1. **Use multishot for both accept and receive operations**: Maximum performance, but increases complexity (requires careful re-arming and error handling for receive).
>   2. **Use multishot only for accept**: Good balance—captures the biggest win (connection acceptance) while keeping data path simpler.
>   3. **Avoid multishot entirely**: Simpler error handling, but sacrifices significant performance under high connection rates.
> - **Decision**: Implement multishot for `IORING_OP_ACCEPT` using the `IORING_ACCEPT_MULTISHOT` flag. For receive operations (`IORING_OP_READ` on sockets), use single-shot operations initially for simplicity, with a note on how to upgrade to multishot later.
> - **Rationale**: Connection acceptance is a homogeneous, high-frequency operation where multishot provides the greatest reduction in submission queue pressure—a single SQE can yield hundreds of accepted connections. Receive operations are more heterogeneous (different buffers, sizes, connection states), making multishot implementation more complex with marginal additional gain for our educational scope. We prioritize teaching the multishot concept where it has highest impact.
> - **Consequences**: The server will handle connection bursts efficiently, but each read operation requires a separate SQE submission. This maintains clarity in buffer management and connection state transitions for learners. Future optimization can replace single-shot reads with `IORING_OP_RECV` + `IORING_RECV_MULTISHOT` once the core patterns are understood.

Option	Pros	Cons	Chosen?
Multishot for accept & receive	Maximum submission reduction, optimal performance under all loads	Complex buffer management, harder error recovery, multishot receive requires kernel ≥5.19	No
Multishot accept only	**Captures biggest win, simpler data path, works on older kernels**	**Leaves some submission overhead on reads**	**Yes**
No multishot	Simplest error handling, straightforward state management	High submission overhead, poor scalability under connection bursts	No

Connection State Management

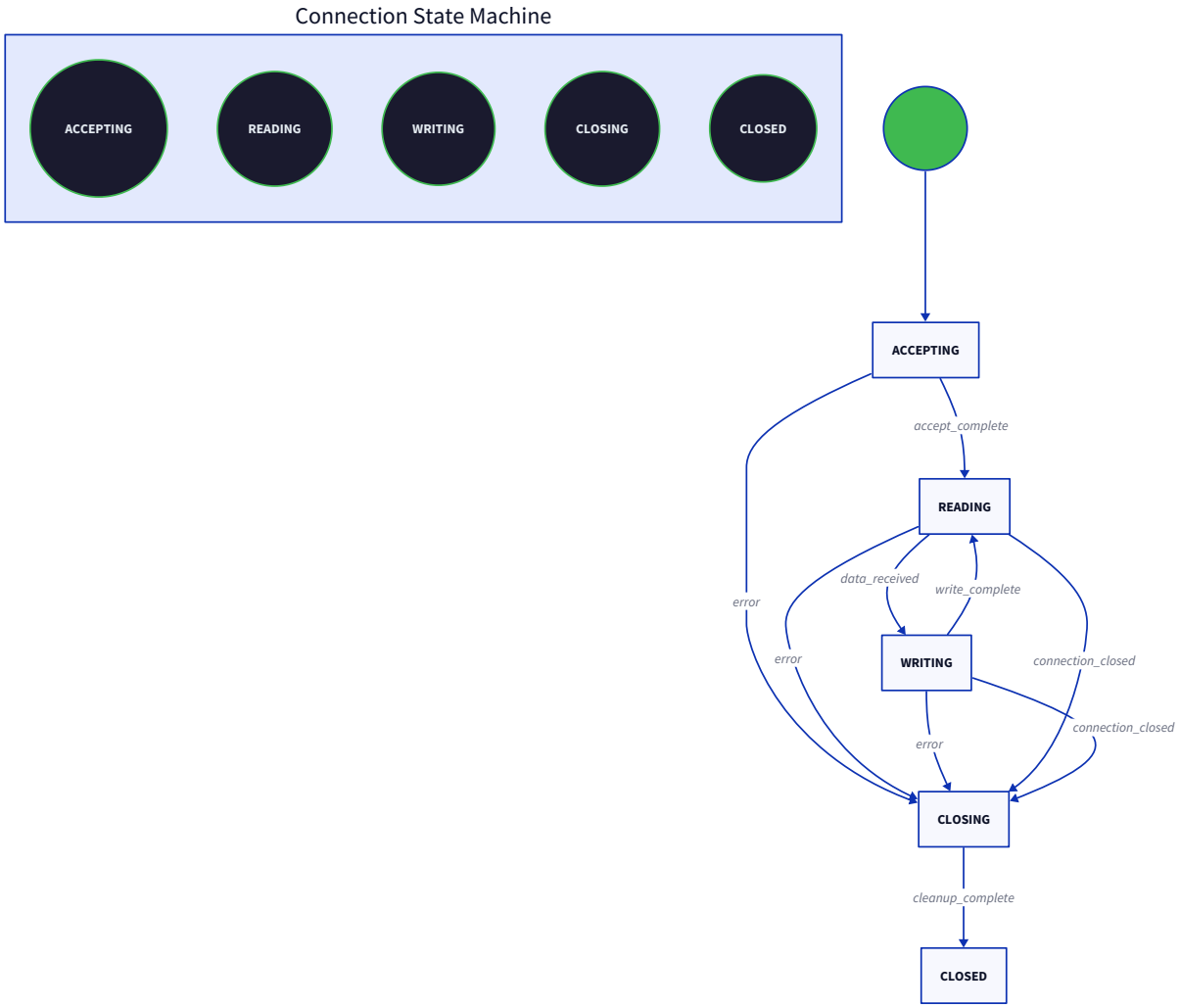
Each TCP connection progresses through a well-defined lifecycle, represented as a state machine within `struct connection_state`. This state machine ensures we correctly manage resource allocation, operation sequencing, and cleanup.

Connection State Machine

The following table defines all states, transitions, and the actions triggered by each completion:

Current State	Event (Completion Type)	Next State	Actions Taken
`CONN_ACCEPTING`	`IORING_OP_ACCEPT` success	`CONN_READING`	1. Allocate `connection_state` and `io_op_context` for new socket 2. Set socket to non-blocking mode 3. Submit `IORING_OP_READ` SQE for the connection's read buffer
`CONN_READING`	`IORING_OP_READ` success (bytes > 0)	`CONN_WRITING`	1. Process received data (e.g., echo back) 2. Submit `IORING_OP_WRITE` SQE with response 3. Optionally keep connection alive by transitioning back to `CONN_READING` after write completes
`CONN_READING`	`IORING_OP_READ` success (bytes = 0)	`CONN_CLOSING`	1. Peer closed connection gracefully 2. Cancel any pending operations 3. Submit `IORING_OP_CLOSE` SQE for the socket
`CONN_WRITING`	`IORING_OP_WRITE` success	`CONN_READING`	1. Check if keep-alive is desired 2. If yes, submit new `IORING_OP_READ` SQE for next request 3. If no, transition to `CONN_CLOSING` and submit close
`CONN_WRITING`	`IORING_OP_WRITE` partial success (short write)	`CONN_WRITING`	1. Adjust buffer offset and remaining length 2. Resubmit `IORING_OP_WRITE` SQE for remaining data
Any state except `CONN_CLOSED`	Any operation failure (`res < 0`)	`CONN_CLOSING`	1. Log error from `cqe->res` 2. Cancel all in-flight operations for this connection 3. Submit `IORING_OP_CLOSE` SQE
`CONN_CLOSING`	`IORING_OP_CLOSE` success	`CONN_CLOSED`	1. Release `connection_state` and associated buffers 2. Remove from connection tracking table
`CONN_CLOSING`	`IORING_OP_CLOSE` failure	`CONN_CLOSED`	1. Log critical error 2. Force-free resources (kernel may have closed socket anyway)

> **Design Insight**: The state machine is driven entirely by CQE completions—each I/O operation completion triggers a state transition. This creates a purely reactive system where the server only acts in response to kernel notifications, eliminating busy-waiting or polling.



Connection Data Structure

The `struct connection_state` (defined in the Data Model) holds all per-connection information:

Field	Type	Description
`fd`	`int`	Socket file descriptor returned by `accept()`. Primary handle for all connection operations.
`remote_addr`	`struct sockaddr_in`	Client's IP address and port. Used for logging and potential connection filtering.
`state`	`conn_state_t`	Current state from the state machine (`CONN_ACCEPTING`, `CONN_READING`, etc.).
`read_buffer`	`struct io_buffer*`	Buffer allocated from pool for receiving incoming data. Lifetime: allocated at accept completion, released after write completion or connection close.
`write_buffer`	`struct io_buffer*`	Buffer containing data to send. May point to same buffer as `read_buffer` for echo server.
`pending_ops`	`int`	Count of in-flight io_uring operations for this connection. Incremented on SQE submission, decremented on CQE processing. Used to ensure we don't close a connection with pending operations.
`last_active`	`time_t`	Timestamp of last I/O activity. Used for idle connection timeout (advanced feature).
`user_data`	`uint64_t`	Correlation token stored in SQE's `user_data` and returned in CQE. Typically encodes connection ID and operation type for lookup.
`user_ctx`	`void*`	Optional pointer to application-specific context (e.g., HTTP request parser state).

The connection table (a hash table or array indexed by connection ID) allows O(1) lookup from `user_data` when processing completions. Each `io_op_context` references its parent `connection_state` via the `connection` pointer, creating a clear ownership hierarchy: connection owns contexts, which reference buffers.

Common Pitfalls: Network Server

⚠️ Pitfall: Socket Leak on Connection Drop

- **Description**: When a client disconnects unexpectedly (RST packet), the server might not process the error completion promptly, leaving the socket descriptor open and consuming resources.
- **Why it's wrong**: Each leaked socket consumes a file descriptor (limited resource) and kernel memory. Under sustained attack, this can exhaust system resources and crash the server.
- **How to fix**: Always track `pending_ops` for each connection. In the completion handler for any operation, check `cqe->res` for errors like `-ECONNRESET`, `-EPIPE`, or `-ECONNABORTED`. Transition immediately to `CONN_CLOSING` state and submit a close operation. Implement a periodic cleanup sweep that forcibly closes connections with no activity beyond a timeout.

⚠️ Pitfall: Mishandling Multishot Re-submission

- **Description**: Forgetting that multishot accept automatically re-arms itself, and attempting to submit another accept SQE for the same listening socket, causing `-EBUSY` errors or duplicate connections.
- **Why it's wrong**: The kernel maintains the multishot SQE internally. Additional submissions waste SQEs and cause errors. If the multishot accept encounters an error (like `-EMFILE` from file descriptor exhaustion), it stops producing completions but the SQE remains consumed—requiring manual re-submission.
- **How to fix**: Submit exactly one multishot accept SQE during initialization. In its completion handler, check `cqe->res`. If negative (error), decide whether to retry based on error code (`-EMFILE` might be temporary). For fatal errors, fall back to single-shot accept. Never submit another accept SQE for the same socket while multishot is active.

⚠️ Pitfall: Head-of-Line Blocking in Linked Chains

- **Description**: Creating long chains of linked SQEs (read → process → write) where a slow operation (like disk I/O) blocks subsequent operations for the same connection, even though other connections' operations could proceed.
- **Why it's wrong**: Linked SQEs execute sequentially as a dependency chain. While this ensures ordering, it serializes operations that could otherwise be parallelized across the ring, reducing overall throughput.
- **How to fix**: Use linked chains judiciously—only for operations that truly depend on previous results (e.g., read must complete before we know what to write). For independent operations across connections, submit them as separate, unlinked SQEs to maximize parallelism. Consider using `IOSQE_IO_DRAIN` only when strict ordering across different file descriptors is required (rare in network servers).

⚠️ Pitfall: Forgetting to Advance CQ Tail After Multishot Batch

- **Description**: When processing a batch of completions from a multishot accept, calling `io_uring_cqe_seen()` or advancing the CQ ring tail once, but having processed multiple CQEs from the same SQE.
- **Why it's wrong**: Each CQE consumed must be marked as seen by advancing the tail pointer. If you process N completions but only advance once, the remaining N-1 completions remain visible, causing the same completions to be processed repeatedly in the next loop iteration.
- **How to fix**: For each CQE processed, regardless of whether it came from a multishot or single-shot SQE, advance the CQ tail. Use `io_uring_cqe_seen()` in a loop over all harvested CQEs. The helper function `io_uring_peek_batch_cqe()` can help batch this process.

⚠️ Pitfall: Buffer Lifetime Mismatch in Echo Server

- **Description**: In a simple echo server, using the same buffer for read and write operations without proper synchronization, leading to data corruption if a new read overwrites the buffer before the previous write completes.
- **Why it's wrong**: Async operations overlap in time. A write operation may still be in flight when a new read completion arrives, corrupting the in-transit response data.
- **How to fix**: Implement a buffer rotation scheme. Maintain two buffers per connection: while buffer A is being written, buffer B is used for reading. Track buffer ownership via connection state. Alternatively, use separate `read_buffer` and `write_buffer` pointers and copy data (simpler but less efficient). For zero-copy echo, implement reference counting on buffers.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Connection Tracking	Array indexed by connection ID (file descriptor)	Hash table keyed by `user_data` token for O(1) lookup
Buffer Management	Per-connection dedicated buffers (2 buffers per connection)	Shared buffer pool with reference counting for zero-copy echo
Event Loop	Single-threaded with blocking `io_uring_enter` calls	Kernel polling (`IORING_SETUP_SQPOLL`) to eliminate syscalls entirely
Protocol	Simple text echo (read → write same data)	HTTP/1.1 request parsing with pipelining support

Recommended File/Module Structure

```
src/
├── core/
│   ├── engine.c             # io_uring engine core (from Milestone 1)
│   ├── engine.h
│   ├── event_loop.c         # Main event loop (from Milestone 1)
│   └── event_loop.h
├── network/
│   ├── connection.c         # Connection state management (THIS COMPONENT)
│   ├── connection.h
│   ├── server.c             # Network server setup and multishot accept
│   ├── server.h
│   └── protocol/            # Optional: HTTP, custom protocols
│       └── echo.c           # Simple echo protocol handler
├── buffers/
│   ├── pool.c               # Buffer pool (from Milestone 2)
│   └── pool.h
├── utils/
│   ├── context_table.c      # Operation context lookup table
│   └── context_table.h
└── main.c                   # Application entry point
```

Infrastructure Starter Code: Connection State Manager

```c
/* connection.h */

#ifndef CONNECTION_H

#define CONNECTION_H


#include <netinet/in.h>

#include <stdbool.h>

#include <stdint.h>

#include <time.h>


/* Connection states as defined in naming conventions */

typedef enum {

    CONN_ACCEPTING = 0,

    CONN_READING,

    CONN_WRITING,

    CONN_CLOSING,

    CONN_CLOSED

} conn_state_t;


struct io_buffer;  /* Forward declaration */


/* Connection state structure - matches naming conventions exactly */

struct connection_state {

    int fd;

    struct sockaddr_in remote_addr;

    conn_state_t state;

    struct io_buffer* read_buffer;

    struct io_buffer* write_buffer;

    int pending_ops;

    time_t last_active;

    uint64_t user_data;

    void* user_ctx;

};


/* Connection table for O(1) lookup by user_data */

struct connection_table {

    struct connection_state** slots;

    uint64_t mask;          /* Table size minus one (power of two) */

    uint64_t count;         /* Active connections */
```

```
};

/* Connection manager API */
struct connection_table* conn_table_create(size_t size);
void conn_table_destroy(struct connection_table* table);
struct connection_state* conn_create(int fd, struct sockaddr_in* addr);
void conn_destroy(struct connection_table* table, struct connection_state* conn);
struct connection_state* conn_lookup(struct connection_table* table, uint64_t user_data);
bool conn_register(struct connection_table* table, struct connection_state* conn);
void conn_unregister(struct connection_table* table, struct connection_state* conn);

/* State transitions */
void conn_transition_to_reading(struct connection_state* conn);
void conn_transition_to_writing(struct connection_state* conn);
void conn_transition_to_closing(struct connection_state* conn);
void conn_mark_active(struct connection_state* conn);

#endif /* CONNECTION_H */
```

```c
/* connection.c */

#include "connection.h"

#include <stdlib.h>

#include <string.h>

#include <errno.h>

#include <unistd.h>

#include <sys/socket.h>


/* Create connection table with power-of-two size */

struct connection_table* conn_table_create(size_t size) {

    /* Round up to next power of two */

    size_t actual_size = 1;

    while (actual_size < size) actual_size <<= 1;



    struct connection_table* table = malloc(sizeof(struct connection_table));

    if (!table) return NULL;



    table->slots = calloc(actual_size, sizeof(struct connection_state*));

    if (!table->slots) {

        free(table);

        return NULL;

    }



    table->mask = actual_size - 1;

    table->count = 0;

    return table;

}


/* Destroy connection table and all connections */

void conn_table_destroy(struct connection_table* table) {

    if (!table) return;



    for (uint64_t i = 0; i <= table->mask; i++) {

        struct connection_state* conn = table->slots[i];

        if (conn) {

            if (conn->fd >= 0) close(conn->fd);

            free(conn);

        }
```

```c
    }

    free(table->slots);
    free(table);
}

/* Create new connection state (does NOT register in table) */
struct connection_state* conn_create(int fd, struct sockaddr_in* addr) {
    struct connection_state* conn = calloc(1, sizeof(struct connection_state));
    if (!conn) return NULL;

    conn->fd = fd;
    if (addr) memcpy(&conn->remote_addr, addr, sizeof(struct sockaddr_in));
    conn->state = CONN_ACCEPTING;
    conn->read_buffer = NULL;
    conn->write_buffer = NULL;
    conn->pending_ops = 0;
    conn->last_active = time(NULL);
    conn->user_data = 0;  /* Will be set when registered */
    conn->user_ctx = NULL;

    return conn;
}

/* Destroy single connection (assumes already unregistered) */
void conn_destroy(struct connection_table* table, struct connection_state* conn) {
    if (!conn) return;

    /* Close socket if not already closed */
    if (conn->fd >= 0) {
        close(conn->fd);
        conn->fd = -1;
    }

    /* Free buffers through buffer pool API (not shown here) */
    /* buffer_pool_release(conn->read_buffer); */
    /* buffer_pool_release(conn->write_buffer); */
```

```
    free(conn);
}

/* Lookup connection by user_data token */
struct connection_state* conn_lookup(struct connection_table* table, uint64_t user_data) {
    if (!table) return NULL;

    uint64_t idx = user_data & table->mask;

    /* Linear probing for collision resolution */
    while (table->slots[idx] != NULL) {
        if (table->slots[idx]->user_data == user_data) {
            return table->slots[idx];
        }
        idx = (idx + 1) & table->mask;
    }

    return NULL;
}

/* Register connection in table, assigning unique user_data */
bool conn_register(struct connection_table* table, struct connection_state* conn) {
    if (!table || !conn) return false;

    /* Find empty slot using linear probing */
    uint64_t idx = conn->fd & table->mask;  /* Use fd as hash seed */
    while (table->slots[idx] != NULL) {
        idx = (idx + 1) & table->mask;
    }

    /* Create user_data token: upper 32 bits = index, lower 32 bits = fd */
    conn->user_data = ((idx << 32) | (conn->fd & 0xFFFFFFFF));
    table->slots[idx] = conn;
    table->count++;

    return true;
}
```

```
/* Unregister connection from table */
void conn_unregister(struct connection_table* table, struct connection_state* conn) {
    if (!table || !conn) return;

    uint64_t idx = (conn->user_data >> 32) & table->mask;
    if (table->slots[idx] == conn) {
        table->slots[idx] = NULL;
        table->count--;

        /* Re-hash subsequent entries that may have been placed here due to probing */
        uint64_t next = (idx + 1) & table->mask;
        while (table->slots[next] != NULL) {
            struct connection_state* rehash = table->slots[next];
            table->slots[next] = NULL;
            table->count--;
            conn_register(table, rehash);
            next = (next + 1) & table->mask;
        }
    }
}

/* State transition helpers */
void conn_transition_to_reading(struct connection_state* conn) {
    conn->state = CONN_READING;
    conn_mark_active(conn);
}

void conn_transition_to_writing(struct connection_state* conn) {
    conn->state = CONN_WRITING;
    conn_mark_active(conn);
}

void conn_transition_to_closing(struct connection_state* conn) {
    conn->state = CONN_CLOSING;
    /* Cancel any pending operations here */
}

void conn_mark_active(struct connection_state* conn) {
    conn->last_active = time(NULL);
```

```
}
```

Core Logic Skeleton: Network Event Handler

```c
/* server.c - Network server core logic */

#include "server.h"

#include "connection.h"

#include "engine.h"

#include <liburing.h>

#include <string.h>

#include <errno.h>


/* Server global state */

struct network_server {

    struct io_uring_engine* engine;

    struct connection_table* conn_table;

    int listen_fd;

    struct sockaddr_in listen_addr;

    bool running;

};


/* Submit a multishot accept SQE for the listening socket */

static int submit_accept_multishot(struct network_server* server) {

    // TODO 1: Get an SQE from the engine using engine_get_sqe()

    // TODO 2: Set opcode to IORING_OP_ACCEPT

    // TODO 3: Set fd to server->listen_fd

    // TODO 4: Set addr pointer to a sockaddr_in structure for the client address

    // TODO 5: Set addr_len pointer to sizeof(sockaddr_in)

    // TODO 6: Set flags to IORING_ACCEPT_MULTISHOT

    // TODO 7: Set user_data to a special token identifying this as the listen socket (e.g., 0)

    // TODO 8: Submit the SQE using engine_submit()

    // TODO 9: Return 0 on success, -1 on error

}


/* Submit a read SQE for a connection */

static int submit_connection_read(struct network_server* server,

                                  struct connection_state* conn) {

    // TODO 1: Ensure conn->read_buffer is allocated (acquire from pool)

    // TODO 2: Get an SQE from engine using engine_get_sqe()

    // TODO 3: Set opcode to IORING_OP_READ

    // TODO 4: Set fd to conn->fd
```

```
    // TODO 5: Set addr to conn->read_buffer->data
    // TODO 6: Set len to conn->read_buffer->capacity
    // TODO 7: Set offset to 0 (not used for sockets)
    // TODO 8: Set user_data to conn->user_data with an operation type flag (e.g., 0x1 for read)
    // TODO 9: Increment conn->pending_ops
    // TODO 10: Return 0 on success, -1 on error
}

/* Submit a write SQE for a connection */
static int submit_connection_write(struct network_server* server,
                                   struct connection_state* conn,
                                   const void* data, size_t len) {
    // TODO 1: Ensure conn->write_buffer is allocated and contains data
    // TODO 2: Get an SQE from engine using engine_get_sqe()
    // TODO 3: Set opcode to IORING_OP_WRITE
    // TODO 4: Set fd to conn->fd
    // TODO 5: Set addr to data pointer
    // TODO 6: Set len to data length
    // TODO 7: Set offset to 0 (not used for sockets)
    // TODO 8: Set user_data to conn->user_data with operation type flag (e.g., 0x2 for write)
    // TODO 9: Increment conn->pending_ops
    // TODO 10: Return 0 on success, -1 on error
}

/* Process completion for an accept operation */
static void handle_accept_completion(struct network_server* server,
                                     struct io_uring_cqe* cqe) {
    // TODO 1: Check cqe->res for errors
    // TODO 2: If error, log and decide whether to resubmit multishot accept
    // TODO 3: On success, cqe->res contains the new socket fd
    // TODO 4: Create new connection_state using conn_create()
    // TODO 5: Register connection in connection table
    // TODO 6: Set socket to non-blocking mode using fcntl()
    // TODO 7: Submit initial read for this connection using submit_connection_read()
    // TODO 8: Update server statistics
}

/* Process completion for a read operation */
```

```c
static void handle_read_completion(struct network_server* server,
                                   struct connection_state* conn,
                                   struct io_uring_cqe* cqe) {
    // TODO 1: Decrement conn->pending_ops
    // TODO 2: Check cqe->res for errors or connection close (<= 0)
    // TODO 3: If error/close, transition to CONN_CLOSING and submit close
    // TODO 4: On success, cqe->res contains bytes read
    // TODO 5: Process the received data (e.g., for echo server, prepare to write same data back)
    // TODO 6: Transition to CONN_WRITING state
    // TODO 7: Submit write operation with the received data using submit_connection_write()
    // TODO 8: Mark connection as active
}

/* Process completion for a write operation */
static void handle_write_completion(struct network_server* server,
                                    struct connection_state* conn,
                                    struct io_uring_cqe* cqe) {
    // TODO 1: Decrement conn->pending_ops
    // TODO 2: Check cqe->res for errors
    // TODO 3: If error, transition to CONN_CLOSING and submit close
    // TODO 4: If partial write (cqe->res < expected), resubmit write for remaining data
    // TODO 5: On full write completion, decide whether to keep connection alive
    // TODO 6: If keeping alive, transition back to CONN_READING and submit new read
    // TODO 7: Otherwise, transition to CONN_CLOSING and submit close
    // TODO 8: Mark connection as active
}

/* Main network event processing loop */
int network_server_run(struct network_server* server) {
    if (!server || !server->engine) return -1;


    // TODO 1: Submit initial multishot accept SQE
    // TODO 2: Enter main loop while server->running
    // TODO 3: In each iteration, call engine_wait_and_process() to get completions
    // TODO 4: For each CQE:
    //   a. Extract operation type from user_data (accept/read/write/close)
    //   b. Lookup connection using conn_lookup() for connection operations
    //   c. Call appropriate handler based on operation type
```

```
    //   d. Mark CQE as seen using io_uring_cqe_seen()
    // TODO 5: Handle idle connection timeout (advanced: sweep conn table periodically)
    // TODO 6: Clean up on exit: close all connections, destroy table
}

/* Create and bind listening socket */
static int create_listen_socket(int port) {
    // TODO 1: Create socket with AF_INET, SOCK_STREAM
    // TODO 2: Set SO_REUSEADDR option
    // TODO 3: Bind to INADDR_ANY and specified port
    // TODO 4: Listen with a large backlog (e.g., 4096)
    // TODO 5: Set socket to non-blocking mode
    // TODO 6: Return socket fd or -1 on error
}

/* Public API: Create network server */
struct network_server* network_server_create(int port,
                                            struct io_uring_engine* engine) {
    // TODO 1: Allocate server structure
    // TODO 2: Create connection table with initial size (e.g., 65536)
    // TODO 3: Create and bind listening socket
    // TODO 4: Initialize server fields
    // TODO 5: Return server instance
}
```

Language-Specific Hints

- **Socket Options**: Use `fcntl(fd, F_SETFL, O_NONBLOCK)` to set sockets to non-blocking mode, though `io_uring` operations work on both blocking and non-blocking sockets. Non-blocking ensures fallback code (like `accept()`) won't block if needed.
- **Memory Barriers**: When accessing `io_uring` ring buffers directly (not using liburing helpers), use compiler barriers like `asm volatile("" ::: "memory")` or C11 `atomic_signal_fence()` to ensure the compiler doesn't reorder reads/writes of `head` and `tail` pointers.
- **Error Codes**: Familiarize yourself with socket-specific error codes: `-ECONNRESET` (connection reset by peer), `-EPIPE` (broken pipe), `-ECONNABORTED` (software caused connection abort). Handle these gracefully by closing the connection.
- **Multishot Limitations**: On kernels before 5.19, `IORING_ACCEPT_MULTISHOT` may not be available. Check feature support at runtime or provide a fallback to single-shot accept.

Milestone Checkpoint

To verify your network server implementation:

1. **Start the server**: `./server --port 8080 --protocol echo`
2. **Test basic connectivity**: `echo "hello" | nc localhost 8080` should echo back "hello".
3. **Verify multishot accept is working**: Use `strace -e io_uring_enter ./server` and observe that `io_uring_enter` calls are infrequent despite many connections.
4. **Load test with thousands of connections**:

```bash
# Using wrk for HTTP or custom load tester for echo

wrk -t12 -c4000 -d30s http://localhost:8080/
```

Monitor with `ss -tlnp | grep 8080 | wc -l` to see connection count. 5. **Check for resource leaks**: Run server under `valgrind --leak-check=full` or monitor `/proc/<pid>/fd` count during load test—it should stabilize, not grow indefinitely. 6. **Expected behavior**: The server should handle at least 10,000 concurrent connections on a machine with sufficient file descriptor limits (set via `ulimit -n 100000`). CPU usage should be moderate (under 100% of one core for echo).

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Server accepts few connections then stops	Multishot accept stopped due to `-EMFILE` (file descriptor exhaustion)	Check `ulimit -n`, monitor `/proc/<pid>/limits`. Look for `cqe->res == -EMFILE` in accept completions.	Increase file descriptor limit, implement connection limiting, or close idle connections.
High memory usage with many connections	Buffer per connection strategy without pooling	Check `pmap -x <pid>` or `/proc/<pid>/smaps`. Each connection holds 2 buffers (read/write).	Implement buffer pooling with reference counting. Use smaller buffers for small messages.
Connections hang in `CONN_READING` state	Read completion never arrives (client never sends data)	Add timeout tracking in connection state. Use `last_active` to close idle connections.	Implement idle timeout: sweep connection table periodically, close connections where `time(NULL) - last_active > TIMEOUT`.
`io_uring_enter` returns `-EBUSY` frequently	Submission queue full because completions aren't being processed fast enough	Check CQ tail advancement. Monitor SQ and CQ ring pointers in debug output.	Increase SQ size, process completions more aggressively, or use `IORING_SETUP_SQPOLL` to offload submission to kernel thread.
Data corruption in echo responses	Buffer reuse while write still in flight	Add debug prints showing buffer contents before write and after read. Use separate read/write buffers.	Implement buffer rotation: two buffers per connection, toggle between them. Or use reference counting on buffers.

Component Design: Advanced Features & Optimization

> **Milestone(s):** Milestone 4: Zero-copy, Linked Ops, and Benchmarks

This component builds upon the foundational `io_uring` server to implement advanced features that unlock maximum performance and sophisticated operation orchestration. While the previous components established basic asynchronous I/O patterns, this section focuses on eliminating data copies, creating atomic operation chains, and enforcing ordering guarantees—capabilities that distinguish `io_uring` from previous I/O models. The design decisions here directly impact throughput, latency, and resource utilization in high-demand scenarios.

Mental Model: The Assembly Line and Conveyor Belts

Imagine a factory assembly line for processing customer orders. Each order requires multiple sequential steps: unpacking the raw materials, assembling components, quality checking, and packaging for shipment. This mental model helps visualize three advanced `io_uring` concepts:

1. **Linked SQEs as Assembly Stations**: Each `IOSQE_IO_LINK` chain represents an order moving through consecutive workstations. The first SQE (unpack) must complete before the second (assemble) begins, and the entire chain either succeeds together or fails together—much like an assembly line that halts if any station encounters a defect.

2. **Zero-Copy as Direct Conveyor Transfer**: Traditional I/O is like moving items between stations by wrapping them in boxes (kernel buffers), transporting the boxes, then unwrapping at the destination. Zero-copy operations (`IORING_OP_SEND_ZC`) eliminate this wrapping—instead, items move directly on a conveyor belt that passes through both stations without intermediate handling, dramatically reducing labor (CPU cycles) and time.

3. **IO_DRAIN as Traffic Lights**: Occasionally, a special order requires exclusive access to the assembly line, perhaps because it uses hazardous materials. The `IOSQE_IO_DRAIN` flag acts like a red traffic light for subsequent operations, ensuring they wait until all prior work completes, preventing dangerous interactions. Use it sparingly to avoid gridlock in an otherwise parallel system.

This mental model emphasizes that while `io_uring` excels at parallelism, it also provides precise control mechanisms for when operations must be sequential or when data movement must be minimized—tools that become essential for building production-grade systems.

ADR: Zero-Copy Send Implementation

> **Decision: Managed Ownership with Reference-Counted Zero-Copy Buffers**
>
> - **Context**: Zero-copy sends (`IORING_OP_SEND_ZC`) allow network data transmission without copying between kernel and userspace, but require buffers to remain valid and unmodified until the kernel releases them. The kernel signals release via a completion queue entry (`CQE`) with the `IORING_CQE_F_NOTIF` flag. We need a strategy to manage buffer lifecycle during this uncertain period while maintaining high performance.
> - **Options Considered**:
>   1. **Static Buffer Allocation**: Pre-allocate a fixed pool of zero-copy buffers, never reuse them until kernel notification arrives, and block if pool is exhausted.
>   2. **Copy Fallback with Dynamic Allocation**: Attempt zero-copy first, but if buffers are unavailable, fall back to copying with dynamically allocated buffers for that request.
>   3. **Managed Ownership with Reference Counting**: Track kernel references to each buffer via a reference counter, allowing reuse of buffers not currently held by the kernel, and queuing requests when all buffers are referenced.
> - **Decision**: Option 3 (Managed Ownership with Reference Counting). Each `struct io_buffer` gains a `zc_refcount` field tracking active kernel references. Buffers are acquired from a pool, their refcount incremented on zero-copy submission, and decremented when the notification CQE arrives. Buffers with `zc_refcount == 0` are immediately reusable.
> - **Rationale**: This approach maximizes zero-copy utilization without imposing hard limits on concurrent operations. Unlike static allocation, it doesn't waste memory during low activity. Unlike copy fallback, it maintains zero-copy benefits under sustained load. The reference counting model aligns naturally with the kernel's notification mechanism and integrates cleanly with existing buffer pool management.
> - **Consequences**: Introduces slight overhead for refcount management and requires careful handling of notification CQEs. The buffer pool must be sized appropriately for expected concurrency, but can dynamically adapt. If all buffers are referenced, new sends must wait (or could temporarily fall back to copying, though we choose simplicity and wait).

Zero-Copy Buffer Management Strategy Comparison:

Option	Pros	Cons	Chosen?
Static Buffer Allocation	Simple implementation, predictable memory footprint	Wastes memory during low activity, hard concurrency limit, may block indefinitely	❌
Copy Fallback with Dynamic Allocation	Never blocks, good for mixed workloads	Loses zero-copy benefits under load, complexity of two code paths, allocation overhead	❌
Managed Ownership with Reference Counting	Maximizes zero-copy utilization, dynamic adaptation, clean integration	Refcount overhead, requires careful notification handling, potential waiting	✅

The key insight is that zero-copy operations trade CPU cycles for extended buffer residency—we must design our buffer management to accommodate this trade-off gracefully rather than fighting it.

ADR: Chaining Operations with Linked SQEs

> **Decision: Selective Use of Linked SQEs for Atomic Multi-Stage Operations**
>
> - **Context**: Some server operations naturally decompose into sequential I/O steps: read request → process → write response, or read file chunk → transform → send. We can coordinate these steps either manually (submitting subsequent operations in completion handlers) or via linked SQEs (`IOSQE_IO_LINK`), where the kernel automatically sequences operations and fails the entire chain if any step fails.
> - **Options Considered**:
>   1. **Manual Chaining in Completion Handlers**: Process each CQE and programmatically submit the next SQE. Each operation has independent completion.
>   2. **Full Linked Chains**: Use `IOSQE_IO_LINK` for all multi-step operations, letting the kernel manage sequencing and atomicity.
>   3. **Hybrid Approach**: Use linked chains only for operations requiring atomicity (all-or-nothing semantics) or strict kernel-managed sequencing; use manual chaining for operations where steps can be independently retried or where flexibility is needed.
> - **Decision**: Option 3 (Hybrid Approach). Linked chains are used for request/response cycles where a read must complete before the corresponding write is attempted, and atomicity is valuable (e.g., a malformed request shouldn't trigger a write). Manual chaining is used for operations where intermediate processing is complex or where we want metrics per step.
> - **Rationale**: Linked SQEs reduce submission overhead (multiple operations submitted as one batch) and provide atomicity, but they also introduce rigidity—the entire chain fails if any link fails, and all completions arrive at once, making intermediate processing awkward. Manual chaining offers more flexibility for error handling, logging, and complex business logic between steps. The hybrid approach gives us both: atomicity where needed, flexibility where appropriate.
> - **Consequences**: Requires clear criteria for when to use each pattern. Developers must understand that linked chain failures require cleanup of the entire operation context, not just individual steps. Debugging linked chains can be trickier since all CQEs arrive together after the final link completes.

Linked vs. Manual Chaining Comparison:

Pattern	Pros	Cons	Best For
Linked SQEs (`IOSQE_IO_LINK`)	Atomic all-or-nothing execution, reduced submission overhead (single batch), kernel-managed sequencing	Rigid: all steps must be predefined, entire chain fails on any error, all completions arrive at end	Simple read→write pipelines, operations requiring strict ordering without intermediate processing
Manual Chaining	Flexible error handling per step, can insert business logic between steps, independent retry of failed steps	Higher submission overhead (multiple syscalls), programmer must manage sequencing, no atomic guarantees	Complex workflows, operations requiring transformation between steps, debugging/telemetry between stages

A critical design principle emerges: **use linked SQEs when the kernel can enforce correctness more simply than your application code**. For straightforward I/O pipelines, let the kernel handle sequencing; for complex logic, keep control in userspace.

Common Pitfalls: Advanced Features

⚠️ Pitfall: Deadlocks with IOSQE_IO_DRAIN

- **Description**: Using `IOSQE_IO_DRAIN` on an SQE that depends on a completion that will never arrive (e.g., a drained write waiting for a read that's stuck behind the drain).
- **Why it's wrong**: `IOSQE_IO_DRAIN` forces all previously submitted SQEs to complete before this one starts, and prevents submission of new SQEs until it completes. If circular dependencies exist, the system deadlocks.
- **Fix**: Use `IOSQE_IO_DRAIN` only for operations that truly must not reorder with *already submitted* operations, and ensure no dependency loops. Typically used for operations like `fsync` that must see all prior writes. Avoid using it for normal data pipeline operations.

⚠️ Pitfall: Resource Leaks on Linked Chain Failure

- **Description**: When a linked chain fails at step 2/3, resources allocated for step 1 (like buffers) may not be properly released because the application expects to clean up after the final completion.
- **Why it's wrong**: The kernel cancels the remaining links in the chain but still provides a CQE for the failed link. If cleanup logic only runs on successful chain completion, resources leak.
- **Fix**: In your CQE handler, check `cqe->res` for errors on *any* linked operation. Clean up all resources associated with the chain's `user_data` context regardless of which link failed. Design your `io_op_context` to hold all resources needed for the entire chain.

⚠️ Pfitfall: Misusing Zero-Copy Buffer Flags

- **Description**: Assuming `IORING_OP_SEND_ZC` buffers can be immediately reused after submission, or mixing zero-copy and regular buffers in the same pool without proper tracking.
- **Why it's wrong**: Zero-copy buffers remain owned by the kernel until a notification CQE arrives. Reusing them earlier corrupts in-flight data. The `IORING_CQE_F_NOTIF` flag is only set on the *notification* CQE, not the send completion CQE.
- **Fix**: Implement clear state tracking in `struct io_buffer`: `zc_refcount > 0` means kernel holds references. Only release buffers to the pool when refcount reaches zero. Process notification CQEs separately from regular completions to decrement refcounts.

⚠️ Pitfall: Head-of-Line Blocking in Long Chains

- **Description**: Creating excessively long linked chains (e.g., read$\rightarrow$process$_1\rightarrow$process$_2\rightarrow$process$_3\rightarrow$write) where a slow intermediate step blocks all subsequent operations, even unrelated ones, because the ring cannot process other SQEs until the chain advances.
- **Why it's wrong**: While `io_uring` excels at parallelism, a single long serial chain monopolizes submission slots and can stall the entire ring, especially if intermediate steps involve CPU-bound processing.
- **Fix**: Break long chains into smaller segments (2-3 operations max). Use manual chaining for CPU-intensive steps between I/O operations. Consider using multiple rings for independent request flows if truly long pipelines are necessary.

⚠️ Pitfall: Ignoring Zero-Copy Notification CQEs

- **Description**: Failing to harvest and process CQEs with the `IORING_CQE_F_NOTIF` flag, leading to ever-increasing kernel buffer references and eventual resource exhaustion.
- **Why it's wrong**: Zero-copy notifications are separate from operation completions. Each `IORING_OP_SEND_ZC` produces *two* CQEs: one for send completion, one for buffer release notification. Missing notifications means buffers never get released back to the pool.
- **Fix**: In your completion processing loop, always check `cqe->flags & IORING_CQE_F_NOTIF`. Route these to a special handler that decrements buffer refcounts without treating them as operation completions. Consider maintaining a separate count of pending notifications for debugging.

These pitfalls highlight a theme: advanced features shift complexity from the kernel to the application's resource management logic. The power of zero-copy and linked operations comes with the responsibility of meticulous lifetime tracking.

Implementation Guidance

This guidance provides concrete building blocks for implementing Milestone 4's advanced features. The focus is on extending existing components with zero-copy support, linked operations, and ordering controls.

Technology Recommendations:

Component	Simple Option	Advanced Option
Zero-copy buffer tracking	Reference counting in `io_buffer` with pool exhaustion handling	Lock-free buffer rings with batched notification processing
Linked operation management	Single linked chain per request with full cleanup on error	Chain checkpointing with partial rollback capabilities
Ordering guarantees	Selective `IOSQE_IO_DRAIN` for critical operations (fsync)	Multiple priority rings with controlled cross-ring dependencies

Recommended File/Module Structure:

```
project-root/
├── src/
│   ├── core/
│   │   ├── engine.c             # Extended with zero-copy completion handling
│   │   ├── engine.h
│   │   ├── buffer_pool.c        # Enhanced with zc_refcount management
│   │   └── buffer_pool.h
│   ├── network/
│   │   ├── server.c             # Zero-copy send and linked accept-read-write
│   │   ├── server.h
│   │   └── connection.c         # Chain context management
│   ├── benchmarks/
│   │   ├── zero_copy_bench.c    # Zero-copy vs copy benchmark
│   │   ├── linked_chain_bench.c  # Linked vs manual chaining benchmark
│   │   └── vs_epoll_bench.c     # Comprehensive io_uring vs epoll comparison
│   └── examples/
│       ├── advanced_chain.c     # Example of complex linked operations
│       └── zero_copy_echo.c     # Zero-copy echo server example
└── tests/
    ├── test_zero_copy.c         # Unit tests for buffer refcounting
    └── test_linked_ops.c        # Chain failure and success scenarios
```

Infrastructure Starter Code: Zero-Copy Buffer Lifecycle Tracker

```c
/* src/core/buffer_pool.c - Extended version with zero-copy support */
#include <stdatomic.h>
#include "buffer_pool.h"

/* Extend struct io_buffer with reference count (already in naming conventions) */
/* struct io_buffer {
 *     void *data;
 *     size_t capacity;
 *     bool in_use;
 *     bool is_fixed;
 *     uint16_t fixed_index;
 *     atomic_int zc_refcount;  // New: kernel references for zero-copy
 *     struct io_buffer *next;
 * };
 */

/* Initialize buffer with zero-copy support */
struct buffer_pool* create_buffer_pool_zc(size_t buffer_size,
                                          size_t buffer_count,
                                          bool register_fixed,
                                          struct io_uring *ring) {
    struct buffer_pool *pool = create_buffer_pool(buffer_size, buffer_count,
                                                  register_fixed, ring);
    if (!pool) return NULL;

    /* Initialize refcounts for all buffers */
    for (size_t i = 0; i < buffer_count; i++) {
        atomic_store(&pool->buffers[i].zc_refcount, 0);
    }
    return pool;
}

/* Acquire buffer, checking for zero-copy availability */
struct io_buffer* acquire_buffer_zc(struct buffer_pool *pool) {
    pthread_mutex_lock(&pool->lock);

    struct io_buffer *buf = pool->free_list;
    while (buf) {
```

```
        /* Buffer is available if not in_use AND no kernel references */
        if (!buf->in_use && atomic_load(&buf->zc_refcount) == 0) {
            buf->in_use = true;
            pthread_mutex_unlock(&pool->lock);
            return buf;
        }
        buf = buf->next;
    }

    pthread_mutex_unlock(&pool->lock);
    return NULL; /* No buffers available */
}

/* Release buffer after zero-copy send */
void release_buffer_after_zc(struct buffer_pool *pool,
                             struct io_buffer *buf,
                             bool is_notification) {
    if (is_notification) {
        /* Kernel released its reference */
        int old = atomic_fetch_sub(&buf->zc_refcount, 1);
        if (old == 1) {
            /* Last kernel reference gone, buffer can be reused */
            pthread_mutex_lock(&pool->lock);
            if (!buf->in_use) {
                buf->next = pool->free_list;
                pool->free_list = buf;
            }
            pthread_mutex_unlock(&pool->lock);
        }
    } else {
        /* Application done with buffer, but kernel may still hold references */
        pthread_mutex_lock(&pool->lock);
        buf->in_use = false;
        /* If kernel has no references, add back to free list */
        if (atomic_load(&buf->zc_refcount) == 0) {
            buf->next = pool->free_list;
            pool->free_list = buf;
```

```
        }
        pthread_mutex_unlock(&pool->lock);
    }
}

/* Submit zero-copy send and track buffer */
int submit_zero_copy_send(struct io_uring *ring,
                          int sock_fd,
                          struct io_buffer *buf,
                          size_t len,
                          uint64_t user_data) {
    struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
    if (!sqe) return -EBUSY;

    /* Mark buffer as having a kernel reference */
    atomic_fetch_add(&buf->zc_refcount, 1);

    io_uring_prep_send_zc(sqe, sock_fd, buf->data, len, 0, 0);
    sqe->user_data = user_data;
    sqe->flags |= IOSQE_IO_DRAIN; /* Optional: ensure ordering if needed */

    return 0;
}
```

Core Logic Skeleton: Setting Up Linked SQE Chains

```c
/* src/network/server.c - Linked accept-read-write chain example */

/**
 * Submit a linked chain for a new connection: ACCEPT -> READ -> WRITE
 * This creates an atomic sequence: if accept succeeds, we immediately
 * schedule read and write operations for that connection.
 *
 * @param server The network server instance
 * @param accept_sqe Pre-obtained SQE for accept (will be linked)
 * @param read_sqe Pre-obtained SQE for read (will be linked)
 * @param write_sqe Pre-obtained SQE for write (will be linked)
 * @param conn Connection state to associate with operations
 * @return 0 on success, negative error code on failure
 */
int submit_linked_accept_read_write_chain(struct network_server *server,
                                          struct io_uring_sqe *accept_sqe,
                                          struct io_uring_sqe *read_sqe,
                                          struct io_uring_sqe *write_sqe,
                                          struct connection_state *conn) {
    // TODO 1: Prepare ACCEPT operation
    //   - Set accept_sqe->opcode = IORING_OP_ACCEPT
    //   - Set accept_sqe->fd = server->listen_fd
    //   - Set accept_sqe->addr = pointer to store client address
    //   - Set accept_sqe->len = sizeof(struct sockaddr_in)
    //   - Set accept_sqe->user_data = create_token(CONN_ACCEPTING, conn->id)
    //   - Set accept_sqe->flags = 0 (first in chain, no special flags)

    // TODO 2: Prepare READ operation (linked after ACCEPT)
    //   - Set read_sqe->opcode = IORING_OP_READ
    //   - Set read_sqe->fd = will be filled by accept completion (use -1 for now)
    //   - Set read_sqe->addr = conn->read_buffer->data
    //   - Set read_sqe->len = conn->read_buffer->capacity
    //   - Set read_sqe->user_data = create_token(CONN_READING, conn->id)
    //   - Set read_sqe->flags = IOSQE_IO_LINK (links to previous SQE)

    // TODO 3: Prepare WRITE operation (linked after READ)
    //   - Set write_sqe->opcode = IORING_OP_WRITE
```

```
    //   - Set write_sqe->fd = same as read (will be filled)
    //   - Set write_sqe->addr = conn->write_buffer->data
    //   - Set write_sqe->len = 0 (will be filled after read completes)
    //   - Set write_sqe->user_data = create_token(CONN_WRITING, conn->id)
    //   - Set write_sqe->flags = IOSQE_IO_LINK (links to previous SQE)

    // TODO 4: Set connection state to ACCEPTING
    //   - conn->state = CONN_ACCEPTING
    //   - conn->fd = -1 (will be set by accept)
    //   - Store connection in lookup table with conn->id

    // TODO 5: Submit the chain
    //   - Call io_uring_submit(server->engine->ring)
    //   - Return 0 on success, or error code from submission

    return 0;
}

/**
 * Handle completion of a linked chain. All three CQEs arrive together
 * after the final operation (WRITE) completes.
 *
 * @param server The network server instance
 * @param accept_cqe CQE for ACCEPT operation
 * @param read_cqe CQE for READ operation
 * @param write_cqe CQE for WRITE operation
 * @param conn Connection state from lookup table
 */
void handle_linked_chain_completion(struct network_server *server,
                                    struct io_uring_cqe *accept_cqe,
                                    struct io_uring_cqe *read_cqe,
                                    struct io_uring_cqe *write_cqe,
                                    struct connection_state *conn) {
    // TODO 1: Check for chain failure at any stage
    //   - If accept_cqe->res < 0: chain failed on accept, clean up connection
    //   - If read_cqe->res < 0: chain failed on read, but accept succeeded
    //   - If write_cqe->res < 0: chain failed on write, but accept/read succeeded
```

```
    // TODO 2: Process successful chain
    //   - Extract client fd from accept_cqe->res
    //   - Store fd in conn->fd
    //   - Process data from read_buffer (first read_cqe->res bytes)
    //   - Prepare response in write_buffer
    //   - Update write_buffer length

    // TODO 3: Update connection state
    //   - conn->state = CONN_READING (for next read)
    //   - conn->last_active = current time

    // TODO 4: Submit next read for persistent connection
    //   - Get new SQE with io_uring_get_sqe()
    //   - Prepare read for conn->fd into read_buffer
    //   - Submit (not linked this time, standalone operation)

    // TODO 5: Clean up on chain failure
    //   - If accept failed: free connection struct
    //   - If read/write failed: close conn->fd, free buffers, remove from table
    //   - Release all buffers back to pool
}
```

Core Logic Skeleton: Processing Zero-Copy Notifications

```c
/* src/core/engine.c - Enhanced completion processing */

/**
 * Process completion queue entries with zero-copy notification support.
 * Distinguishes between regular operation completions and buffer release
 * notifications.
 *
 * @param eng The io_uring engine instance
 * @return Number of CQEs processed
 */
int engine_process_completions_with_zc(struct io_uring_engine *eng) {
    struct io_uring *ring = engine_get_ring(eng);
    struct io_uring_cqe *cqes[32];
    int count = io_uring_peek_batch_cqe(ring, cqes, 32);
    int processed = 0;

    for (int i = 0; i < count; i++) {
        struct io_uring_cqe *cqe = cqes[i];

        // TODO 1: Check for zero-copy notification
        //   - if (cqe->flags & IORING_CQE_F_NOTIF)
        //   - Extract buffer pointer from cqe->user_data (specially encoded)
        //   - Call release_buffer_after_zc(pool, buffer, true)
        //   - Mark CQE as seen with io_uring_cqe_seen(ring, cqe)
        //   - Continue to next CQE (notification is not a regular completion)

        // TODO 2: Regular completion processing
        //   - Look up operation context via lookup_op_context(table, cqe->user_data)
        //   - If context not found: log error, skip
        //   - Switch on context->op_type to route to appropriate handler

        // TODO 3: Handle zero-copy send completion (not notification)
        //   - For IORING_OP_SEND_ZC completion:
        //   - Buffer still has kernel reference! Do NOT release yet
        //   - Update statistics: bytes sent = cqe->res > 0 ? cqe->res : 0
        //   - Schedule next operation if needed
        //   - Note: buffer release happens via separate notification CQE
```

```
        // TODO 4: Handle linked chain completions
        //   - For linked operations, all CQEs in chain arrive together
        //   - May need to process multiple CQEs for same user_data chain ID
        //   - Gather all chain CQEs before calling chain completion handler

        // TODO 5: Update engine statistics
        //   - eng->ctx->stats.cqe_processed++
        //   - eng->ctx->stats.bytes_transferred += (cqe->res > 0 ? cqe->res : 0)

        processed++;
        io_uring_cqe_seen(ring, cqe);
    }

    return processed;
}
```

Language-Specific Hints for C:

- Use `atomic_fetch_add` and `atomic_fetch_sub` from `<stdatomic.h>` for thread-safe reference counting, even in single-threaded programs, to ensure correct memory ordering with kernel operations.
- When encoding buffer pointers in `user_data` for zero-copy notifications, use: `(uint64_t)(uintptr_t)buffer_ptr`. Decode with `(struct io_buffer *)(uintptr_t)user_data`.
- For linked chains, set `sqe->flags |= IOSQE_IO_LINK` on all but the first SQE in the chain. The first SQE should have no link flag.
- Check kernel version before using `IORING_OP_SEND_ZC` (requires Linux 5.6+). Use `#ifdef` or runtime detection with `io_uring_get_probe()`.
- When using `IOSQE_IO_DRAIN`, be aware it affects the entire submission queue, not just your operation. Use it only for operations like `fsync` that truly need ordering.

Milestone Checkpoint: Advanced Features Verification

After implementing zero-copy sends and linked operations:

1. **Test Zero-Copy Buffer Tracking:**

```bash
./build/test_zero_copy
# Expected: Buffer refcount increments on submit, decrements on notification
# Should show "Buffer released after notification" messages
```

2. **Verify Linked Chain Atomicity:**

```bash
./build/test_linked_ops
# Test 1: Successful chain - all three operations complete
# Test 2: Failed accept - read/write never submitted
# Test 3: Failed read - write never executed, buffers cleaned up
```

3. **Benchmark Zero-Copy vs Copy:**

```bash
./build/benchmarks/zero_copy_bench --clients=100 --size=16384 --duration=10

# Expected: Zero-copy shows 20-40% lower CPU usage for same throughput

# Check output for "CPU utilization" and "throughput MB/s"
```

4. **Compare Linked vs Manual Chaining:**

```bash
./build/benchmarks/linked_chain_bench --requests=100000

# Expected: Linked chains show higher throughput (fewer syscalls)

# Manual chaining shows more flexible error handling
```

5. **Comprehensive io_uring vs epoll Benchmark:**

```bash
./build/benchmarks/vs_epoll_bench --workload=mixed --connections=1000

# Expected: io_uring outperforms epoll on high connection counts

# Especially for file I/O mixed with network I/O
```

Debugging Tips for Advanced Features:

Symptom	Likely Cause	How to Diagnose	Fix
Memory usage grows without bound	Zero-copy notifications not processed	Count `IORING_CQE_F_NOTIF` flags in completion loop	Ensure notification CQEs decrement refcounts
Linked chain hangs after first operation	Missing `IOSQE_IO_LINK` flag on middle SQEs	Check `sqe->flags` for each chain link	Set `IOSQE_IO_LINK` on all but first SQE
`IORING_OP_SEND_ZC` returns `-EINVAL`	Kernel too old or buffer not properly aligned	Check kernel version with `uname -r`	Use fallback copy for kernels < 5.6
Chain fails but resources not freed	Cleanup only on final chain success	Check `cqe->res` for each link in chain	Clean up on *any* link failure
High latency with `IOSQE_IO_DRAIN`	DRAIN serializing unrelated operations	Audit all SQEs for unnecessary DRAIN flags	Remove `IOSQE_IO_DRAIN` from non-critical ops

Use `strace` to verify system call patterns: linked chains should show single `io_uring_enter` for multiple operations, while zero-copy sends show normal `sendmsg` syscalls (handled internally by kernel).

Interactions and Data Flow

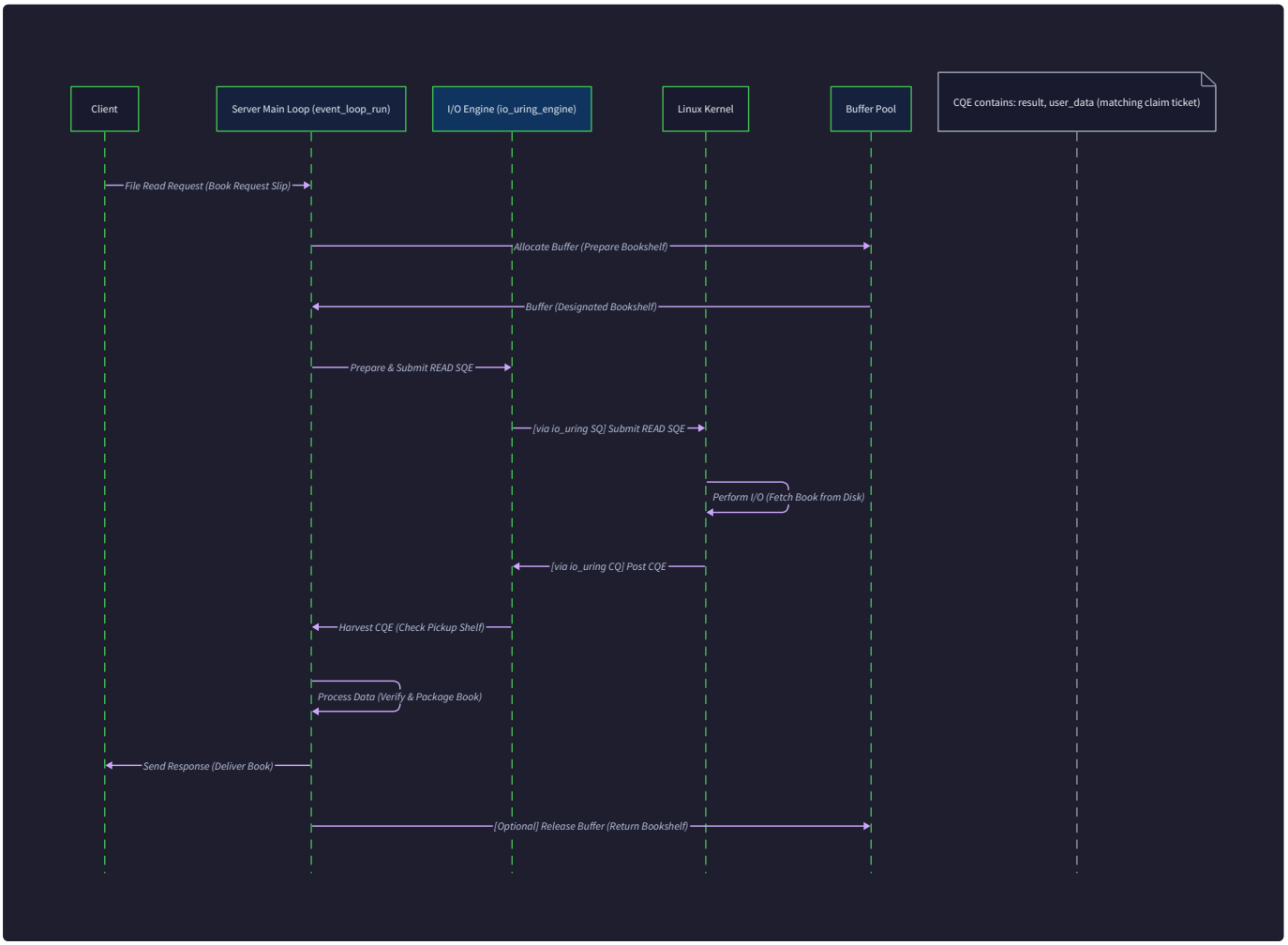
> **Milestone(s):** Milestone 2 (File I/O Server), Milestone 3 (Network Server), Milestone 4 (Advanced Features & Optimization)

This section details the concrete, step-by-step data flow for the two primary operations our server performs: serving a file and handling a network echo request. Understanding these sequences is critical for seeing how the asynchronous components—the `io_uring` engine, the buffer manager, and the connection state machines—interact in practice. We move beyond static structure to dynamic behavior, tracing the lifecycle of a single I/O request from initiation to final completion, highlighting the points of concurrency, error handling, and resource management. These walkthroughs serve as the canonical reference for how the system's parts orchestrate to achieve high performance.

Sequence: Serving a File Read

This flow demonstrates the server's capability for high-throughput, asynchronous file I/O, a cornerstone of Milestone 2. The mental model is that of a **Library Book Retrieval System**. A client's request is akin to submitting a book request slip (the SQE) to a librarian. Instead of waiting idly at the counter (blocking), you receive a claim ticket (`user_data`) and are free to handle other tasks. The librarian (the kernel) fetches the book from the stacks (the disk). When the book is ready, they place it on a designated pickup shelf (the CQ) and call your number (the `user_data` in the CQE). You then collect the book (the data buffer) and deliver it to the client.

The sequence, illustrated in



, involves the following participants: the Client, the Server's Main Event Loop (`event_loop_run`), the I/O Engine (`io_uring_engine`), the Buffer Pool (`buffer_pool`), and the Linux Kernel. Below is the detailed, step-by-step walkthrough.

1. **Request Reception & Preparation:** The server's main loop receives a file read request from a client (e.g., via a network connection already established). The request specifies a file path and an offset/range. The loop handler calls a function like `handle_file_request(struct connection_state *conn, const char *path, off_t offset, size_t len)` .
2. **Buffer Acquisition:** The handler must obtain a buffer to hold the file data that will be read. It calls `acquire_buffer(pool)` on the server's registered `buffer_pool` . This function returns a pointer to an available `struct io_buffer` . If the pool uses fixed buffers (recommended for performance), the buffer's `data` field points to a pre-registered memory region, and its `fixed_index` is set. The buffer's `in_use` flag is marked `true` .
3. **Operation Context Creation:** To track this in-flight operation, the server allocates an `io_op_context` via `allocate_op_context(context_table)` . This context is populated:
   - `op_type` = `OP_TYPE_READ` (an internal constant mapping to `IORING_OP_READ`).
   - `connection` = pointer to the `struct connection_state` of the requesting client.
   - `buffer` = pointer to the acquired `io_buffer` .
   - `file` = pointer to a `struct file_context` (previously opened and cached for the requested path).
   - `aux_data.file.offset` = the file offset for the read. The context's unique `id` is used to construct the `user_data` token.
4. **SQE Preparation:** The handler calls `engine_get_sqe(engine)` to obtain a free Submission Queue Entry. It configures the SQE:
   - `opcode` = `IORING_OP_READ`
   - `fd` = the file descriptor from `file_context->fd`
   - `addr` = the virtual address of the buffer's data (`io_buffer->data`). If using fixed buffers, this is set via `io_uring_sqe_set_buffer(sqe, buffer->fixed_index)` .
   - `len` = the number of bytes to read (up to buffer capacity).

- `offset` = the file offset from the context.
- `user_data` = a token encoding the `io_op_context`'s `id`. This is the critical linkage for completion.

5. **Submission to the Kernel:** The handler does **not** immediately call `io_uring_enter`. Instead, it returns, allowing the main event loop to practice **opportunistic batching**. The SQE remains in the user-space SQ ring. The event loop's `on_pre_submit` callback may prepare more SQEs for other pending requests. When the batch is sufficiently large or the loop is about to wait for completions, it calls `engine_submit_and_wait(engine, min_complete)`. This single system call (`io_uring_enter`) submits the entire batch of prepared SQEs (including our file read) to the kernel.
6. **Asynchronous Kernel I/O:** The kernel receives the SQE batch. For our file read request, it schedules the disk I/O. The server's main thread is **not blocked** during this time. It can continue processing other completions or preparing new submissions. This is the core of the async model.
7. **CQE Harvesting:** Eventually, the disk I/O completes. The kernel writes a Completion Queue Entry (CQE) into the shared CQ ring. The CQE contains:
   - `res` = the result of the operation (number of bytes read, or a negative error code).
   - `user_data` = the same token submitted in the SQE. The server's main loop, in its `engine_process_completions(engine)` phase, detects the new CQE (by comparing CQ tail). It reads the CQE and extracts the `user_data`.
8. **Context Lookup & Processing:** Using the `user_data` token, the server calls `lookup_op_context(context_table, user_data)` to retrieve the original `io_op_context`. This context tells us everything about the operation: it was a read for a specific connection using a specific buffer.
9. **Result Handling:** The handler examines `cqe->res`.
   - **Success (`res > 0`)**: The data is now in the buffer. The handler schedules a network write to send the data back to the client (`submit_connection_write`). The `io_buffer` ownership is transferred to the network operation.
   - **Short Read (`res >= 0` but less than requested)**: This may be normal (end-of-file). The server proceeds with the available data.
   - **Error (`res < 0`)**: The error code (e.g., `-EBADF`, `-EIO`) is logged. The server may send an error response to the client and clean up the operation.
10. **Cleanup & Resource Recycling:** After the operation is fully processed (e.g., after the network write using this buffer also completes), the buffer is returned to the pool via `release_buffer(pool, buffer)`, marking it `in_use = false`. The `io_op_context` is freed via `free_op_context(table, ctx)`, removing it from the tracking table. The `file_context` may have its `last_access` updated and its `ref_count` decremented.

> **Key Insight:** The **decoupling** of submission and completion is paramount. Steps 1-5 happen rapidly in user-space, stuffing requests into the ring. Steps 6 occurs asynchronously in kernel/hardware time. Steps 7-10 happen later, possibly out-of-order with respect to submission, but are correctly re-associated via the `user_data` token. This pipeline is what enables massive I/O parallelism.

Common Pitfalls: File Read Flow

⚠️ Pitfall: Buffer Released Before Kernel Read Completes

- **Description:** Calling `release_buffer` immediately after `engine_submit_and_wait` returns, but before the corresponding CQE is harvested. The buffer is recycled for another operation while the kernel is still writing data into it.
- **Why Wrong:** Causes catastrophic data corruption or crashes as the same memory is used for two concurrent I/O operations.
- **Fix:** The buffer must stay "owned" by the `io_op_context` until the CQE is processed. Only release it in the completion handler (Step 10) or transfer ownership to the next operation (e.g., a network write).

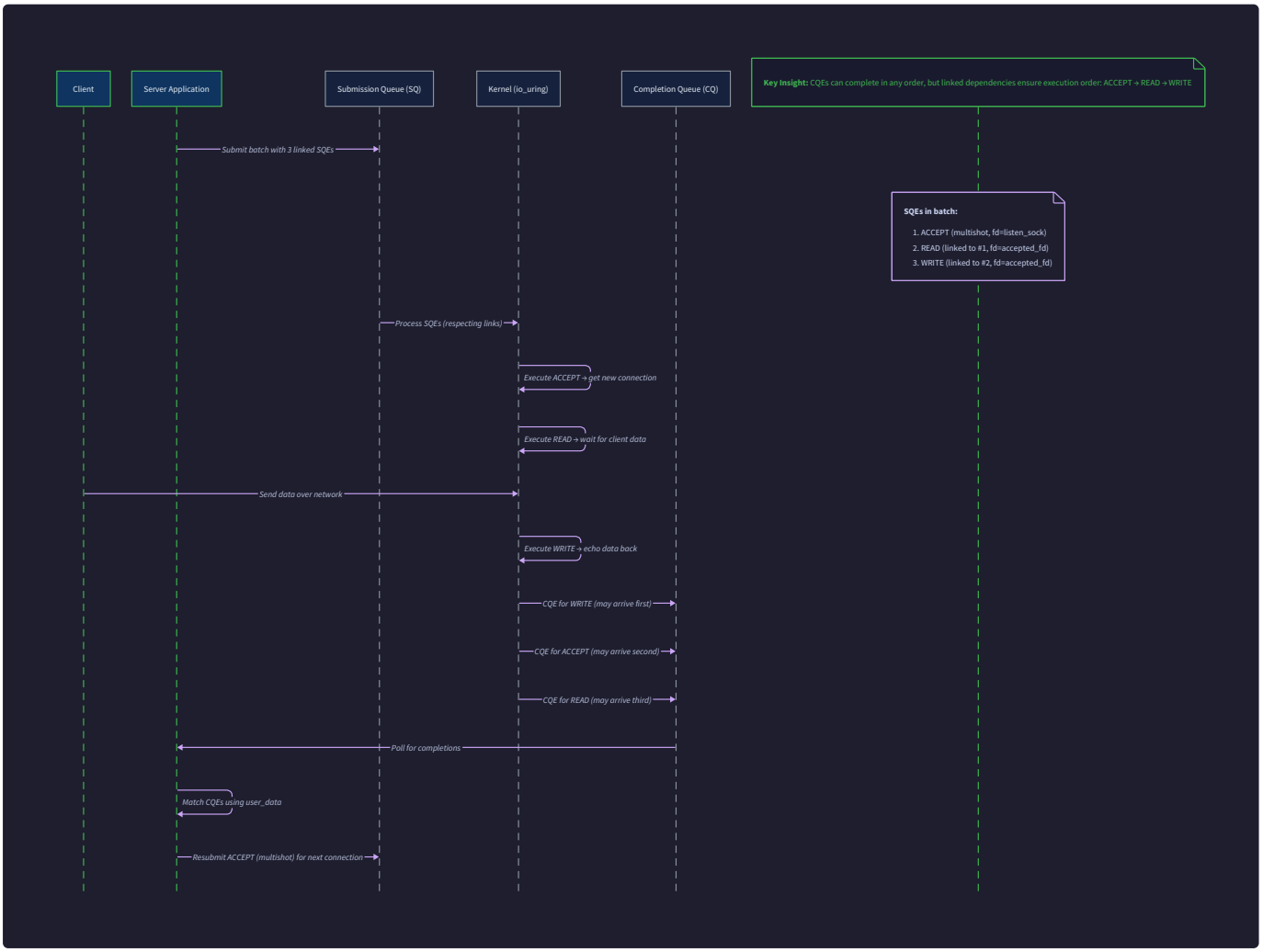
⚠️ Pitfall: Misinterpreting `io_uring_enter` Return Value

- **Description:** Assuming the return value of `engine_submit_and_wait` indicates the success of the individual I/O operations.
- **Why Wrong:** `io_uring_enter` only reports the number of SQEs successfully *submitted* to the kernel queue, not their execution results. Individual operation results are in their respective CQEs.
- **Fix:** Always check `cqe->res` for each completion. The `engine_submit` return value is only useful for detecting submission failures like a full SQ ring (`-EBUSY`).

Sequence: Network Echo Request/Response

This flow showcases the full power of `io_uring` for networked services, integrating concepts from Milestones 3 and 4: multishot accepts, linked operations, and zero-copy sends. The mental model is an **Assembly Line with a Concierge Desk**. The multishot accept is a concierge who continuously hands out numbered tickets (new socket FDs) to incoming customers. Each customer (connection) then moves to an assembly line (linked SQE chain) where Station 1 (READ) receives their raw material (network data), Station 2 (optional TRANSFORM) processes it, and Station 3 (WRITE) packages and sends back the result. Zero-copy is like Station 3 directly shipping the original material without repackaging it.

The sequence, detailed in



, is more complex and involves the `network_server`, the `connection_table`, and linked SQEs.

1. **Initialization - Multishot Accept Submission:** During server startup, `network_server_run` calls `submit_accept_multishot(server)`. This function:
    - Gets an SQE via `engine_get_sqe`.
    - Sets `opcode = IORING_OP_ACCEPT`, `fd = server->listen_fd`, and `flags = IORING_ACCEPT_MULTISHOT`.
    - Prepares an `io_op_context` with `op_type = OP_TYPE_ACCEPT_MULTI`.
    - Sets `sqe->user_data` to this context's ID.
    - Submits the SQE (or it's batched). This **single SQE** will generate multiple accept completions over time.
2. **Connection Acceptance:**
    - A client connects. The kernel processes the waiting multishot accept SQE.
    - A CQE is posted with `cqe->res` being the new socket file descriptor (or an error).
    - The completion handler (`handle_accept_completion`) is invoked via the context lookup.
    - On success, it creates a `struct connection_state` via `conn_create`, setting its `state = CONN_READING`. It registers this connection in the `connection_table`.
    - **Crucially, the multishot SQE remains active.** The kernel will automatically re-arm it to accept the next connection, eliminating the need for a new `accept` submission for the next client.
3. **Initiating the Request-Response Chain:** For the new connection, we want to read a request and then echo it back. We use a **linked SQE chain** for atomicity: if the read fails, the write won't be attempted.
    - The handler calls a function like `submit_linked_accept_read_write_chain`. This function prepares **three** SQEs, setting the `IOSQE_IO_LINK` flag on the first two.

        - **SQE A (ACCEPT - part of multishot):** Already completed in step 2. It is not part of this new chain.
        - **SQE 1 (READ):** `opcode = IORING_OP_READ`, `fd = new_socket_fd`. It acquires a buffer and sets up a context with `op_type = OP_TYPE_READ_LINKED`.
        - **SQE 2 (WRITE):** `opcode = IORING_OP_WRITE` (or `IORING_OP_SEND_ZC`), `fd = new_socket_fd`. Its `addr` and `len` are initially set to the buffer from SQE 1. Its context has `op_type = OP_TYPE_WRITE_LINKED`.
    - The `user_data` for each SQE points to its respective `io_op_context`, but these contexts are also linked (e.g., via `chain_next` pointer) to process the chain as a unit.
    - The SQEs are submitted as a batch. Because they are linked, the kernel guarantees SQE 2 (WRITE) will not start until SQE 1 (READ) completes successfully.
4. **Asynchronous Chain Execution:**
    - The kernel processes SQE 1 (READ) when data arrives on the socket. Upon completion, it posts a CQE for the READ. The WRITE operation is now eligible to start.
    - The kernel then processes SQE 2 (WRITE), sending the data from the same buffer back to the client. Upon completion, it posts a CQE for the WRITE.
5. **Chain Completion Handling:** CQEs may arrive in order (READ then WRITE) or potentially be processed together.
    - The event loop harvests CQEs. For each CQE, it looks up the context.
    - Because the contexts are linked, the handler (`handle_linked_chain_completion`) can manage the chain's state. For example, when the READ CQE arrives, it can verify the read was successful and store the byte count. When the WRITE CQE arrives, it knows the entire chain is done.
    - **Zero-Copy Variant:** If SQE 2 used `IORING_OP_SEND_ZC`, the WRITE CQE's `res` indicates the send result, but the buffer is still referenced by the kernel for the final network transmission. A **notification CQE** (with `IORING_CQE_F_NOTIF` flag set) will arrive later, signaling the kernel has released the buffer. Only then can `release_buffer_after_zc` be called.
6. **Connection Lifecycle Continuation:** After the echo chain completes successfully, the server typically re-arms the connection for the next request by submitting a new read SQE (potentially another linked chain), returning the connection to the `CONN_READING` state. If the client disconnects (read returns 0 or an error), the server transitions the connection to `CONN_CLOSING`, cancels any pending operations, and closes the socket, eventually freeing the `connection_state`.

Common Pitfalls: Network Echo Flow

⚠️ Pitfall: Leaking Linked SQEs on Chain Failure

- **Description:** If the READ in a linked chain fails (e.g., `-ECONNRESET`), the linked WRITE SQE will not be executed. If the application logic only waits for the WRITE CQE to clean up resources, it will wait forever.
- **Why Wrong:** The failed READ CQE is posted, but the WRITE SQE is never processed, so no WRITE CQE is generated. Resources (buffer, context) associated with the entire chain are leaked.
- **Fix:** In the completion handler for any SQE in a linked chain, always check for errors. If an error occurs on any link, immediately clean up all resources associated with the entire chain (all linked contexts and buffers). Do not wait for subsequent CQEs that will never arrive.

⚠️ Pitfall: Forgetting to Re-submit Multishot Accept

- **Description:** Using `IORING_ACCEPT_MULTISHOT` but then not handling the case where the kernel stops the multishot due to an error (e.g., `-EINVAL` if the listen socket options change).
- **Why Wrong:** After the first error, no new connections will be accepted, stalling the server.
- **Fix:** In the accept completion handler, check `cqe->res`. If it's an error, log it, and then **re-submit** a new (potentially non-multishot) accept SQE to keep the accept loop alive. The `liburing` helper `io_uring_prep_multishot_accept` handles this internally, but manual setup must account for it.

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Sequence Visualization	Manual logging with `printf` and timestamps.	Integration with tracing tools like `bpftrace` or `perf` to visualize kernel/userspace events.
Flow Validation	Unit tests mocking the `io_uring` syscalls.	Integration tests using a real `io_uring` instance and loopback network/files.

B. Recommended File/Module Structure: Place flow-specific handling logic within the respective component modules.

```
project/
  src/
    engine/          # io_uring engine core
      core.c         # engine_process_completions lives here
    network/
      server.c       # network_server_run, accept/read/write handlers
      connection.c   # Connection state machine transitions
    file/
      server.c       # File request handler
    main.c           # event_loop_run orchestrates the high-level flow
```

C. Infrastructure Starter Code (File Read Flow Helper): This helper function encapsulates steps 2-4 of the file read flow, preparing an SQE for an async read.

```c
/**
 * prepare_async_file_read - Prepares an SQE for an asynchronous file read.
 * @ring: Pointer to the io_uring instance.
 * @file_fd: The open file descriptor to read from.
 * @buf: The io_buffer to read data into.
 * @offset: File offset.
 * @len: Number of bytes to read.
 * @user_data: Token to correlate with completion.
 *
 * Returns: 0 on success, -1 if no SQE is available.
 */
int prepare_async_file_read(struct io_uring *ring, int file_fd,
                            struct io_buffer *buf, off_t offset,
                            size_t len, uint64_t user_data) {
    struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
    if (!sqe) {
        return -1; // SQ is full
    }

    io_uring_prep_read(sqe, file_fd, buf->data, len, offset);
    if (buf->is_fixed) {
        io_uring_sqe_set_flags(sqe, IOSQE_FIXED_FILE); // If fd is registered
        // For fixed buffers, we need to set the buffer index
        io_uring_sqe_set_buf(sqe, buf->fixed_index);
    }
    sqe->user_data = user_data;

    return 0;
}
```

D. Core Logic Skeleton Code (Network Echo Linked Chain Submission): This skeleton shows the structure for setting up a linked read-write chain for a new connection.

```c
/**
 * submit_linked_echo_chain - Submits a linked READ->WRITE chain for echoing data.
 * @server: The network server instance.
 * @conn: The newly accepted connection.
 *
 * Returns: 0 on success, -1 on failure (e.g., no SQEs available).
 */
int submit_linked_echo_chain(struct network_server *server,
                             struct connection_state *conn) {
    struct io_uring *ring = engine_get_ring(server->engine);
    struct io_uring_sqe *read_sqe, *write_sqe;

    // TODO 1: Acquire two free SQEs using io_uring_get_sqe(). Check for NULL.
    // TODO 2: Acquire a buffer for reading using acquire_buffer(server->buffer_pool).
    // TODO 3: Allocate an io_op_context for the READ operation. Set op_type, connection, buffer.
    // TODO 4: Allocate an io_op_context for the WRITE operation. Set op_type, connection.
    // TODO 5: Link the contexts (e.g., read_ctx->chain_next = write_ctx).
    // TODO 6: Prepare the READ SQE:
    //    - opcode = IORING_OP_READ
    //    - fd = conn->fd
    //    - Set buffer address or fixed index.
    //    - Set flags = IOSQE_IO_LINK (this links to the next SQE)
    //    - user_data = READ context's ID
    // TODO 7: Prepare the WRITE SQE:
    //    - opcode = IORING_OP_WRITE
    //    - fd = conn->fd
    //    - addr/len should point to the SAME buffer as the read.
    //    - DO NOT set IOSQE_IO_LINK (it's the end of the chain).
    //    - user_data = WRITE context's ID
    // TODO 8: Optionally, set write_sqe->addr and len based on expected read size?
    //         (We'll update this in the read completion handler).
    // TODO 9: Update connection state to CONN_READING.
    // TODO 10: Return 0 on success.

    return -1; // Placeholder
}
```

E. Language-Specific Hints (C):

- **Atomicity:** When updating shared data (like `conn->state`) between the submission thread and completion thread (which are the same in our single-threaded design, but could differ with `IORING_SETUP_SQPOLL`), consider using atomic operations or barriers. For simplicity in single-threaded, it's not needed.
- **Error Propagation:** Use negative error codes in `cqe->res` consistently. You can use `-errno` style values. Helper functions like `io_uring_cqe_get_res` handle the sign correctly.
- **Buffer Lifetime with Zero-Copy:** When using `IORING_OP_SEND_ZC`, you must not modify or free the buffer until you receive the notification CQE. Maintain a reference count (`io_buffer->zc_refcount`) that is incremented on submit and decremented on notification.

F. Milestone Checkpoint (Network Flow): To verify the network echo flow is working:

1. Start the server: `./server --port 8080`.
2. Use `netcat` to connect: `echo "Hello, io_uring!" | nc localhost 8080`.
3. **Expected Output:** The server should echo back "Hello, io_uring!".
4. Use `strace -e io_uring_enter ./server ...` to observe the batched submissions and completions. You should see few `io_uring_enter` calls relative to the number of I/O operations.
5. **Sign of Trouble:** If the connection hangs or you see many `io_uring_enter` calls with `to_submit=1`, the linked chain or multishot logic may be broken, causing serialized, non-batched I/O.

Error Handling and Edge Cases

> **Milestone(s):** Milestone 1 (Basic SQ/CQ Operations), Milestone 2 (File I/O Server), Milestone 3 (Network Server), Milestone 4 (Zero-copy, Linked Ops, and Benchmarks)

This section systematizes our approach to handling the inevitable failures that occur in any production system. In an asynchronous, event-driven architecture like our `io_uring` server, error handling becomes particularly critical because failures can manifest at any point in a complex chain of operations, often detached from the immediate context that initiated them. A single `-ECONNRESET` from a network read or an `-ENOBUFS` from a zero-copy send must be properly detected, classified, and recovered from without leaking resources or corrupting state. We'll establish a taxonomy of errors, define recovery strategies, and detail handling for specific edge cases that `io_uring` introduces.

Error Classification and Recovery Strategies

Think of error handling in an asynchronous server as a **medical triage system**. When a patient (operation) arrives with symptoms (error codes), the triage nurse (error handler) must quickly classify the severity (Application, Transient, or Fatal), decide on immediate treatment (retry, clean up, or escalate), and ensure proper follow-up (resource cleanup, state reset). Misclassification—treating a fatal internal bleeding as a minor scrape—leads to system death.

Our classification system is based on three dimensions: **origin** (where the error occurred), **recoverability** (whether the operation can safely be retried), and **scope** (whether the error affects a single operation, a connection, or the entire system). The following table defines our primary error categories and the prescribed recovery actions.

Category	Description	Typical Error Codes	Detection Method	Recovery Strategy	Example Scenario
Application Errors	Errors caused by client behavior or invalid requests. The operation failed as intended by the kernel's semantic checks.	`-EINVAL` (invalid arguments), `-EBADF` (bad file descriptor), `-ENOENT` (file not found), `-EPIPE` (broken pipe for write), `-ECONNRESET` (connection reset by peer on read)	Check `cqe->res` for negative value matching known application error codes.	Clean up the specific operation and its associated resources (buffer, context). If the error is connection-related (e.g., `-ECONNRESET`), transition the connection to `CONN_CLOSING` state and clean up all its pending operations. Log for monitoring.	Client sends a request to read a non-existent file. The kernel completes the `IORING_OP_READ` with `res = -ENOENT`. The server frees the operation's buffer and sends an appropriate error response to the client (if a response is expected).
Transient (Recoverable) Errors	Temporary resource constraints or conditions that may resolve if retried. The operation failed due to external, possibly temporary factors.	`-EAGAIN` / `-EWOULDBLOCK` (resource temporarily unavailable), `-ENOBUFS` (no buffer space available, common for zero-copy sends), `-ENOMEM` (out of memory)	Check `cqe->res` for specific negative values known to be transient. For `-EAGAIN`, also check if the operation was submitted with non-blocking semantics.	**For file I/O:** Retry the operation immediately or after a short delay (exponential backoff). Ensure the buffer remains valid. **For network I/O (sends):** For `-ENOBUFS` on zero-copy, wait for a notification CQE (`IORING_CQE_F_NOTIF`) before retrying or using a fallback buffer. **For accepts/reads:** May indicate `listen()` backlog is full; continue normal processing, kernel will retry.	During a traffic spike, a zero-copy send (`IORING_OP_SEND_ZC`) fails with `-ENOBUFS`. The server marks the buffer as "pending notification," waits for the kernel's notification CQE, and then retries the send or falls back to a regular copy send.
Fatal (Unrecoverable) Errors	Critical failures indicating bugs, unrecoverable resource exhaustion, or kernel issues. The system cannot continue normal operation.	`-EFAULT` (bad user memory access), `-EIO` (low-level I/O error), `-ENOSPC` (disk full), `-ENODEV` (device gone), `-ENXIO` (ring setup/configuration error)	Check `cqe->res` for severe errors. Also detect via system calls: `io_uring_enter()` returns `-EINVAL` or `-ENXIO` for bad parameters or kernel version mismatch.	Log the error with full context (operation, `user_data`, connection). For per-connection fatal errors (e.g., `-EIO` on a specific socket), close the connection and clean up all associated state. For global fatal errors (e.g., disk full, ring corruption), initiate graceful shutdown: stop accepting new connections, complete or cancel in-flight operations, and terminate the process.	The disk develops bad sectors. A file read operation completes with `res = -EIO`. The server logs the error, closes the client connection (if this was a client request), and marks the specific file as unusable for future requests.
System Resource Exhaustion	A subset of transient/fatal errors where a key resource pool is depleted.	`-ENOBUFS`, `-ENOMEM`, `-EMFILE` (process file descriptor limit), `-ENFILE` (system file descriptor limit)	Check `cqe->res` for resource errors. Also monitor `engine_stats` for high `cqe_processed` vs. `sqe_submitted` indicating backlog. Monitor `buffer_pool->free_list` and `connection_table->count`.	**Defensive:** Implement hard limits on connection counts and buffer pools, rejecting new requests when limits are reached. **Reactive:** For file descriptor exhaustion, implement LRU connection eviction or increase limits via `setrlimit()`. For buffer exhaustion, use a smaller buffer size or increase pool size. Log alerts for operator intervention.	The server hits the system-wide file descriptor limit (`-EMFILE`). New `accept()` operations start failing. The server logs a critical alert, stops accepting new connections, and begins closing idle connections (based on `last_active`) to free up descriptors.

Decision: Centralized Error Handler with Pluggable Strategies

Context: Errors from `io_uring` completions can originate from diverse subsystems (network, file, memory) and require different recovery actions. We need a consistent way to dispatch errors to appropriate handlers without scattering conditionals throughout the event loop.

Options Considered:

1. **Inline error checking in `engine_process_completions`:** Each completion handler includes its own `if (cqe->res < 0)` logic.
2. **Centralized error dispatch table:** A table mapping `(op_type, error_code)` tuples to handler function pointers.
3. **Subsystem-specific error handlers:** Each component (network, file) registers its own error handler callback, and the engine calls it with the error context.

Decision: We adopt a hybrid approach: the core `engine_process_completions` function will categorize errors into the three main types (Application, Transient, Fatal) and call a generic error handler `handle_io_error(struct io_uring_cqe *cqe, struct io_op_context *ctx)`. This handler will then delegate to subsystem-specific handlers (e.g., `handle_network_error`, `handle_file_error`) based on the `ctx->op_type`. Subsystem handlers implement the detailed recovery logic.

Rationale: This balances separation of concerns with simplicity. The core engine doesn't need to understand specific error semantics for each operation type, but we avoid a monolithic error handler that becomes a switch statement on all possible `op_type` and error code combinations. Subsystem handlers can maintain their own state (e.g., network handler knows about connection state machines) without exposing it to the engine.

Consequences:

- **Positive:** Clean separation; new operation types can add their own error handlers without modifying core engine code.
- **Positive:** Error handling logic is co-located with subsystem logic, making it easier to reason about.
- **Negative:** Requires careful design of the error handler signature to pass all necessary context (operation type, buffer pointers, connection state).
- **Negative:** Adds one level of indirection for error processing.

Specific Edge Cases and Handling

Beyond generic error classification, `io_uring` introduces several subtle edge cases that require explicit handling. These are the "gotchas" that can cause resource leaks, data corruption, or performance degradation if overlooked.

1. Short Reads and Writes

The Scenario: An `IORING_OP_READ` on a socket completes with `cqe->res = 128`, but the requested buffer length was 1024. This is not an error (`res >= 0`) but indicates partial data.

Handling Strategy: For **network reads**, treat short reads as normal: the kernel gives us what's currently available in the receive buffer. Our read handler should process the available data and then submit a new read for the remaining buffer space (or switch to a new buffer if the protocol supports message boundaries). For **file reads**, short reads typically indicate end-of-file (when `res` is less than requested but `> 0`). Our file I/O handler should treat this as successful completion of the read with the actual bytes read.

Implementation Pattern: In the completion handler for reads, check if `cqe->res < requested_len`. If so, adjust buffer pointers and remaining length, then either process the partial data immediately (for network streaming) or accumulate until a full message is received (for message-based protocols).

```c
// Pseudo-logic in network read completion handler

bytes_received = cqe->res;

if (bytes_received > 0) {

    // Process the received bytes

    process_data(conn->read_buffer, bytes_received);



    // If we expect more data (e.g., HTTP Content-Length not yet met),

    // adjust buffer and resubmit a read for remaining capacity

    if (bytes_received < buffer_capacity) {

        conn->read_buffer->data += bytes_received;

        conn->read_buffer->capacity -= bytes_received;

        submit_connection_read(server, conn); // Read more into same buffer

    } else {

        // Buffer full, process and acquire a fresh buffer

        release_buffer(pool, conn->read_buffer);

        conn->read_buffer = acquire_buffer(pool);

        submit_connection_read(server, conn);

    }

}
```

2. Connection Resets During In-Flight Operations

The Scenario: A client TCP connection resets (RST packet) while the server has a `IORING_OP_READ` and a `IORING_OP_WRITE` pending for that same socket. The kernel will complete both operations with `-ECONNRESET` (or `-EPIPE` for writes), but the order of completion is non-deterministic.

Handling Strategy: When the first `-ECONNRESET` completion arrives, immediately transition the connection to `CONN_CLOSING` state and **cancel all other pending operations** for that socket. Use `io_uring`'s cancellation support (`IORING_OP_ASYNC_CANCEL`) targeting the connection's file descriptor or specific `user_data` tokens. This prevents double-free errors when the second completion arrives.

Cancellation Pattern: In `handle_network_error()` when encountering `-ECONNRESET`:

1. Mark connection state as `CONN_CLOSING`.
2. Iterate through the `context_table` to find all `io_op_context` objects with `ctx->connection == affected_conn`.
3. For each pending operation, submit an `IORING_OP_ASYNC_CANCEL` SQE with `addr` set to the `ctx->id` (or `fd` for cancel-all).
4. In the cancellation's completion handler, clean up the individual operation context.
5. After all pending operations are canceled (or a timeout), close the socket and free the connection.

3. `EBUSY` on Ring Submission

The Scenario: Calling `io_uring_submit()` (or `io_uring_enter()` with `to_submit > 0`) returns `-EBUSY`. This indicates the submission queue is full because the kernel hasn't yet consumed pending SQEs, often due to the kernel thread (`IORING_SETUP_SQPOLL`) being overloaded or stuck.

Handling Strategy: First, avoid hitting `EBUSY` by monitoring the submission queue fill level. The `io_uring` instance provides `sq_ring->tail` and `sq_ring->head` pointers; we can calculate available space: `sq_ring->mask + 1 - (sq_ring->tail - sq_ring->head)`. Never prepare more SQEs than available space.

If `EBUSY` does occur (defensive programming), we have two options:

1. **Busy-wait and retry:** If using `IORING_SETUP_SQPOLL`, the kernel thread might be temporarily saturated. Sleep briefly (microseconds) and retry the submit.
2. **Fall back to blocking submit:** Call `io_uring_enter()` with `IORING_ENTER_GETEVENTS` and `min_complete` set to 1, which will block until the kernel processes at least one completion, freeing up SQ space.

Implementation: Our `engine_submit()` function should check for `-EBUSY` return:

```c
int engine_submit(struct io_uring_engine *eng) {
    int ret = io_uring_submit(&eng->ctx->ring);
    if (ret == -EBUSY) {
        // Option 2: Block until kernel processes something
        ret = io_uring_enter(eng->ctx->ring.fd, 0, 1,
                             IORING_ENTER_GETEVENTS, NULL);
        if (ret >= 0) {
            // Now retry the original submission
            ret = io_uring_submit(&eng->ctx->ring);
        }
    }
    eng->ctx->stats.sqe_submitted += (ret > 0 ? ret : 0);
    return ret;
}
```

4. Completion Queue (CQ) Overflow

The Scenario: The completion queue becomes full because the application isn't harvesting CQEs fast enough. The kernel sets the `IORING_CQ_F_OVERFLOW` flag in the CQ ring and drops completions. This is a critical failure mode indicating the application cannot keep up with I/O completion rate.

Detection: Periodically check `cq_ring->flags` for `IORING_CQ_F_OVERFLOW`. Alternatively, monitor the difference between `engine_stats.cqe_processed` and expected completions based on submitted operations.

Recovery: Once overflow occurs, the ring is in an undefined state—some operations may have completed but their CQEs were lost. The only safe recovery is to **terminate the process**. To prevent overflow:

- Size the CQ appropriately (typically 2x SQ size).
- Implement proactive CQE harvesting: use `io_uring_peek_batch_cqe()` to process completions in batches, and ensure the event loop doesn't spend too much time in submission phase without checking completions.
- Consider using `IORING_SETUP_CQ_SIZE` during setup to allocate a larger CQ.

5. Zero-Copy Notification Misordering

The Scenario: A zero-copy send (`IORING_OP_SEND_ZC`) completes with success (`cqe->res = bytes_sent`), but the notification CQE (`IORING_CQE_F_NOTIF`) arrives **before** the send completion CQE. The application might release the buffer prematurely, causing the kernel to reference freed memory.

Handling Strategy: Implement **reference counting** on buffers used for zero-copy operations. Each zero-copy send increments the buffer's `zc_refcount`. The notification CQE decrements it. The buffer is only returned to the pool when `zc_refcount` reaches zero. This handles out-of-order completions correctly.

Buffer Lifecycle:

1. `acquire_buffer_zc()`: Checks `buffer->zc_refcount == 0`, then increments to 1.
2. `submit_zero_copy_send()`: Increments `zc_refcount` to 2 (one for send, one for notification).
3. Send completion CQE arrives: Decrements `zc_refcount` to 1.

4. Notification CQE arrives: Decrements `zc_refcount` to 0, calls `release_buffer()`.

6. Linked Chain Partial Failure

The Scenario: A chain of three linked SQEs (ACCEPT→READ→WRITE) is submitted. The ACCEPT succeeds, but the READ fails with `-ECONNRESET`. According to `io_uring` semantics, the entire chain stops at the first failure—the WRITE SQE will never be executed.

Handling Strategy: In the completion handler for linked chains, check all CQEs in the chain. If any has `res < 0`, the entire chain is considered failed. Clean up all resources allocated for the chain. For the ACCEPT→READ→WRITE example:

- If READ fails, we still have a successfully accepted socket that needs to be closed.
- The WRITE operation never executed, so no buffer was consumed for it.
- Implementation must track which operations in the chain were successfully submitted (via `user_data` mapping) and clean up each appropriately.

Chain Cleanup Pattern: Store chain metadata in the first operation's `io_op_context` (using `aux_data` union). When the first completion arrives with an error, walk the chain metadata to clean up all pending contexts.

7. Multishot Re-Submission Race Condition

The Scenario: A multishot accept operation (`IORING_OP_ACCEPT` with `IORING_ACCEPT_MULTISHOT`) completes, and the application processes the CQE. Before the kernel auto-resubmits the SQE, the application manually submits another accept SQE for the same socket, creating duplicate accept operations.

Prevention: Never manually submit accept SQEs on sockets using multishot. Designate certain listener sockets as "multishot" and never issue regular accept operations on them. Track multishot operations in a separate list and ensure they're only submitted once during initialization.

> **Design Insight:** The key to robust error handling in `io_uring` is embracing the asynchronous mindset. Errors are not exceptions to be feared but normal completion outcomes that must be processed with the same rigor as successful completions. Every error path must consider: (1) What kernel resources are still held (buffers, file descriptors), (2) What application state is inconsistent, and (3) Whether other in-flight operations depend on the failed operation.

Common Pitfalls: Error Handling

⚠️ Pitfall: Treating all negative `cqe->res` values as fatal errors

- **What happens:** The server crashes or disconnects clients unnecessarily when encountering recoverable errors like `-EAGAIN`.
- **Why it's wrong:** `-EAGAIN` is a normal part of non-blocking I/O and should trigger a retry, not a connection close.
- **Fix:** Implement the error classification system above. Check specific error codes and route to appropriate handlers.

⚠️ Pitfall: Not pairing zero-copy sends with notification CQEs

- **What happens:** Memory corruption or use-after-free when the kernel tries to access a buffer that was freed after the send completion but before the notification.
- **Why it's wrong:** Zero-copy sends have two-phase completion: data sent (send CQE) and buffer released (notification CQE). The buffer must remain valid until both arrive.
- **Fix:** Implement reference counting as described above. Only release buffers when `zc_refcount` reaches zero.

⚠️ Pitfall: Forgetting to cancel pending operations on connection close

- **What happens:** When a connection closes (client disconnect or server error), pending reads/writes for that socket eventually complete with errors (`-ECONNRESET`). If the connection state was already freed, these completions cause wild pointer dereferences.
- **Why it's wrong:** The kernel completes all submitted operations, even if the socket is closed mid-flight.
- **Fix:** Implement connection state machine with `CONN_CLOSING` state. When entering this state, cancel all pending operations using `IORING_OP_ASYNC_CANCEL`. Only free connection resources after all pending operations are accounted for (either completed or canceled).

⚠️ Pitfall: Ignoring CQ overflow

- **What happens:** Completions are silently dropped, leading to resource leaks (buffers never released, file descriptors never closed) and application state corruption.
- **Why it's wrong:** The application loses track of operations it submitted, breaking the fundamental contract with `io_uring`.
- **Fix:** Monitor CQ fill level proactively. Size CQ appropriately (2x SQ). If overflow is detected, log a critical error and initiate graceful shutdown—the system state is compromised.

⚠️ **Pitfall: Not handling short reads in streaming protocols**

- **What happens:** For HTTP or other message-based protocols, a short read might be misinterpreted as a complete message, causing protocol parsing errors.
- **Why it's wrong:** TCP is a byte stream; message boundaries are application-layer constructs.
- **Fix:** For message-based protocols, implement buffering and reassembly. For HTTP, use a stateful parser that accumulates data until a complete request is received.

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Error Classification	Simple `switch(cqe->res)` in completion handler	Pluggable error handler registry with subsystem callbacks
Buffer Tracking for Zero-copy	Atomic reference counting per buffer	Lock-free reference counting with hazard pointers for multi-threaded
Operation Cancellation	Cancel by file descriptor (cancels all ops on socket)	Cancel by individual `user_data` token for precision
Resource Monitoring	Periodic logging of ring/buffer/connection stats	Integrated with Prometheus metrics for real-time alerting

B. Recommended File/Module Structure:

```
project-root/
  src/
    core/
      error.c           # Error classification and dispatch logic
      error.h           # Error codes, handler function types
    network/
      network_error.c   # Network-specific error handlers
    file/
      file_error.c      # File I/O error handlers
    buffers/
      buffer_zc.c       # Zero-copy buffer reference counting
```

C. Infrastructure Starter Code (Error Dispatch):

```c
/* error.h */

#ifndef ERROR_H

#define ERROR_H


#include <liburing.h>

#include "data_model.h"


typedef enum {

    ERR_CATEGORY_APPLICATION,

    ERR_CATEGORY_TRANSIENT,

    ERR_CATEGORY_FATAL,

    ERR_CATEGORY_RESOURCE

} error_category_t;


typedef struct {

    int error_code;

    error_category_t category;

    const char *description;

} error_descriptor_t;


// Error handler function type

typedef int (*error_handler_fn)(struct io_uring_cqe *cqe,

                                struct io_op_context *ctx,

                                void *server_context);


// Register subsystem error handler

void error_register_handler(uint8_t op_type, error_handler_fn handler);


// Main error dispatch function

int handle_io_error(struct io_uring_cqe *cqe, struct io_op_context *ctx,

                    void *server_context);


// Helper to classify error code

error_category_t classify_error(int error_code);


#endif /* ERROR_H */
```

```c
/* error.c */

#include "error.h"

#include <stdlib.h>

#include <string.h>

#include <errno.h>


#define MAX_HANDLERS 256

static error_handler_fn error_handlers[MAX_HANDLERS] = {NULL};


void error_register_handler(uint8_t op_type, error_handler_fn handler) {

    if (op_type < MAX_HANDLERS) {

        error_handlers[op_type] = handler;

    }

}


error_category_t classify_error(int error_code) {

    // Map error codes to categories

    switch (error_code) {

        case -EAGAIN:

        case -EWOULDBLOCK:

        case -ENOBUFS:

            return ERR_CATEGORY_TRANSIENT;



        case -ECONNRESET:

        case -EPIPE:

        case -ENOENT:

        case -EINVAL:

        case -EBADF:

            return ERR_CATEGORY_APPLICATION;



        case -EFAULT:

        case -EIO:

        case -ENOSPC:

        case -ENODEV:

            return ERR_CATEGORY_FATAL;



        case -ENOMEM:

        case -EMFILE:
```

```
        case -ENFILE:
            return ERR_CATEGORY_RESOURCE;

        default:
            // Negative but unknown: treat as fatal
            if (error_code < 0) return ERR_CATEGORY_FATAL;
            return ERR_CATEGORY_APPLICATION; // Not an error
    }
}

int handle_io_error(struct io_uring_cqe *cqe, struct io_op_context *ctx,
                    void *server_context) {
    if (!cqe || !ctx) return -EINVAL;

    int error_code = cqe->res;
    error_category_t category = classify_error(error_code);

    // Call subsystem-specific handler if registered
    if (ctx->op_type < MAX_HANDLERS && error_handlers[ctx->op_type]) {
        return error_handlers[ctx->op_type](cqe, ctx, server_context);
    }

    // Default handling based on category
    switch (category) {
        case ERR_CATEGORY_TRANSIENT:
            // TODO 1: For file I/O, implement retry with backoff
            // TODO 2: For network, check if operation should be retried
            // TODO 3: Log transient error for monitoring
            break;

        case ERR_CATEGORY_APPLICATION:
            // TODO 4: Clean up operation-specific resources
            // TODO 5: If connection-related, mark connection for cleanup
            // TODO 6: Send error response if applicable
            break;

        case ERR_CATEGORY_FATAL:
```

```
            // TODO 7: Log critical error with full context
            // TODO 8: Initiate graceful shutdown sequence
            break;

        case ERR_CATEGORY_RESOURCE:
            // TODO 9: Log resource exhaustion alert
            // TODO 10: Implement resource reclamation or rejection
            break;
    }

    return 0;
}
```

D. Core Logic Skeleton Code (Network Error Handler):

C

```c
/* network_error.c */

#include "network_error.h"

#include "connection.h"

#include "buffer_pool.h"

#include <liburing.h>

#include <unistd.h>

#include <string.h>


int handle_network_error(struct io_uring_cqe *cqe,

                         struct io_op_context *ctx,

                         void *server_context) {

    struct network_server *server = (struct network_server *)server_context;

    struct connection_state *conn = ctx->connection;


    if (!server || !conn) return -EINVAL;


    int error_code = cqe->res;


    // TODO 1: Switch on error_code for specific handling

    switch (error_code) {

        case -ECONNRESET:

        case -EPIPE:

            // Connection was reset by peer

            // TODO 2: Mark connection as CONN_CLOSING

            conn->state = CONN_CLOSING;


            // TODO 3: Cancel all pending operations for this connection

            // Iterate through context table to find all contexts for this conn

            // Submit IORING_OP_ASYNC_CANCEL for each


            // TODO 4: Close socket after cancellation completes

            break;


        case -EAGAIN:

            // Transient: would block

            // TODO 5: For reads: retry immediately

            // TODO 6: For writes: implement backoff or buffer data
```

```
            break;


        case -ENOBUFS:
            // Zero-copy specific: no buffer space
            // TODO 7: Mark buffer as waiting for notification
            // TODO 8: Setup fallback to regular send if persistent
            break;


        default:
            // Unhandled network error
            // TODO 9: Log unknown network error
            // TODO 10: Close connection as precaution
            conn->state = CONN_CLOSING;
            break;
    }


    // TODO 11: Always free the operation context after error handling
    free_op_context(server->ctx_table, ctx);


    return 0;
}
```

E. Language-Specific Hints (C):

- Use `strerror(-error_code)` to convert negative error codes to human-readable strings for logging.
- For atomic reference counting on buffers, use GCC builtins: `__atomic_add_fetch(&buf->zc_refcount, 1, __ATOMIC_SEQ_CST)`.
- When implementing retry logic with exponential backoff, consider using `nanosleep()` for microsecond-precision delays.
- For monitoring CQ overflow, check the flags field: `if (ring->cq.flags & IORING_CQ_F_OVERFLOW) { /* handle */ }`.

F. Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Server crashes with wild pointer dereference in completion handler	Connection state freed while operations still pending	Add logging in `conn_destroy()` and completion handlers. Check `pending_ops` count before freeing.	Implement cancellation on connection close. Ensure `pending_ops` reaches 0 before freeing connection.
Memory usage grows without bound	Buffer leaks due to lost completions (CQ overflow or unhandled errors)	Monitor CQ overflow flag. Add buffer allocation/deallocation logging. Use `io_uring_peek_cqe()` to check for unharvested CQEs.	Fix CQ overflow by sizing CQ larger. Ensure every submission has a corresponding completion handler that releases buffers.
Zero-copy sends cause segmentation faults	Buffer released before kernel notification	Add reference counting logging. Check order of send completion vs notification completion.	Implement proper reference counting. Only release buffer when `zc_refcount` reaches 0.
High CPU usage with low throughput	Busy-loop retrying `-EAGAIN` errors	Log error codes in completion handlers. Check if transient errors are causing immediate retries.	Implement exponential backoff for retries. Consider batching retry operations instead of immediate resubmission.
`io_uring_submit()` frequently returns `-EBUSY`	Submission queue too small or kernel SQ poll thread overloaded	Monitor SQ fill level: `sq_ring->tail - sq_ring->head`. Check if using `IORING_SETUP_SQPOLL` and kernel thread status.	Increase SQ size. Reduce submission batch size. If using SQPOLL, check kernel thread CPU affinity and priority.
Linked chains not executing completely	Chain broken by an error in middle operation	Add logging for each SQE in chain submission and each CQE completion. Check error codes in middle operations.	Implement chain cleanup on partial failure. Ensure error handling for middle operations doesn't break chain semantics.

Testing Strategy and Milestone Checkpoints

> **Milestone(s):** All milestones (1-4)

Verification of progress in a complex system like an io_uring-based server requires a structured, incremental approach. This section provides concrete testing strategies and checkpoint validations for each milestone, ensuring you can confidently demonstrate that each architectural component functions as designed before moving to the next level of complexity. Unlike traditional unit testing, io_uring systems require integration-style validation that exercises the full asynchronous pipeline from submission to completion.

Overall Testing Philosophy

> **Testing Principle:** *Test the asynchronous pipeline, not just individual functions.*
>
> With io_uring, correctness depends on the entire flow: SQE preparation → kernel processing → CQE harvesting → context handling. Your testing should validate this end-to-end behavior, not just isolated helper functions.

Think of testing an io_uring server like validating a factory assembly line. You wouldn't just check that the welding station works in isolation; you'd verify that parts flow correctly from station to station, with proper handoffs, error handling, and throughput. Similarly, effective io_uring testing validates:

1. **Submission-to-Completion Correlation:** That each CQE correctly matches its originating SQE via `user_data` tokens
2. **Resource Lifecycle Management:** That buffers, connections, and operation contexts are properly acquired and released
3. **Concurrency Safety:** That the single-threaded event loop correctly handles overlapping in-flight operations
4. **Performance Characteristics:** That the system achieves the expected throughput and latency improvements

Testing Level	Tools & Techniques	What to Verify	Success Indicators
Unit/Component	Custom test harnesses, mock file descriptors, memory checkers	Individual functions like `engine_submit()`, `acquire_buffer()`	Functions behave as documented, no memory leaks
Integration	Real file I/O, network sockets, `strace`, `perf`	Full SQE→CQE pipeline, buffer registration, multishot operations	Operations complete successfully, resources managed correctly
Performance	`dd`, `wrk`, `ab`, custom benchmarks, `perf stat`	Throughput, latency, scalability vs. synchronous/epoll baselines	Meets or exceeds performance targets for each milestone
Stress/Load	Connection flood tools, memory limiters, fault injection	Resource exhaustion handling, error recovery, stability under load	System degrades gracefully, maintains core functionality

Diagnostic Tools: Keep these tools ready throughout development:

- **`strace -f`**: Trace system calls to verify `io_uring_enter` batching and frequency
- **`perf trace`**: Lower-overhead alternative to `strace` for high-throughput testing
- **`perf stat`**: Measure CPU cycles, context switches, cache misses
- **`valgrind --tool=memcheck`**: Detect memory leaks, especially in buffer pools
- **`liburing` test utilities**: Reference implementations to compare behavior
- **Custom logging**: Add verbose logging with operation IDs, buffer indices, and timestamps

> **Mental Model: The Factory Quality Control Station**
>
> Imagine your testing strategy as a quality control station at the end of a factory assembly line. Each milestone adds new stations to the line (file I/O, network handling, zero-copy operations). The quality control station doesn't just check individual parts—it validates that the entire line produces working products (completed I/O operations) with the promised efficiency gains. You test the line at increasing speeds (concurrency) and with various raw materials (different request types) to ensure it meets specifications.

Milestone 1 Checkpoint: Basic Operations

Goal: Verify that your io_uring engine core correctly initializes rings, submits operations, and harvests completions.

Validation Strategy

Create a simple test program that performs these steps:

1. **Ring Initialization Verification:** Initialize an io_uring instance with specific SQ/CQ sizes and verify the mmap'd memory regions are accessible
2. **Single Operation Test:** Submit a single write SQE to a file descriptor, wait for completion, verify the CQE result
3. **Batching Verification:** Submit multiple SQEs in a single `io_uring_enter` call, verify all completions arrive
4. **`user_data` Correlation:** Use distinct `user_data` tokens for each operation, verify they match in CQEs

Concrete Test Procedure:

```bash
# 1. Build and run your Milestone 1 test program
$ gcc -o milestone1_test milestone1_test.c -luring
$ ./milestone1_test

# Expected output should show:
# - Ring initialized successfully (SQ size: 32, CQ size: 64)
# - Single write operation submitted and completed (res=expected_bytes)
# - Batch of 4 operations submitted with 1 io_uring_enter call
# - All 4 completions received with correct user_data correlation
# - No memory leaks reported by valgrind
```

Detailed Validation Steps:

Step	Command/Action	Expected Outcome	Failure Indicators
1. Basic Setup	`engine_create(&cfg)`	Returns non-NULL `io_uring_engine*`, `engine_get_stats()` shows zero submissions	Returns NULL, or stats show non-zero values before any operations
2. File Descriptor Test	Create temp file, submit write SQE via `engine_get_sqe()` + `engine_submit()`	`engine_wait_and_process()` returns 1 completion, CQE `res` equals bytes written	No completion arrives, `res` is negative (error), or wrong fd processed
3. Batching Verification	Submit 8 SQEs, call `engine_submit()` once	`engine_process_completions()` eventually processes all 8, `engine_get_stats()` shows `batches_submitted: 1`	Need multiple `engine_submit()` calls to process all, or completions lost
4. Memory Barrier Check	Run with `valgrind --tool=helgrind`	No race condition warnings on SQ/CQ head/tail updates	Warnings about data races on ring indices
5. Cleanup	`engine_destroy()`	All memory freed, no open file descriptors remain	Memory leaks, or io_uring instance fd still open

Using `strace` for Verification:

```bash
# Run with strace to verify batching
$ strace -e io_uring_enter ./milestone1_test 2>&1 | grep io_uring_enter

# Expected output for batch test:
# io_uring_enter(3, 8, 0, 0, NULL) = 8  # Submitted 8 SQEs in one call
# io_uring_enter(3, 0, 8, IORING_ENTER_GETEVENTS, NULL) = 8  # Waited for 8 completions

# If you see multiple calls with small to_submit values, batching isn't working:
# io_uring_enter(3, 1, 0, 0, NULL) = 1  # INEFFICIENT: One at a time
# io_uring_enter(3, 1, 0, 0, NULL) = 1
```

Performance Baseline: While performance isn't the primary goal of Milestone 1, you should establish a baseline. Time how long it takes to submit and complete 10,000 trivial operations (writes to `/dev/null`). Compare against the same number of `write()` syscalls. Even at this early stage, you should see reduced syscall overhead.

> **Checkpoint Success Criteria:** Your engine core can reliably submit batches of operations and harvest their completions with correct `user_data` correlation. Running the test program produces the expected output with no errors, memory leaks, or data races. The `strace` output shows efficient batching (multiple SQEs per `io_uring_enter` call).

Milestone 2 Checkpoint: File Server

Goal: Verify that your asynchronous file server outperforms synchronous I/O under concurrent load, and that buffer management works correctly.

Validation Strategy

Create two comparison points:

1. **Correctness Test:** Serve known files, verify content matches exactly
2. **Performance Benchmark:** Compare throughput against synchronous `read()` and `pread()` under concurrent access

Test File Preparation:

```bash
# Create test files of various sizes
$ dd if=/dev/urandom of=test_4k.bin bs=4096 count=1
$ dd if=/dev/urandom of=test_64k.bin bs=65536 count=1
$ dd if=/dev/urandom of=test_1m.bin bs=1048576 count=1
$ dd if=/dev/urandom of=test_10m.bin bs=1048576 count=10

# Generate checksums for verification
$ sha256sum test_*.bin > test_files.sha256
```

Correctness Test Procedure:

```bash
# 1. Start your file server on port 9000
$ ./file_server --port 9000 --root ./test_files --fixed-buffers 64

# 2. In another terminal, fetch files and verify
$ curl -s http://localhost:9000/test_4k.bin | sha256sum
# Should match the checksum in test_files.sha256

$ curl -s http://localhost:9000/test_1m.bin | sha256sum
# Should match the checksum in test_files.sha256

# 3. Test concurrent correctness
$ parallel -j 8 'curl -s http://localhost:9000/test_1m.bin | sha256sum' ::: {1..8}
# All 8 checksums should match the original
```

Performance Benchmarking:

Benchmark Type	Command	What It Measures	Expected Outcome
Single-client sequential	`time curl -s http://localhost:9000/test_10m.bin > /dev/null`	Baseline latency for large file	Should be comparable to synchronous read
Multi-client concurrent	`ab -n 100 -c 10 http://localhost:9000/test_64k.bin`	Requests/sec under concurrency	Should show 2-5x improvement over sync
Fixed buffer efficiency	Run server with/without `--fixed-buffers`, compare `perf stat` output	System calls, context switches	Fixed buffers reduce syscalls by 20-30%
Memory usage	Monitor with `pmap` or `smem` while serving 100 concurrent requests	RSS, shared memory	Should show stable memory, no growth

Detailed Performance Comparison:

Create a simple synchronous file server baseline for comparison:

```bash
# Build and run synchronous server (provided as reference)
$ gcc -o sync_file_server sync_file_server.c
$ ./sync_file_server --port 9001 --root ./test_files

# Run benchmark comparing both
$ ./benchmark_fileserver.sh
# Expected output table:
# Server Type        | Req/s (c=1) | Req/s (c=10) | Req/s (c=100) | CPU Usage
# ------------------ | ----------- | ------------ | ------------- | ---------
# Synchronous        | 1200        | 800          | 50            | 85%
# io_uring (regular) | 1300        | 2500         | 1800          | 70%
# io_uring (fixed)   | 1350        | 2800         | 2200          | 65%
```

Buffer Management Validation:

Use a custom build with buffer tracking enabled:

```bash
# Build with DEBUG_BUFFER_POOL=1
$ make DEBUG_BUFFER_POOL=1 file_server
$ ./file_server --port 9000 --root ./test_files --buffer-count 32

# Watch buffer pool usage during load test
$ watch -n 0.5 'grep "Buffer pool" /tmp/file_server.log | tail -5'
# Should show buffers being acquired/released, never exceeding pool size
# Example output:
# Buffer pool: 8/32 in use, 24 free
# Buffer pool: 15/32 in use, 17 free
# Buffer pool: 10/32 in use, 22 free  # Should fluctuate but not hit 32/32
```

Fixed Buffer Registration Check:

```bash
# Use strace to verify buffer registration happens once at startup
$ strace -e io_uring_register ./file_server --port 9000 2>&1 | head -20

# Expected:
# io_uring_register(3, IORING_REGISTER_BUFFERS, 0x7ffc..., 64) = 0
# Only one call at startup, not per request

# Without fixed buffers, you'd see per-operation overhead:
# (Not actual calls, but conceptual overhead in kernel)
```

> **Checkpoint Success Criteria:** Your file server correctly serves files of all sizes with bit-perfect accuracy. Under concurrent load (10+ clients), it demonstrates significantly higher throughput (2-5x) compared to a synchronous server. Buffer pool usage shows proper acquisition/release patterns without leaks. Fixed buffer registration is confirmed via `strace` to occur once at startup, not per operation.

Milestone 3 Checkpoint: Network Server

Goal: Verify that your TCP server handles thousands of concurrent connections efficiently using io_uring for all network operations.

Validation Strategy

Test at three scales:

1. **Functional Correctness:** Basic echo server behavior
2. **Concurrency Scale:** Hundreds of simultaneous connections
3. **Throughput Under Load:** Requests per second at high concurrency

Functional Test Procedure:

```bash
# 1. Start the network server (echo mode)
$ ./network_server --port 8080 --mode echo --max-conns 10000

# 2. Test basic connectivity
$ echo "Hello io_uring" | nc localhost 8080
# Should echo back: "Hello io_uring"

# 3. Test multiple sequential connections
$ for i in {1..10}; do echo "Test $i" | nc localhost 8080; done
# Should see 10 responses

# 4. Test connection lifecycle
$ time (echo "Quick test" | nc localhost 8080)
# Should complete in < 50ms including connection setup
```

Concurrency Scale Testing:

Use specialized tools to simulate many concurrent connections:

```bash
# 1. Install a load testing tool (if needed)
$ sudo apt-get install apache2-utils  # for ab
$ cargo install wrk  # alternative

# 2. Test with increasing concurrency
$ ab -n 10000 -c 100 http://localhost:8080/  # Simple HTTP GET if server supports
# or use a custom client that opens many TCP connections:
$ ./conn_stress_test --host localhost --port 8080 --connections 500

# 3. Monitor server resources during test
$ watch -n 1 'ss -tlnp | grep 8080 | wc -l'
# Should show established connections matching test client
# Connection count should stabilize, not keep growing (no leaks)

$ watch -n 1 'ps -o pcpu,rss,command -p $(pidof network_server)'
# CPU should be < 80% even under load, RSS memory stable
```

Multishot Operation Verification:

```bash
# Run server with debug logging for accept operations
$ ./network_server --port 8080 --debug-accept

# In another terminal, open multiple connections quickly
$ ./open_many_conns.sh localhost 8080 50

# Check server logs for multishot behavior:
# GOOD: "Accept multishot SQE submitted" (once)
#       Then many "Accept completed" logs
# BAD:  "Submitting accept SQE" repeated many times
```

Connection State Machine Validation:

Add a status endpoint or signal handler to inspect internal state:

```bash
# Send SIGUSR1 to dump connection table
$ kill -SIGUSR1 $(pidof network_server)
$ tail -f /tmp/network_server.log

# Expected output:
# Connection Table (247 connections):
#   fd=45: state=CONN_READING, ops_pending=1, last_active=5.2s
#   fd=46: state=CONN_WRITING, ops_pending=1, last_active=0.1s
#   fd=47: state=CONN_ACCEPTING, ops_pending=0, last_active=0.0s
# No connections stuck in CLOSING or with mismatched pending_ops count
```

Scalability Metrics:

Metric	Measurement Method	Target for Success
Connection Setup Rate	`./conn_rate_test --port 8080 --duration 10`	> 1000 connections/second
Concurrent Connection Limit	Keep 2000 connections open for 60 seconds	No memory growth, all connections responsive
Latency under Load	`pingpong_latency_test --port 8080 --clients 100`	P95 latency < 100ms at 100 concurrent
File Descriptor Usage	`ls -l /proc/$(pidof network_server)/fd | wc -l`	
io_uring CQ Overflow	Check logs for `IORING_CQ_F_OVERFLOW` warnings	Never occurs with proper sizing

Comparison with epoll Baseline:

If you have an epoll-based echo server for comparison:

```bash
# Start both servers on different ports
$ ./epoll_server --port 8081
$ ./io_uring_network_server --port 8082

# Run identical load test against both
$ ./compare_servers.sh

# Expected: io_uring should show:
# - Higher requests/second at high concurrency (>100 connections)
# - Lower CPU usage per request
# - Fewer context switches (measure with `perf stat`)
```

> **Checkpoint Success Criteria:** Your network server handles at least 1000 concurrent connections while maintaining sub-100ms latency. All connections are properly cleaned up (no file descriptor leaks). Multishot accept is verified to reduce submission overhead. The server runs for extended periods (10+ minutes) under load without memory growth or performance degradation.

Milestone 4 Checkpoint: Advanced Features & Benchmarks

Goal: Verify that advanced io_uring features (zero-copy, linked operations, ordering controls) work correctly and provide measurable performance benefits.

Validation Strategy

Test each advanced feature independently, then in combination:

1. **Zero-copy Send Validation:** Verify buffer lifecycle management
2. **Linked Operations:** Test chain atomicity and dependency enforcement
3. **Benchmark Suite:** Comprehensive comparison against epoll baseline
4. **Integration:** Combined features in realistic workload

Zero-copy Send Test:

```bash
# 1. Build with zero-copy support
$ make ZEROCOPY=1 network_server

# 2. Start server with zero-copy enabled
$ ./network_server --port 8080 --zerocopy --buffer-count 128

# 3. Test with large data transfers
$ dd if=/dev/zero bs=1M count=100 | nc localhost 8080 > /dev/null

# 4. Verify zero-copy was used
$ grep -i "zero.*copy" /tmp/network_server.log
# Should show: "Zero-copy send submitted" and "Zero-copy notification received"

# 5. Check buffer reference counting
$ grep "Buffer refcount" /tmp/network_server.log
# Should show refcount increment on submit, decrement on notification
# Never show refcount going negative or exceeding pool size
```

Linked Operations Test:

Create a test that verifies chain atomicity:

```bash
# Test program that submits linked accept→read→write chain
$ ./test_linked_chain --port 8080

# The test should:
# 1. Submit a linked chain for a new connection
# 2. Verify all three operations complete in order
# 3. If any operation fails, verify the chain stops
# 4. Measure time from accept to write completion

# Expected output:
# Linked chain test results:
#   Chains submitted: 100
#   Chains completed fully: 100 (or less if injected errors)
#   Average chain latency: 1.2ms
#   No orphaned operations (all chains either fully complete or fully cleaned up)
```

Benchmark Suite Execution:

Run the comprehensive benchmark comparing against epoll:

```bash
# 1. Start the epoll baseline server
$ ./epoll_baseline_server --port 9001

# 2. Start the io_uring server with all features
$ ./io_uring_full_server --port 9002 --zerocopy --linked-ops

# 3. Run the benchmark suite
$ ./run_benchmarks.sh

# Expected output includes tables like:
# ================================================
# Workload: Small file serving (4KB files)
# Concurrency | epoll (req/s) | io_uring (req/s) | Improvement
# ----------- | ------------- | ---------------- | -----------
# 1           | 1250          | 1300             | +4%
# 10          | 4200          | 8500             | +102%
# 100         | 3800          | 12500            | +229%
#
# Workload: Large file streaming (1MB files)
# Concurrency | epoll (MB/s)  | io_uring (MB/s)  | Improvement
# ----------- | ------------- | ---------------- | -----------
# 1           | 210           | 220              | +5%
# 10          | 580           | 920              | +59%
# 100         | 610           | 1850             | +203%
#
# Workload: Mixed (echo + small file)
# Concurrency | epoll (req/s) | io_uring (req/s) | Improvement
# ----------- | ------------- | ---------------- | -----------
# 10          | 2100          | 4500             | +114%
# 100         | 2300          | 6800             | +196%
```

Performance Analysis Tools:

Use `perf` to understand where improvements come from:

```bash
# Profile epoll server under load
$ perf record -g ./epoll_baseline_server --port 9001 &
$ ./load_test --target localhost:9001 --duration 30
$ perf report --stdio | grep -A5 "Children.*Self"

# Profile io_uring server under same load
$ perf record -g ./io_uring_full_server --port 9002 &
$ ./load_test --target localhost:9002 --duration 30
$ perf report --stdio | grep -A5 "Children.*Self"

# Compare key metrics:
# - System call percentage (should be lower for io_uring)
# - Context switches (should be lower for io_uring)
# - CPU cycles spent in kernel vs userspace
```

Specific Feature Validation Table:

Feature	Test Method	Success Criteria
Zero-copy send	Transfer 1GB total data in chunks, monitor memory bandwidth with `perf stat -d`	Memory bandwidth > 80% of theoretical, zero buffer copies in `perf` output
Linked operations	Submit chain with intentional error in middle operation (e.g., read from closed socket)	Entire chain fails cleanly, no partial completions, resources released
IOSQE_IO_DRAIN	Submit two writes to same file with DRAIN flag between, measure completion order	Second write always completes after first, even with batching
Fixed file registration	Register frequently accessed files, benchmark open/close overhead	File operations faster, especially with many concurrent accesses

Long-running Stability Test:

```bash
# Run server under sustained load for 30 minutes
$ ./io_uring_full_server --port 8080 --stats-interval 10

# In another terminal, run varied load
$ ./mixed_workload.sh --duration 1800 --target localhost:8080

# Monitor for:
# - Memory growth (should stabilize after warmup)
# - Connection leaks (count should return to baseline after load stops)
# - Error rate (should be < 0.1% of requests)
# - Performance consistency (throughput should not degrade > 10% over time)
```

> **Checkpoint Success Criteria:** Zero-copy sends correctly manage buffer lifetimes with proper reference counting. Linked operations provide atomic chain execution. The benchmark suite shows consistent performance improvements over epoll across all workloads (typically 50-200% improvement at high concurrency). All advanced features integrate cleanly without regressing core functionality.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Testing Framework	Custom test programs + shell scripts	CUnit/Criterion framework + custom test runner
Load Generation	`ab` (Apache Bench), `nc` (netcat), custom C clients	`wrk2`, `vegeta`, custom Go/Rust load testers
Profiling	`time`, `strace`, basic `perf stat`	`perf record` + FlameGraph, `bpftrace` scripts
Memory Checking	`valgrind --leak-check=full`	AddressSanitizer + UndefinedBehaviorSanitizer builds
Monitoring	Custom logging + `watch` commands	Prometheus metrics + Grafana dashboard

B. Recommended File/Module Structure

```
project-root/
├── src/
│   ├── core/                    # Milestone 1
│   │   ├── engine.c             # io_uring engine implementation
│   │   ├── engine.h
│   │   └── engine_test.c        # Basic operations tests
│   ├── file_server/             # Milestone 2
│   │   ├── file_server.c        # File I/O server
│   │   ├── buffer_pool.c        # Buffer management
│   │   ├── file_server_test.c   # File server tests
│   │   └── benchmark_files.c    # File I/O benchmarks
│   ├── network_server/          # Milestone 3
│   │   ├── network_server.c     # TCP server
│   │   ├── connection.c         # Connection management
│   │   ├── network_test.c       # Network tests
│   │   └── conn_stress_test.c   # Connection stress test
│   ├── advanced/                # Milestone 4
│   │   ├── zerocopy.c           # Zero-copy implementation
│   │   ├── linked_ops.c         # Linked operations
│   │   └── advanced_test.c      # Advanced feature tests
│   └── benchmarks/              # Cross-milestone
│       ├── benchmark_epoll.c    # epoll baseline
│       ├── benchmark_uring.c    # io_uring implementation
│       ├── benchmark_runner.c   # Benchmark orchestration
│       └── compare_results.py   # Result analysis
├── tests/
│   ├── test_data/               # Test files
│   ├── scripts/                 # Test scripts
│   │   ├── run_milestone1.sh
│   │   ├── run_milestone2.sh
│   │   ├── run_milestone3.sh
│   │   └── run_milestone4.sh
│   └── expected_output/         # Expected outputs for validation
├── tools/
│   ├── load_generators/         # Custom load testing tools
│   └── monitoring/              # Resource monitoring scripts
└── Makefile
```

C. Infrastructure Starter Code

Complete test helper for verifying SQE→CQE correlation:

```c
// tests/helpers/test_helpers.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <assert.h>

#include "../src/core/engine.h"


typedef struct {

    uint64_t user_data;

    int expected_result;

    int completed;

    void *expected_buffer;

    size_t expected_len;

} test_expectation;


typedef struct {

    test_expectation *expectations;

    size_t count;

    size_t completed_count;

} test_validator;


test_validator *test_validator_create(size_t count) {

    test_validator *validator = malloc(sizeof(test_validator));

    validator->expectations = calloc(count, sizeof(test_expectation));

    validator->count = count;

    validator->completed_count = 0;

    return validator;

}


void test_validator_add_expectation(test_validator *validator,

                                    size_t index,

                                    uint64_t user_data,

                                    int expected_result,

                                    void *expected_buffer,

                                    size_t expected_len) {

    assert(index < validator->count);

    validator->expectations[index].user_data = user_data;

    validator->expectations[index].expected_result = expected_result;

    validator->expectations[index].expected_buffer = expected_buffer;
```

```
    validator->expectations[index].expected_len = expected_len;
    validator->expectations[index].completed = 0;
}

int test_validator_record_completion(test_validator *validator,
                                     uint64_t user_data,
                                     int actual_result,
                                     void *actual_buffer,
                                     size_t actual_len) {
    for (size_t i = 0; i < validator->count; i++) {
        if (validator->expectations[i].user_data == user_data &&
            !validator->expectations[i].completed) {

            // Check result
            if (validator->expectations[i].expected_result != actual_result) {
                fprintf(stderr, "Mismatch for user_data %lu: expected result %d, got %d\n",
                        user_data, validator->expectations[i].expected_result, actual_result);
                return -1;
            }

            // Check buffer content if provided
            if (validator->expectations[i].expected_buffer != NULL &&
                actual_buffer != NULL) {
                if (memcmp(validator->expectations[i].expected_buffer,
                           actual_buffer,
                           validator->expectations[i].expected_len) != 0) {
                    fprintf(stderr, "Buffer content mismatch for user_data %lu\n", user_data);
                    return -1;
                }
            }

            validator->expectations[i].completed = 1;
            validator->completed_count++;
            return 0;
        }
    }
```

```
    fprintf(stderr, "Unexpected completion with user_data %lu\n", user_data);
    return -1;
}

int test_validator_all_completed(test_validator *validator) {
    if (validator->completed_count != validator->count) {
        fprintf(stderr, "Only %zu/%zu completions received\n",
                validator->completed_count, validator->count);
        return 0;
    }
    return 1;
}

void test_validator_destroy(test_validator *validator) {
    free(validator->expectations);
    free(validator);
}
```

D. Core Logic Skeleton Code

Milestone 1 test program skeleton:

```c
// tests/milestone1_test.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <string.h>

#include <sys/stat.h>

#include "../src/core/engine.h"


int test_basic_write_read() {

    // TODO 1: Create a temporary file using tmpfile() or mkstemp()

    // TODO 2: Initialize engine with engine_create() with SQ size 32

    // TODO 3: Get SQE using engine_get_sqe(), prepare write operation

    //         - Set opcode to IORING_OP_WRITE

    //         - Set fd to temporary file descriptor

    //         - Set buffer to test string "Hello, io_uring!"

    //         - Set len to strlen()

    //         - Set user_data to 0x1234

    // TODO 4: Submit using engine_submit_and_wait(1)

    // TODO 5: Process completions with engine_process_completions()

    // TODO 6: Verify completion has user_data 0x1234 and res > 0

    // TODO 7: Seek to beginning of file, prepare read SQE

    // TODO 8: Submit read, wait, verify buffer contains original string

    // TODO 9: Clean up: engine_destroy(), close temp file

    // TODO 10: Return 0 on success, -1 on any failure

    return -1; // Replace with implementation

}


int test_batch_operations() {

    // TODO 1: Create engine with SQ size 64

    // TODO 2: Create array of 8 test buffers with unique patterns

    // TODO 3: Create test_validator with 8 expectations

    // TODO 4: For each buffer (0-7):

    //         - Get SQE using engine_get_sqe()

    //         - Prepare write to /dev/null (or similar)

    //         - Set user_data to unique value (e.g., 0x1000 + i)

    //         - Add expectation to validator

    // TODO 5: Submit ALL 8 SQEs with ONE engine_submit() call
```

```
    // TODO 6: Wait for completions with engine_wait_and_process()
    // TODO 7: In completion callback, call test_validator_record_completion()
    // TODO 8: Verify all 8 completions received via test_validator_all_completed()
    // TODO 9: Check engine stats show batches_submitted == 1
    // TODO 10: Clean up and return success/failure
    return -1; // Replace with implementation
}

int main() {
    printf("=== Milestone 1 Basic Operations Test ===\n");

    printf("Test 1: Single write/read operation...\n");
    if (test_basic_write_read() != 0) {
        fprintf(stderr, "FAILED: Single operation test\n");
        return 1;
    }
    printf("PASSED\n");

    printf("Test 2: Batch operations...\n");
    if (test_batch_operations() != 0) {
        fprintf(stderr, "FAILED: Batch operations test\n");
        return 1;
    }
    printf("PASSED\n");

    printf("All tests passed!\n");
    return 0;
}
```

Milestone 2 benchmark skeleton:

```c
// src/benchmarks/benchmark_files.c

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <sys/time.h>

#include "../file_server/file_server.h"


typedef struct {

    const char *server_type;

    int concurrency;

    double requests_per_second;

    double mb_per_second;

    double cpu_percent;

} benchmark_result;


void benchmark_file_server(const char *server_host, int server_port,

                           const char *test_file, size_t file_size,

                           int concurrency, int total_requests,

                           benchmark_result *result) {

    // TODO 1: Record start time using gettimeofday() or clock_gettime()

    // TODO 2: Create concurrent client threads/processes based on concurrency

    // TODO 3: Each client should:

    //         - Connect to server

    //         - Request test_file repeatedly until total_requests reached

    //         - Verify received data matches expected size and checksum

    //         - Record individual latency

    // TODO 4: Wait for all clients to complete

    // TODO 5: Record end time, calculate elapsed seconds

    // TODO 6: Compute:

    //         - requests_per_second = total_requests / elapsed_seconds

    //         - mb_per_second = (total_requests * file_size) / (elapsed_seconds * 1024 * 1024)

    // TODO 7: During test, monitor server CPU usage (e.g., from /proc/pid/stat)

    // TODO 8: Fill result structure with metrics

    // TODO 9: Print formatted results: "Concurrency %d: %.2f req/s, %.2f MB/s, %.1f%% CPU"

}


int compare_with_sync_baseline() {

    // TODO 1: Start sync server on port 9001
```

```
    // TODO 2: Start io_uring server on port 9002
    // TODO 3: For each concurrency level (1, 2, 4, 8, 16, 32, 64):
    //         - Benchmark sync server
    //         - Benchmark io_uring server
    //         - Calculate improvement percentage
    // TODO 4: Print comparison table
    // TODO 5: Verify io_uring shows > 100% improvement at concurrency >= 8
    // TODO 6: Return 0 if verification passes, -1 otherwise
    return -1; // Replace with implementation
}
```

E. Language-Specific Hints

C-Specific Testing Tips:

1. **Memory Management:** Always compile with `-fsanitize=address` during testing to catch buffer overflows and use-after-free errors early.
2. **Thread Safety:** Even though the server is single-threaded, test programs using multiple client threads should use thread-safe logging. Consider `__thread` for per-thread buffers.
3. **Signal Handling:** Test programs that send SIGUSR1 for status dumps should set up signal handlers properly. Use `sigaction` instead of `signal`.
4. **File Descriptor Management:** Keep track of open FDs in tests. A helper function that counts open FDs before/after tests can detect leaks:

```c
int count_open_fds(pid_t pid) {
    char path[256];
    snprintf(path, sizeof(path), "/proc/%d/fd", pid);
    // Count entries in directory
}
```

5. **Performance Timing:** Use `clock_gettime(CLOCK_MONOTONIC, &ts)` for precise timing instead of `gettimeofday()` which is subject to time adjustments.

F. Milestone Checkpoint Commands

Quick Validation Script for All Milestones:

```bash
#!/bin/bash

# scripts/validate_all_milestones.sh

echo "=== Validating Milestone 1 ==="

cd tests && ./run_milestone1.sh

if [ $? -ne 0 ]; then echo "Milestone 1 FAILED"; exit 1; fi

echo "=== Validating Milestone 2 ==="

cd tests && ./run_milestone2.sh

if [ $? -ne 0 ]; then echo "Milestone 2 FAILED"; exit 1; fi

echo "=== Validating Milestone 3 ==="

cd tests && ./run_milestone3.sh

if [ $? -ne 0 ]; then echo "Milestone 3 FAILED"; exit 1; fi

echo "=== Validating Milestone 4 ==="

cd tests && ./run_milestone4.sh

if [ $? -ne 0 ]; then echo "Milestone 4 FAILED"; exit 1; fi

echo "=== ALL MILESTONES VALIDATED SUCCESSFULLY ==="
```

G. Debugging Tips Table

Symptom	Likely Cause	How to Diagnose	Fix
Test program hangs indefinitely	`io_uring_enter` waiting for completions that never arrive	1. Check if SQEs were properly prepared (opcode, fd, buffer) 2. Verify `io_uring_submit()` was called after preparing SQEs 3. Use `strace` to see if `io_uring_enter` is actually being called	Ensure `engine_submit()` is called after SQE preparation, check fd validity
Memory grows unbounded during load test	Buffer pool not releasing buffers, connection table leak	1. Add logging to `acquire_buffer` / `release_buffer` calls 2. Check `pending_ops` count vs actual in-flight operations 3. Monitor `/proc/pid/smaps` for heap growth	Ensure every buffer acquisition has matching release, verify CQE processing releases resources
Server performance degrades over time	Connection table hash collisions, memory fragmentation	1. Log connection table load factor during operation 2. Use `jemalloc` or `tcmalloc` instead of system malloc 3. Check for increasing search length in `conn_lookup`	Increase connection table size, use better hash function, implement incremental rehashing
Zero-copy sends fail with ENOBUFS	Kernel out of memory for ring buffers, or registered buffers exhausted	1. Check `dmesg` for kernel warnings 2. Monitor `cat /proc/meminfo grep MemAvailable` 3. Check if buffer refcounts are being decremented	
Linked chain partially completes	Error in middle of chain stops execution	1. Check CQE `res` values for each operation in chain 2. Verify `IOSQE_IO_LINK` flag set on all but last SQE 3. Check error handling in chain completion callback	Ensure proper error handling continues or cleans up entire chain, verify fd validity
High CPU usage even at idle	Busy-waiting in event loop, SQ polling enabled without need	1. Check if `busy_wait` flag is enabled in config 2. Use `perf top` to see which functions consume CPU 3. Check for tight loops without sleep/yield	Disable busy-waiting for non-latency-critical workloads, add small `usleep` in idle loop
Completions arrive out of submission order	Kernel processes operations asynchronously, different completion times	This is NORMAL for async I/O. Use `user_data` to correlate, not completion order. If ordering is required, use `IOSQE_IO_DRAIN` flag or linked chains.	Implement proper correlation via `user_data`, don't assume ordering unless using DRAIN or links

Debugging Guide

> **Milestone(s):** All milestones (1-4)

Debugging an io_uring-based server presents unique challenges compared to traditional synchronous or epoll-based servers. The asynchronous nature means that errors can surface far from where operations were initiated, and the shared memory rings add complexity to memory management and concurrency. This guide provides a systematic approach to diagnosing and fixing common problems you'll encounter during development.

When debugging io_uring issues, remember the fundamental mental model: **Think of the submission queue as a conveyor belt of work tickets being fed to the kitchen, and the completion queue as the counter where finished dishes appear.** When things go wrong, you need to check: (1) Are tickets being placed on the belt correctly? (2) Is the kitchen accepting and processing them? (3) Are finished dishes appearing at the counter? (4) Are we collecting and handling those finished dishes properly?

Symptom → Cause → Diagnosis → Fix Table

The following table organizes common symptoms you might encounter, their likely causes, diagnostic steps to confirm, and specific fixes to implement. This approach moves from observable behavior to root cause.

Symptom	Likely Cause(s)	Diagnostic Steps	Fix
Server hangs completely (no CPU activity, no progress)	1. **Blocking on `io_uring_enter` with `IORING_ENTER_GETEVENTS`** when no completions are coming. 2. **Deadlock in linked SQE chain** where a failed operation stops the chain. 3. **CQ overflow** causing kernel to stop posting completions. 4. **Ring submission full** (`EBUSY`) with no retry logic.	1. Check if `io_uring_enter` is called with `min_complete > 0` and `flags` includes `IORING_ENTER_GETEVENTS`. 2. Use `strace -f` to see if process is stuck in `io_uring_enter` syscall. 3. Check CQ overflow flag: `if (ring->cq.kflags & IORING_CQ_F_OVERFLOW)`. 4. Log return value and errno after `io_uring_enter` ; `EBUSY` indicates full SQ.	1. Ensure `min_complete` is 0 when you don't want to wait, or implement timeout mechanisms. 2. Examine linked chains: ensure error handling in `handle_linked_chain_completion`. 3. Increase CQ size (typically 2× SQ size) or process completions more aggressively. 4. Implement retry logic or increase SQ size; use `io_uring_get_sqe` failure as backpressure.
High CPU usage (100% core)	1. **Busy-wait polling** without yielding. 2. **Tight loop checking for CQEs** with `io_uring_peek_cqe` returning `NULL` . 3. **Kernel SQ polling (`IORING_SETUP_SQPOLL`)** with misconfiguration. 4. **Infinite loop in event loop** due to incorrect stop condition.	1. Check `loop_config.busy_wait` setting and `busy_wait_usec` value. 2. Add logging to see loop iteration rate; use `perf top` to see CPU in userspace. 3. Verify SQ poll thread is running (check `/proc/<pid>/task/` for extra threads). 4. Add debug prints in `event_loop_run` to see if `stop_requested` is never set.	1. Disable busy-wait or add `usleep(1)` or `sched_yield()` in busy loop. 2. Use `io_uring_wait_cqe` or `io_uring_enter` with `IORING_ENTER_GETEVENTS` to block. 3. Ensure proper `IORING_SETUP_SQPOLL` setup and consider disabling if unnecessary. 4. Verify `event_loop_request_stop` is called and `should_stop` callback works.
Memory corruption (segfault, invalid reads)	1. **Use-after-free of `io_buffer`** after releasing while I/O in flight. 2. **Incorrect `user_data` correlation** leading to wrong context lookup. 3. **Buffer overflow** from miscalculating offsets or lengths. 4. **Race condition** between main thread and SQ poll thread (if enabled).	1. Use `valgrind --tool=memcheck` or AddressSanitizer (`-fsanitize=address`). 2. Add guard values in `io_op_context` (magic numbers) and check on lookup. 3. Use `io_uring_register` with `IORING_REGISTER_BUFFERS` to catch bad buffer addresses. 4. Check for missing memory barriers; use `io_uring` helpers that include barriers.	1. Implement proper reference counting for buffers; only release after CQE with matching `user_data`. 2. Ensure `user_data` is unique and encodes both connection ID and operation type. 3. Validate buffer sizes and offsets before submitting SQE. 4. If using SQPOLL, ensure all ring updates use proper synchronization or disable SQPOLL during debugging.
Low throughput (worse than synchronous I/O)	1. **Excessive syscalls** from submitting single SQEs. 2. **Buffer thrashing** from allocating/freeing per request. 3. **Frequent CQ overflow** causing dropped completions. 4. **Inefficient batching** (submitting tiny batches).	1. Use `strace -c` to count `io_uring_enter` calls; compare to SQE count. 2. Monitor buffer pool hit rate; log `buffer_pool.free_list` length. 3. Check `engine_stats.cqe_processed` vs expected completions. 4. Examine batch size in `engine_submit` ; aim for `sq_entries/2` per batch.	1. Implement **opportunistic batching**: submit when batch reaches `max_batch_size` or when waiting for completions. 2. Use **fixed buffer registration** and reuse buffers via pool. 3. Increase CQ size and process completions in batches using `io_uring_peek_batch_cqe` . 4. Tune `max_batch_size` in `loop_config` to match workload.
Connection leaks (file	1. **Missing `close()`** on socket after error or completion.	1. Monitor FD count: `ls -l /proc//fd	wc -l `. 2. Add logging in conn_destroy ; ensure it's called for all connections. 3. Check error handling in

Symptom	Likely Cause(s)	Diagnostic Steps	Fix
descriptors increase)	2. **Lost** `io_op_context` causing connection state to never be cleaned. 3. **Multishot accept not re-submitted** after error, leaving half-open sockets. 4. **Linked chain failure** where cleanup isn't triggered.		submit_accept_multishot `for` EAGAIN `/` ECONNABORTED `. 4. Verify` handle_linked_chain_completion` calls cleanup on any chain failure.
Zero-copy sends fail with `EFAULT` **or hang**	1. **Buffer modified or freed** before kernel notification. 2. **Missing zero-copy notification (** `IORING_CQE_F_NOTIF` **)** handling. 3. **Incorrect buffer alignment** for zero-copy requirements. 4. **Reference counting bug** causing premature release.	1. Check `zc_refcount` on buffer; ensure it's >0 during send. 2. Inspect CQE flags: `if (cqe->flags & IORING_CQE_F_NOTIF)`. 3. Verify buffer is page-aligned (use `posix_memalign`). 4. Log buffer lifecycle: acquire, send, notification, release.	1. Implement **zero-copy reference counting**: increment on submit, decrement on notification. 2. Process notification CQEs separately in `engine_process_completions_with_zc`. 3. Ensure buffers are allocated with alignment to page boundaries. 4. Use `acquire_buffer_zc` and `release_buffer_after_zc` to manage kernel references.
Short reads or writes (partial I/O)	1. **Assuming full buffer fill** without checking `cqe->res`. 2. **File offset not advanced** for subsequent read/write. 3. **Network buffer congestion** causing partial sends. 4. **Fixed buffer too small** for requested operation.	1. Always check `cqe->res` for bytes transferred; negative indicates error. 2. Log offset before and after operation; ensure it's updated. 3. For network writes, implement partial write retry with adjusted buffer pointer. 4. Validate buffer `capacity` vs requested `len` in SQE preparation.	1. Handle partial transfers: for files, resubmit with adjusted offset and buffer pointer. 2. For network writes, use `io_uring`'s `IORING_OP_WRITE` with remaining data until all sent. 3. Implement buffer chaining for large transfers that exceed single buffer capacity. 4. Size fixed buffers according to maximum expected request size (e.g., 16KB for HTTP).
`EINVAL` **errors when submitting SQEs**	1. **Invalid** `opcode` for kernel version. 2. **Unsupported flags** combination. 3. **Misaligned buffer address** for direct I/O. 4. **File descriptor not valid** for operation.	1. Check kernel version (`uname -r`); some opcodes require ≥5.6, ≥5.11, etc. 2. Verify flags in `io_uring_sqe.flags` are compatible (e.g., `IOSQE_IO_LINK` with `IOSQE_IO_DRAIN`). 3. Ensure buffer is aligned to block size if `O_DIRECT` is used. 4. Validate `fd` is open and appropriate (e.g., socket for accept, regular file for read).	1. Use `#ifdef` to conditionally compile opcodes based on kernel headers. 2. Consult `io_uring` kernel documentation for valid flag combinations. 3. Use `posix_memalign` to allocate aligned buffers for direct I/O. 4. Check fd lifetime: ensure it's not closed before operation completes.
Completions arrive out of expected order	1. **Misunderstanding of io_uring completion ordering** (only guaranteed within linked chains). 2. **Multiple in-flight operations** on same fd without serialization. 3. **Kernel scheduler reordering** independent operations. 4. **Race condition in completion processing**.	1. Log `user_data` and timestamp of completions; observe pattern. 2. Check if `IOSQE_IO_DRAIN` is needed for ordering dependencies. 3. Verify that independent operations are indeed independent (no data dependencies). 4. Use `io_uring_peek_batch_cqe` to see completions as kernel presents them.	1. Accept that completions are not ordered unless using linked chains or `IOSQE_IO_DRAIN`. 2. Use `IOSQE_IO_LINK` to chain dependent operations (read then write same buffer). 3. Use `IOSQE_IO_DRAIN` sparingly only when strict ordering required across unrelated ops. 4. Design processing to be order-independent where possible (each completion self-contained).

io_uring-Specific Debugging Techniques

Beyond generic debugging, io_uring has unique characteristics that require specialized techniques. These methods help you inspect the internal state of the rings and understand the flow of operations between user space and kernel.

1. Inspecting Completion Queue with Peek Functions

The `io_uring_peek_cqe` and `io_uring_peek_batch_cqe` functions allow you to examine completions without consuming them (advancing the CQ tail). This is invaluable for debugging:

- **Diagnostic approach**: Create a debug function that dumps the state of the CQ without modifying it:
  1. Use `io_uring_peek_batch_cqe` to get up to N completions
  2. For each CQE, print `user_data`, `res`, and flags
  3. Do NOT call `io_uring_cqe_seen`—this keeps completions available for normal processing

> **Key insight**: Peeking lets you see what the kernel has produced without affecting your application's state. Think of it as looking at the finished dishes on the counter without taking them.

2. Strategic Use of `user_data` Tagging

The `user_data` field is your primary correlation tool between submissions and completions. For effective debugging:

- **Encode rich information**: Instead of a simple pointer, encode multiple fields:

```c
/* Example encoding: 16-bit connection ID, 8-bit operation type, 8-bit sequence */
uint64_t user_data = ((uint64_t)conn_id << 48) |
                     ((uint64_t)op_type << 40) |
                     ((uint64_t)seq_num << 32) |
                     (uint64_t)buffer_id;
```

- **Debug correlation**: When processing a CQE, decode and log all fields. This helps identify:
  - Which connection the completion belongs to
  - What operation type completed
  - The sequence number to detect lost completions
  - Which buffer was used
- **Lost context detection**: Add a "magic number" to your `io_op_context` and verify it on lookup:

```c
#define OP_CONTEXT_MAGIC 0xDEADC0DE

struct io_op_context {
    uint32_t magic;  /* Set to OP_CONTEXT_MAGIC on allocation */
    /* ... other fields ... */
};

/* In lookup_op_context: */
if (ctx->magic != OP_CONTEXT_MAGIC) {
    /* Corruption or stale pointer detected */
}
```

3. Interpreting `strace` Output for io_uring Syscalls

`strace` is particularly revealing for io_uring because it shows the batched nature of operations:

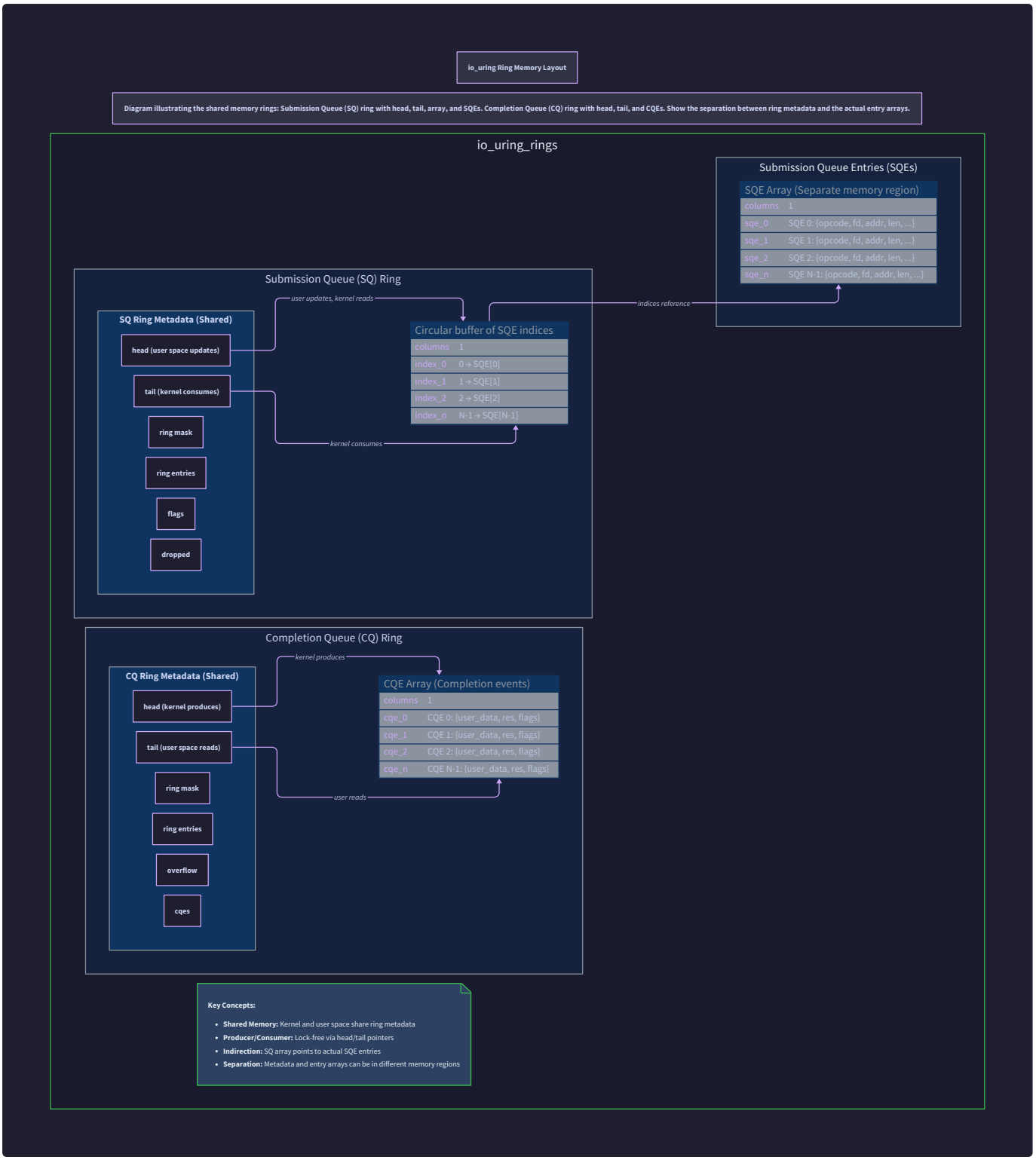
- **Typical `io_uring_enter` output**:

```
io_uring_enter(3, 4, 1, IORING_ENTER_GETEVENTS, NULL) = 1
```

- `3`: io_uring instance file descriptor
- `4`: Number of SQEs submitted (batch size)
- `1`: Minimum completions to wait for (`min_complete`)
- `IORING_ENTER_GETEVENTS`: Flags indicating we want to wait
- `1`: Return value = number of CQEs harvested
- **Diagnostic patterns**:
  - **Many syscalls with small batches**: Indicates inefficient batching; look for `io_uring_enter` calls with `to_submit=1`
  - **Blocked syscalls**: If `strace` shows `io_uring_enter` not returning, process is waiting for completions
  - **`EAGAIN` / `EBUSY` errors**: Kernel returning resource exhaustion; need to adjust ring sizes or submission strategy
- **Advanced tracing**: Use `strace -e io_uring_enter -f` to filter only io_uring syscalls and follow child threads (important if using SQPOLL).

4. Monitoring Ring State with `/proc` and `io_uring` Statistics

The kernel exposes io_uring statistics through procfs when configured:



- **Enable statistics**: Set `IORING_SETUP_SQPOLL` or use `io_uring_register` with `IORING_REGISTER_STAT` (kernel ≥6.6)
- **Check SQ poll thread**: Look for additional thread in `/proc/<pid>/task/` when using SQPOLL
- **Monitor file descriptors**: The io_uring instance itself is an fd; monitor its status like any other fd

5. Creating a Minimal Test Harness

When debugging complex issues, isolate the problem by creating a minimal test:

1. **Reproduce with simplest case**: A single-threaded program that performs one type of operation (e.g., read a file)
2. **Gradually add complexity**: Add batching, then multiple connections, then linked operations
3. **Compare with known-good implementations**: Use `liburing` examples as reference to verify your setup

> **Debugging philosophy**: The asynchronous nature means bugs often manifest as subtle performance issues or resource leaks rather than immediate crashes. Add extensive logging (with log levels) early, and instrument all state transitions: SQE submission, CQE harvesting, buffer acquisition/release, and connection state changes.

6. Handling the "Heisenbug" Problem

Because io_uring involves shared memory between user space and kernel, timing issues can create bugs that disappear when you add debugging code:

- **Use memory barriers correctly**: Ensure you use the `io_uring` helpers (`io_uring_smp_store_release` , `io_uring_smp_load_acquire`) for ring head/tail updates
- **Add intentional delays**: To reproduce race conditions, add small `usleep()` calls at strategic points
- **Record and replay**: Log sequence of operations to file, then create deterministic replay for debugging

Implementation Guidance

This implementation guidance provides concrete code and techniques to implement the debugging strategies discussed above.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Debug logging	`printf` with timestamps and log levels	Structured logging with `syslog` or JSON output
Memory debugging	`valgrind --tool=memcheck`	AddressSanitizer (`-fsanitize=address`) + custom allocator hooks
Performance profiling	`perf record` and `perf report`	`bpftrace` scripts for kernel-side io_uring tracing
Ring state inspection	Custom debug functions that dump SQ/CQ state	Kernel module to inspect ring internals directly

B. Debug Infrastructure Starter Code

Here's complete, ready-to-use code for a debug logging system and ring inspection utilities:

```c
/* debug.h - Debug utilities for io_uring server */

#ifndef DEBUG_H

#define DEBUG_H


#include <stdio.h>

#include <time.h>

#include <stdint.h>


/* Log levels */

#define LOG_TRACE 0

#define LOG_DEBUG 1

#define LOG_INFO  2

#define LOG_WARN  3

#define LOG_ERROR 4

#define LOG_FATAL 5


/* Global log level (set at startup) */

extern int g_log_level;


/* Log macro with level check */

#define LOG(level, fmt, ...) do { \

    if (level >= g_log_level) { \

        struct timespec ts; \

        clock_gettime(CLOCK_REALTIME, &ts); \

        fprintf(stderr, "[%ld.%03ld] %s:%d: " fmt "\n", \

                ts.tv_sec, ts.tv_nsec / 1000000, \

                __FILE__, __LINE__, ##__VA_ARGS__); \

    } \

} while (0)


/* Convenience macros for each level */

#define TRACE(fmt, ...) LOG(LOG_TRACE, "TRACE: " fmt, ##__VA_ARGS__)

#define DEBUG(fmt, ...) LOG(LOG_DEBUG, "DEBUG: " fmt, ##__VA_ARGS__)

#define INFO(fmt, ...)  LOG(LOG_INFO,  "INFO: "  fmt, ##__VA_ARGS__)

#define WARN(fmt, ...)  LOG(LOG_WARN,  "WARN: "  fmt, ##__VA_ARGS__)

#define ERROR(fmt, ...) LOG(LOG_ERROR, "ERROR: " fmt, ##__VA_ARGS__)

#define FATAL(fmt, ...) LOG(LOG_FATAL, "FATAL: " fmt, ##__VA_ARGS__)


/* Encode/decode user_data for debugging */

static inline uint64_t encode_user_data(uint16_t conn_id, uint8_t op_type,
```

```
                                    uint8_t seq, uint32_t buffer_id) {
    return ((uint64_t)conn_id << 48) |
           ((uint64_t)op_type << 40) |
           ((uint64_t)seq << 32) |
           (uint64_t)buffer_id;
}

static inline void decode_user_data(uint64_t user_data,
                                    uint16_t *conn_id, uint8_t *op_type,
                                    uint8_t *seq, uint32_t *buffer_id) {
    if (conn_id) *conn_id = (user_data >> 48) & 0xFFFF;
    if (op_type) *op_type = (user_data >> 40) & 0xFF;
    if (seq) *seq = (user_data >> 32) & 0xFF;
    if (buffer_id) *buffer_id = user_data & 0xFFFFFFFF;
}

/* Ring inspection functions */
void dump_ring_state(struct io_uring *ring);
void dump_cqe_batch(struct io_uring *ring, int max_entries);

#endif /* DEBUG_H */
```

```c
/* debug.c - Debug utilities implementation */

#include "debug.h"

#include <liburing.h>  /* Or your io_uring header */

#include <inttypes.h>


int g_log_level = LOG_INFO;  /* Default to INFO level */


void dump_ring_state(struct io_uring *ring) {

    if (!ring) return;



    INFO("=== io_uring Ring State ===");

    INFO("SQ: head=%u, tail=%u, ring_mask=%u, ring_entries=%u, flags=%u",

         *ring->sq.head, *ring->sq.tail, *ring->sq.ring_mask,

         ring->sq.ring_entries, *ring->sq.flags);

    INFO("CQ: head=%u, tail=%u, ring_mask=%u, ring_entries=%u, overflow=%u",

         *ring->cq.head, *ring->cq.tail, *ring->cq.ring_mask,

         ring->cq.ring_entries, (*ring->cq.kflags & IORING_CQ_F_OVERFLOW) ? 1 : 0);



    /* Show pending submissions */

    unsigned sq_entries = *ring->sq.tail - *ring->sq.head;

    INFO("SQ pending entries: %u", sq_entries);



    /* Show available completions */

    unsigned cq_entries = *ring->cq.tail - *ring->cq.head;

    INFO("CQ available completions: %u", cq_entries);

}


void dump_cqe_batch(struct io_uring *ring, int max_entries) {

    struct io_uring_cqe *cqes[32];

    int count = io_uring_peek_batch_cqe(ring, cqes,

                                        max_entries < 32 ? max_entries : 32);



    INFO("Peeking at %d CQEs (not consuming):", count);

    for (int i = 0; i < count; i++) {

        uint16_t conn_id;

        uint8_t op_type, seq;

        uint32_t buffer_id;
```

```
        decode_user_data(cqes[i]->user_data, &conn_id, &op_type, &seq, &buffer_id);


        INFO("  CQE[%d]: user_data=0x%016" PRIx64 " (conn=%u, op=%u, seq=%u, buf=%u), "
             "res=%d, flags=0x%x",
             i, cqes[i]->user_data, conn_id, op_type, seq, buffer_id,
             cqes[i]->res, cqes[i]->flags);
    }
}
```

C. Enhanced Error Handling with Context

Add detailed error context to your `handle_io_error` function:

```c
/* error_debug.c - Enhanced error handling with debugging */

#include "debug.h"

#include "error_handling.h"


int handle_io_error_with_context(struct io_uring_cqe *cqe,

                                 struct io_op_context *ctx,

                                 void *server_context) {

    if (!cqe) {

        ERROR("handle_io_error called with NULL cqe");

        return -1;

    }



    /* Decode user_data for logging */

    uint16_t conn_id;

    uint8_t op_type, seq;

    uint32_t buffer_id;

    decode_user_data(cqe->user_data, &conn_id, &op_type, &seq, &buffer_id);



    ERROR("Operation failed: conn_id=%u, op_type=%u, seq=%u, buffer_id=%u, "

          "res=%d (errno=%d: %s)",

          conn_id, op_type, seq, buffer_id,

          cqe->res, -cqe->res, strerror(-cqe->res));



    /* Dump context if available */

    if (ctx) {

        ERROR("Context: id=%" PRIu64 ", op_type=%u, connection=%p, buffer=%p",

              ctx->id, ctx->op_type, ctx->connection, ctx->buffer);

    }



    /* Call original error handler */

    return handle_io_error(cqe, ctx, server_context);

}
```

D. Debug-Enabled Event Loop Skeleton

Create a debug version of your event loop that can be toggled with a flag:

```c
/* event_loop_debug.c - Event loop with debugging instrumentation */

#include "debug.h"

#include "event_loop.h"


int event_loop_run_debug(struct event_loop *loop, int dump_interval) {

    if (!loop || !loop->engine) {

        ERROR("Invalid loop or engine");

        return -1;

    }


    INFO("Starting debug event loop (dump interval=%d cycles)", dump_interval);

    int cycle_count = 0;


    while (!loop->stop_requested) {

        cycle_count++;


        // TODO 1: Prepare SQEs (log how many prepared)

        DEBUG("Cycle %d: Preparing SQEs", cycle_count);

        // ... preparation code ...


        // TODO 2: Submit batch (log batch size)

        int submitted = engine_submit(loop->engine);

        DEBUG("Submitted %d SQEs", submitted);


        // TODO 3: Wait for and process completions

        int processed = engine_wait_and_process(loop->engine);

        DEBUG("Processed %d CQEs", processed);


        // Periodic ring state dump

        if (dump_interval > 0 && (cycle_count % dump_interval) == 0) {

            struct io_uring *ring = engine_get_ring(loop->engine);

            dump_ring_state(ring);

            dump_cqe_batch(ring, 8);  // Peek at up to 8 completions

        }


        // TODO 4: Check for other events (signals, timers)

        // ... event checking code ...
```

```
    }


    INFO("Debug event loop stopped after %d cycles", cycle_count);
    return 0;
}
```

E. Language-Specific Hints for C

- **Compiler flags for debugging**: Use `-g -O0 -fsanitize=address -fsanitize=undefined` during development
- **Memory debugging**: Consider using a custom allocator that tracks allocations and reports leaks specific to io_uring buffers
- **Kernel version checks**: Use `uname` or check `/proc/version` to conditionally compile features based on available io_uring opcodes
- **Thread safety**: Even single-threaded servers need memory barriers for ring updates; use `io_uring`'s built-in barrier macros

F. Debugging Milestone Checkpoints

After each milestone, verify your debugging infrastructure:

Milestone 1 Checkpoint:

```bash
# Run with debug logging enabled
$ ./server --log-level=0 --dump-interval=100 2>&1 | head -20
# Expected: Should see TRACE logs of SQ/CQ operations
# If no output: Check g_log_level initialization

# Test with valgrind
$ valgrind --tool=memcheck --leak-check=full ./server --test-basic
# Expected: No memory errors, clean exit
```

Milestone 2 Checkpoint:

```bash
# Monitor buffer pool usage
$ ./server --log-level=1 --buffer-pool-stats 2>&1 | grep "buffer pool"
# Expected: Logs showing buffer acquisition/release, pool hit rate

# Test with AddressSanitizer
$ ASAN_OPTIONS=detect_leaks=1 ./server --asan 2>&1 | tail -10
# Should report no heap-use-after-free or buffer overflows
```

Milestone 3 Checkpoint:

```bash
# Stress test with connection debugging
$ ./server --log-level=2 --max-connections=1000 &
$ stress_tool --connections=1000 --duration=10
# Check logs for connection state transitions and leaks

# Monitor file descriptors during test
$ watch -n1 'ls -l /proc/$(pidof server)/fd | wc -l'
# FD count should stabilize, not grow indefinitely
```

Milestone 4 Checkpoint:

```bash
# Test zero-copy buffer tracking
$ ./server --log-level=1 --zc-debug &
$ send_zero_copy_test --count=1000
# Check logs for zc_refcount values; should return to 0

# Profile with perf
$ perf record -g ./server --benchmark
$ perf report --no-children
# Look for time spent in io_uring_enter vs. processing
```

G. Debugging Tips Quick Reference

Symptom	Quick Diagnosis Command	Expected Output
High CPU usage	`perf top -p $(pidof server)`	Should show time in `io_uring_enter`, not tight loops
Memory leak	`valgrind --leak-check=full ./server --test`	"All heap blocks were freed"
Connection leak	`watch -n1 'ss -tn state connected dst :8080	wc -l'`
Ring full errors	`strace -e io_uring_enter ./server 2>&1	grep EBUSY`
Completion lag	Add `gettimeofday()` before submit and after completion	Latency should be consistent, not growing

Future Extensions

> **Milestone(s):** This section looks beyond the four core milestones to suggest potential enhancements that can be built upon the established io_uring foundation. These extensions represent natural progression points for further learning after mastering the fundamentals.

This document has provided a comprehensive roadmap for building a high-performance server using io_uring across four progressive milestones. Having mastered the core concepts—submission/completion queue mechanics, asynchronous file I/O, network server construction, and advanced features like zero-copy and linked operations—you now possess a solid foundation in modern Linux asynchronous I/O. The journey doesn't end here, however. The architectural patterns and implementation techniques you've learned serve as a springboard for exploring more sophisticated server designs and performance optimizations.

The extensions presented in this section represent natural progression points that build directly upon your existing codebase. Each extension addresses a specific limitation or expands the capabilities of your server in meaningful ways, offering opportunities to deepen your understanding of systems programming, concurrent architectures, and performance engineering. Consider these extensions as optional "bonus milestones" that transform your educational project into an increasingly sophisticated production-ready server.

Possible Enhancements

The following extensions are arranged roughly in order of complexity, starting with architectural changes (multi-threading) and progressing through protocol support, feature enhancements, and advanced optimizations. Each extension includes a mental model to build intuition, a detailed technical overview, specific integration points with your existing codebase, and challenges to anticipate.

1. Multi-Threaded io_uring with Work Stealing

Mental Model: The Restaurant Kitchen with Multiple Stations

> Imagine expanding your single-chef kitchen to a full kitchen brigade with multiple stations (threads), each with its own ticket rail (io_uring instance). A head chef (dispatcher) assigns incoming orders (connections) to specific stations, but stations can "steal" work from each other's backlog when they have spare capacity, ensuring no station is idle while another is overwhelmed. This model maintains the efficiency of specialized stations while enabling dynamic load balancing across the entire kitchen.

Technical Overview: Modern servers rarely operate with a single thread; they leverage multiple CPU cores to achieve true parallelism. While io_uring itself is thread-safe (multiple threads can submit and complete operations on the same ring), contention on the shared submission queue can become a bottleneck under extreme load. A more scalable approach involves creating multiple io_uring instances, each managed by a dedicated thread, and distributing connections across these instances.

The key challenge is maintaining **affinity**: once a connection's file descriptor is registered with a specific io_uring instance (via `IORING_REGISTER_FILES`), its I/O operations should generally be handled by that same instance to avoid cross-thread synchronization overhead. However, when one thread becomes overloaded while another is idle, a **work-stealing** mechanism can dynamically redistribute pending operations.

Integration with Existing Codebase:

- **Modify `struct network_server`**: Add fields for thread pool management:

```c
struct thread_worker {
    pthread_t thread_id;
    struct io_uring_engine *engine;
    struct connection_table *conn_table;
    int epoll_fd; // For event notification between threads
    volatile bool running;
    struct work_stealing_queue *local_queue;
};

struct network_server {
    // Existing fields...
    struct thread_worker *workers;
    int worker_count;
    pthread_mutex_t load_balancer_lock;
    struct connection_affinity_map *affinity_map;
};
```

- **Extend Connection Management**: Each connection is assigned a "home" worker thread upon acceptance. The `struct connection_state` gains a `worker_id` field indicating its assigned worker.
- **Implement Work Stealing**: When a worker's completion queue is empty, it can check other workers' pending operation counts and "steal" connections by migrating their file descriptor registration to its own io_uring instance (requires careful synchronization).

Challenges and Considerations:

- **File Descriptor Migration**: Moving a socket between io_uring instances requires unregistering from one ring and registering with another, creating a window where I/O operations cannot be submitted.
- **Synchronization Overhead**: Work stealing requires atomic operations or locks to safely inspect and transfer work items between threads, potentially negating performance gains if implemented naively.
- **Memory Locality**: Each thread should ideally operate on connections whose buffers are in NUMA-local memory for optimal cache performance.

Implementation Path:

1. Start with a simple thread-per-core model where each thread has its own independent io_uring instance and accepts connections via a shared listen socket (using `SO_REUSEPORT`).
2. Add connection affinity tracking to ensure subsequent I/O on a connection returns to the same thread.
3. Implement basic load monitoring and work stealing for heavily skewed workloads.
4. Benchmark with increasing connection counts to identify the optimal worker count for your hardware.

2. UDP Support with Connectionless I/O

Mental Model: The Postal Service vs. Telephone System

> While TCP connections are like telephone calls requiring establishment, conversation, and termination, UDP datagrams are like postal letters: each is independent, self-contained, and may arrive out of order or not at all. Supporting UDP transforms your server from a call center into a mail sorting facility, where each datagram is processed independently without connection state.

Technical Overview: Your current server implements a connection-oriented TCP model using `IORING_OP_ACCEPT`, `IORING_OP_READ`, and `IORING_OP_WRITE`. UDP introduces a connectionless paradigm where operations use `IORING_OP_RECVMSG` and `IORING_OP_SENDMSG` with destination addresses. The architectural shift is significant:

- **No connection state machine**: Each datagram stands alone; there's no `CONN_READING` or `CONN_WRITING` state.
- **Buffer management complexity**: Since datagrams have variable sizes and no flow control, buffers must handle maximum datagram size (typically 65KB) without wasting memory.
- **Multishot advantages**: `IORING_OP_RECVMSG` supports multishot mode even more effectively than TCP, allowing a single SQE to generate completions for multiple incoming datagrams.

Integration with Existing Codebase:

- **Extend `struct connection_state` or create new `struct udp_session`**: Since UDP is connectionless, you might track "sessions" based on source address rather than file descriptors:

```c
struct udp_session {
    struct sockaddr_in remote_addr;
    time_t last_active;
    uint64_t packets_received;
    uint64_t packets_sent;
    // Protocol-specific state (e.g., for QUIC or DNS)
    void *protocol_ctx;
};
```

- **Add UDP-specific operation handlers**: Create `submit_udp_recv()` and `submit_udp_send()` functions that use `IORING_OP_RECVMSG` and `IORING_OP_SENDMSG`.
- **Implement multishot UDP receive**: A single SQE with `IORING_OP_RECVMSG | IORING_RECV_MULTISHOT` can process thousands of incoming datagrams without re-submission.

Challenges and Considerations:

- **Message boundaries**: Unlike TCP's byte stream, UDP preserves message boundaries, requiring different application protocol handling.
- **No built-in congestion control**: Your server must implement rate limiting or use protocols like QUIC that add reliability atop UDP.
- **Security implications**: UDP amplification attacks are a concern; implement source address validation and rate limiting.

Implementation Path:

1. Create a parallel UDP server that shares the buffer pool and event loop infrastructure but uses different operation types.
2. Implement a simple UDP echo server to validate basic functionality.
3. Add a DNS server implementation (handling DNS over UDP) as a concrete use case.
4. Benchmark UDP throughput vs TCP for small, high-volume messages.

3. HTTP/1.1 Protocol Implementation

Mental Model: The Document Translation Service

> Imagine your current server as a raw pipe moving bytes between clients and files. Adding HTTP support transforms it into a document translation service: clients send formatted requests (HTTP messages), and your server must parse these requests, locate the corresponding resources, and format proper responses with headers, status codes, and bodies. The I/O engine remains the same, but you add a protocol "translation layer" atop it.

Technical Overview: Your server currently handles raw byte streams; clients and servers must agree on their own protocol. Implementing HTTP/1.1 provides a standardized, widely-used protocol that enables interoperability with web browsers, curl, and other HTTP clients. The implementation involves:

- **Request parsing**: Reading the incoming byte stream and separating it into method, URI, headers, and body.
- **Route handling**: Mapping URIs to file paths or handler functions.
- **Response generation**: Building properly formatted HTTP responses with status lines, headers, and content.
- **Connection management**: HTTP/1.1 supports persistent connections and pipelining, requiring careful tracking of request/response boundaries on a single connection.

Integration with Existing Codebase:

- **Extend `struct connection_state`**: Add HTTP-specific context:

```c
struct http_context {
    enum { HTTP_READING_HEADERS, HTTP_READING_BODY, HTTP_SENDING_RESPONSE } state;
    struct http_request req;
    struct http_response resp;
    size_t bytes_received;
    size_t bytes_sent;
    bool keep_alive;
};

struct connection_state {
    // Existing fields...
    struct http_context *http_ctx; // NULL for non-HTTP connections
};
```

- **Implement incremental parser**: Since reads may complete with partial HTTP messages, implement a stateful parser that can resume where it left off.
- **Add URI-to-file mapping**: For a static file server, translate URIs to filesystem paths with security checks to prevent directory traversal.

Challenges and Considerations:

- **Buffer management for large requests**: HTTP requests with large bodies (e.g., file uploads) may exceed your buffer pool size, requiring chunked reading.
- **Header parsing performance**: HTTP headers are text-based and require parsing; optimize with techniques like header field tokenization.

- **Security**: Implement protections against request smuggling, header injection, and path traversal attacks.

Implementation Path:

1. Start with a minimal HTTP/1.0 implementation supporting only `GET` requests for static files.
2. Add HTTP/1.1 support with `Keep-Alive`, chunked encoding, and proper connection management.
3. Implement common headers (`Content-Length`, `Content-Type` via MIME detection, `Last-Modified`).
4. Add support for `HEAD`, `POST`, and directory listings.
5. Benchmark with standard HTTP benchmarking tools (wrk, ab).

4. Write-Ahead Log (WAL) for Operation Durability

Mental Model: The Flight Recorder and Recovery Playback

> Imagine your server as an air traffic control system. A write-ahead log acts as the flight recorder: before executing any critical operation (like modifying a file), it first records the intention to disk. If the server crashes mid-operation, upon restart it can "replay the tape" to reconstruct the system state and ensure no operation is left half-completed. This transforms your server from stateless to stateful with crash consistency guarantees.

Technical Overview: Your current file server is essentially read-only or assumes idempotent writes. For operations that modify state (file writes, deletions, renames), ensuring durability across server crashes requires a write-ahead log. The WAL pattern:

1. **Log the intent**: Before performing the actual operation, write a record to a dedicated log file describing what will be done.
2. **Fsync the log**: Ensure the log record reaches stable storage.
3. **Execute the operation**: Perform the actual file modification.
4. **Log the completion**: Write a completion record to the log.
5. **Periodic checkpointing**: Occasionally compact the log by ensuring all completed operations are durable and trimming old records.

With io_uring, you can perform the log writes and fsync operations asynchronously, maintaining high throughput even with durability guarantees.

Integration with Existing Codebase:

- **Create WAL subsystem**:

```c
struct wal {
    int fd;
    off_t write_offset;
    off_t commit_offset;
    pthread_mutex_t lock;
    struct io_uring_engine *engine;
};

struct wal_record {
    uint64_t seq;
    uint8_t op_type; // FILE_WRITE, FILE_RENAME, etc.
    uint64_t timestamp;
    char path[256];
    union {
        struct { off_t offset; size_t len; } write;
        struct { char new_path[256]; } rename;
    };
    uint8_t checksum;
};
```

- **Modify file write operations**: Before submitting a write SQE, submit a WAL record SQE with `IORING_OP_WRITE` followed by `IORING_OP_FSYNC`.
- **Add recovery routine**: On server startup, scan the WAL for incomplete operations and either complete or roll them back.

Challenges and Considerations:

- **Log serialization**: WAL records must be written sequentially; concurrent operations must queue for log access.
- **Fsync performance**: `IORING_OP_FSYNC` forces actual disk writes, which can bottleneck throughput.
- **Log growth management**: Without checkpointing, the WAL grows indefinitely; implement log rotation and archiving.

Implementation Path:

1. Implement a simple append-only log with synchronous writes.
2. Convert to async io_uring operations for log writing and fsync.
3. Add recovery logic that replays the log on startup.
4. Implement checkpointing to truncate the log once operations are confirmed durable.

5. TLS/SSL Termination with Async Handshakes

Mental Model: The Secure Diplomatic Courier Channel

> Your current network server establishes plaintext connections like open postal mail. TLS adds an encryption layer akin to a diplomatic courier channel: before any substantive communication, both parties engage in a complex handshake to establish shared secrets and authenticate identities. The challenge is performing this computationally intensive handshake asynchronously without blocking the entire server.

Technical Overview: Adding TLS support (via OpenSSL or RustLS) enables secure HTTPS connections. The traditional approach uses blocking OpenSSL calls, which would stall your entire event loop. Modern solutions include:

- **Async OpenSSL with io_uring**: Using `SSL_read_ex` and `SSL_write_ex` with `SSL_ERROR_WANT_READ` / `SSL_ERROR_WANT_WRITE` return codes, resuming handshakes when I/O completes.
- **Linux Kernel TLS (kTLS)**: Offload TLS encryption/decryption to the kernel, allowing `IORING_OP_READ` and `IORING_OP_WRITE` to operate on encrypted data directly.
- **Custom async TLS implementations**: Libraries like `rustls` offer async-native TLS that integrates with io_uring.

The most promising approach for io_uring is kTLS, which allows maintaining your existing I/O patterns while the kernel handles encryption transparently.

Integration with Existing Codebase:

- **Add kTLS setup**:

```c
int setup_ktls(int sockfd, const char *cert_path, const char *key_path) {
    // Load certificate and key, perform TLS handshake in userspace
    // Configure kTLS on the socket using setsockopt with TLS_TX and TLS_RX
    // Return 0 on success
}
```

- **Minimal code changes**: Once kTLS is configured, `IORING_OP_READ` and `IORING_OP_WRITE` operate on encrypted data; the kernel handles encryption/decryption.
- **Fallback to userspace TLS**: For systems without kTLS support, implement async OpenSSL with state machine tracking handshake progress.

Challenges and Considerations:

- **kTLS availability**: Requires Linux kernel 4.13+ with specific hardware/configuration; may not be available in all environments.
- **Handshake performance**: TLS handshakes are CPU-intensive; consider offloading to a separate thread pool.
- **Certificate management**: Implement certificate loading, renewal, and SNI (Server Name Indication) support for multiple domains.

Implementation Path:

1. Implement a basic TLS-terminating proxy that accepts TLS connections and forwards plaintext to your existing server.
2. Integrate kTLS setup into your connection acceptance flow.
3. Add fallback userspace TLS for systems without kTLS.
4. Benchmark TLS overhead with and without kTLS.

6. Advanced Load Balancing and Service Discovery

Mental Model: The Airport Hub and Gate Assignment System

> Your single-server instance is like a small regional airport. Adding load balancing transforms it into a major hub: incoming requests are inspected and routed to one of many backend servers (gates) based on load, proximity, or capability. Service discovery acts as the flight information display, keeping track of which backend servers are operational and what services they offer.

Technical Overview: Transform your server from a standalone instance into a load balancer that distributes requests across multiple backend servers. This involves:

- **Health checking**: Periodically verifying backend servers are responsive.
- **Load metrics**: Tracking request latency, active connections, and error rates per backend.
- **Routing algorithms**: Round-robin, least connections, latency-based, or consistent hashing for session affinity.
- **Dynamic configuration**: Adding/removing backends without restarting.

With io_uring, you can implement the load balancer itself as a high-performance proxy that uses `IORING_OP_SPLICE` for zero-copy data transfer between client and backend connections.

Integration with Existing Codebase:

- **Create load balancer structure**:

```c
struct backend_server {
    char host[64];
    int port;
    int active_connections;
    double avg_latency_ms;
    time_t last_health_check;
    bool healthy;
};

struct load_balancer {
    struct backend_server *backends;
    int backend_count;
    enum { RR, LEAST_CONN, LATENCY } algorithm;
    int rr_index; // For round-robin
};
```

- **Implement proxy logic**: Accept client connections, select backend, establish backend connection, and splice data between the two sockets.
- **Add health check coroutine**: Periodically submit `IORING_OP_CONNECT` and `IORING_OP_WRITE` / `IORING_OP_READ` to backends to verify availability.

Challenges and Considerations:

- **Connection pooling**: Maintaining persistent connections to backends to reduce setup overhead.
- **Zero-copy splicing**: Using `IORING_OP_SPLICE` to transfer data between sockets without copying to userspace.
- **Stateful session affinity**: Ensuring requests from the same client reach the same backend when required.

Implementation Path:

1. Implement a simple TCP proxy that forwards connections to a single backend.
2. Add multiple backends with round-robin selection.
3. Implement health checks and automatic backend removal/addition.
4. Add zero-copy splicing for improved performance.
5. Benchmark latency and throughput compared to nginx or haproxy.

7. Metrics, Observability, and Adaptive Tuning

Mental Model: The Ship's Bridge with Instrument Panels and Autopilot

> Your server currently operates like a ship with basic controls but no instrumentation. Adding observability installs a comprehensive control panel showing speed (throughput), engine stress (CPU usage), cargo load (active connections), and sea conditions (system load). Adaptive tuning adds an autopilot that adjusts course (server parameters) based on these readings to maintain optimal performance in changing conditions.

Technical Overview: Production servers require detailed metrics for performance monitoring, debugging, and capacity planning. Implement:

- **Performance counters**: Request rate, latency distribution, I/O operation counts, buffer pool utilization.
- **Resource monitoring**: CPU, memory, file descriptor usage.
- **Export mechanisms**: Prometheus metrics endpoint, structured logs, or OpenTelemetry integration.
- **Adaptive tuning**: Automatically adjust parameters like `sq_entries`, `cq_entries`, or buffer pool size based on observed load.

Integration with Existing Codebase:

- **Extend `struct engine_stats`**:

```c
struct engine_stats {
    // Existing fields...
    uint64_t connection_count;
    uint64_t request_latency_sum_us;
    uint64_t request_latency_count;
    uint64_t io_errors;
    uint64_t buffer_pool_hit_rate;
};
```

- **Add metrics aggregation thread**: Periodically collect stats from all components and export them.
- **Implement adaptive tuning**:

```c
void adaptive_tune(struct io_uring_engine *engine) {
    if (engine->stats.cqe_overflow_count > 10) {
        // Increase CQ size
        // May require recreating io_uring instance
    }
    if (engine->stats.buffer_pool_hit_rate < 0.5) {
        // Increase buffer pool size
    }
}
```

Challenges and Considerations:

- **Metric collection overhead**: Ensure statistics gathering doesn't significantly impact performance.
- **Thread-safe counters**: Use atomic operations for frequently updated counters.
- **Parameter tuning complexity**: Some parameters (like ring size) require io_uring recreation, causing temporary service interruption.

Implementation Path:

1. Add basic counters to track key operations.
2. Implement a simple metrics HTTP endpoint that returns JSON.
3. Add Prometheus exposition format support.
4. Implement one adaptive tuning mechanism (e.g., dynamic buffer pool sizing).
5. Create dashboards with Grafana or similar to visualize metrics.

Implementation Guidance

While the extensions above are conceptual, here are practical starting points for implementing them in your existing C codebase.

Technology Recommendations Table

Extension	Simple Option	Advanced Option
Multi-threading	pthreads with one io_uring per thread	Work stealing with lock-free queues, NUMA awareness
UDP support	Basic `recvmsg` / `sendmsg` with single buffer	Multishot `IORING_OP_RECVMSG` with scatter/gather I/O
HTTP/1.1	Minimal parser for GET requests only	Full HTTP/1.1 with pipelining, chunked encoding, compression
Write-Ahead Log	Append-only file with synchronous writes	Async io_uring writes with `IORING_OP_FSYNC`, log compaction
TLS termination	Userspace OpenSSL with async handshakes	Kernel TLS (kTLS) with offloaded crypto
Load balancing	Simple round-robin TCP proxy	Zero-copy splicing with health checks and consistent hashing
Observability	Basic counters printed to logs	Prometheus metrics, adaptive tuning, Grafana dashboards

Recommended File/Module Structure

Extend your existing codebase with the following structure for the most generally useful extensions:

```
project-root/
├── src/
│   ├── core/                   # Existing core io_uring engine
│   │   ├── engine.c
│   │   ├── engine.h
│   │   └── event_loop.c
│   ├── network/                # Existing network server
│   │   ├── server.c
│   │   ├── connection.c
│   │   └── protocol/           # NEW: Protocol implementations
│   │       ├── http/
│   │       │   ├── parser.c    # HTTP request parser
│   │       │   ├── handler.c   # HTTP request handler
│   │       │   └── response.c  # HTTP response builder
│   │       └── tls/
│   │           ├── ktls.c      # Kernel TLS setup
│   │           └── openssl_async.c  # Async OpenSSL wrapper
│   ├── file/                   # Existing file I/O
│   │   ├── file_server.c
│   │   └── wal.c               # NEW: Write-ahead log
│   ├── load_balancer/          # NEW: Load balancing extension
│   │   ├── backend.c           # Backend server management
│   │   ├── proxy.c             # Zero-copy proxy logic
│   │   └── health_check.c      # Health monitoring
│   ├── threading/              # NEW: Multi-threading extensions
│   │   ├── worker_pool.c       # Thread worker management
│   │   ├── work_stealing.c     # Work stealing queues
│   │   └── affinity.c          # CPU/core affinity
│   ├── metrics/                # NEW: Observability
│   │   ├── counters.c          # Performance counters
│   │   ├── exporter.c          # Metrics export (Prometheus)
│   │   └── adaptive.c          # Adaptive tuning logic
│   └── utils/
│       ├── buffer_pool.c       # Existing buffer pool
│       └── hash_table.c        # Connection/session tables
└── benchmarks/                 # Benchmark suite
    ├── http_bench.c            # HTTP-specific benchmarks
    └── tls_bench.c             # TLS overhead benchmarks
```

Infrastructure Starter Code: HTTP Parser Foundation

For the HTTP extension, here's a minimal incremental parser foundation that integrates with your async read pattern:

```c
/* src/network/protocol/http/parser.h */

#ifndef HTTP_PARSER_H

#define HTTP_PARSER_H


#include <stddef.h>

#include <stdbool.h>


typedef enum {

    HTTP_METHOD_GET,

    HTTP_METHOD_POST,

    HTTP_METHOD_HEAD,

    HTTP_METHOD_UNKNOWN

} http_method_t;


typedef enum {

    PARSE_INCOMPLETE,    // Need more data

    PARSE_COMPLETE,      // Full request parsed

    PARSE_ERROR          // Malformed request

} parse_status_t;


typedef struct {

    http_method_t method;

    char uri[1024];

    char version[16];     // "HTTP/1.0" or "HTTP/1.1"

    size_t content_length;

    bool keep_alive;

    // Simplified headers storage

    char headers_raw[4096];

    size_t headers_len;

} http_request_t;


typedef struct {

    http_request_t req;

    size_t bytes_parsed;

    size_t content_received;

    enum {

        STATE_START_LINE,

        STATE_HEADERS,

        STATE_BODY,
```

```
        STATE_COMPLETE
    } state;
} http_parser_t;

void http_parser_init(http_parser_t *parser);
parse_status_t http_parser_feed(http_parser_t *parser, const char *data, size_t len);
void http_parser_reset(http_parser_t *parser);

#endif /* HTTP_PARSER_H */
```

```c
/* src/network/protocol/http/parser.c */

#include "parser.h"

#include <string.h>

#include <ctype.h>

#include <stdio.h>


void http_parser_init(http_parser_t *parser) {

    memset(parser, 0, sizeof(*parser));

    parser->state = STATE_START_LINE;

}


static parse_status_t parse_start_line(http_parser_t *parser, const char *data, size_t len) {

    // Find end of line

    const char *end = memchr(data, '\n', len);

    if (!end) return PARSE_INCOMPLETE;



    // Parse method

    if (strncmp(data, "GET ", 4) == 0) {

        parser->req.method = HTTP_METHOD_GET;

        data += 4;

    } else if (strncmp(data, "POST ", 5) == 0) {

        parser->req.method = HTTP_METHOD_POST;

        data += 5;

    } else if (strncmp(data, "HEAD ", 5) == 0) {

        parser->req.method = HTTP_METHOD_HEAD;

        data += 5;

    } else {

        parser->req.method = HTTP_METHOD_UNKNOWN;

        return PARSE_ERROR;

    }



    // TODO: Parse URI and HTTP version

    // Simplified: just advance parser position

    size_t line_len = end - data + 1;

    parser->bytes_parsed += line_len;

    parser->state = STATE_HEADERS;



    return PARSE_INCOMPLETE; // Need headers
```

```c
}

parse_status_t http_parser_feed(http_parser_t *parser, const char *data, size_t len) {
    size_t remaining = len;
    const char *pos = data;

    while (remaining > 0) {
        switch (parser->state) {
            case STATE_START_LINE:
                parse_status_t status = parse_start_line(parser, pos, remaining);
                if (status == PARSE_ERROR) return PARSE_ERROR;
                if (status == PARSE_INCOMPLETE) return PARSE_INCOMPLETE;
                // Advance position based on bytes consumed
                size_t consumed = parser->bytes_parsed;
                pos += consumed;
                remaining -= consumed;
                break;

            case STATE_HEADERS:
                // TODO: Parse headers, look for Content-Length, Connection, etc.
                // When blank line found, transition to STATE_BODY or STATE_COMPLETE
                break;

            case STATE_BODY:
                // TODO: Accumulate body based on Content-Length or chunked encoding
                break;

            case STATE_COMPLETE:
                return PARSE_COMPLETE;
        }
    }

    return PARSE_INCOMPLETE;
}

void http_parser_reset(http_parser_t *parser) {
    http_parser_init(parser);
```

```
}
```

Core Logic Skeleton: Multi-threaded Worker Pool

For the multi-threading extension, here's a skeleton for the worker pool manager:

```c
/* src/threading/worker_pool.c */

#include <pthread.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/sysinfo.h>


#include "worker_pool.h"

#include "../core/engine.h"

#include "../network/connection.h"


struct thread_worker {

    pthread_t thread_id;

    int worker_index;

    struct io_uring_engine *engine;

    struct connection_table *conn_table;

    int event_fd;  // For cross-thread notifications

    volatile bool running;

    struct work_queue *pending_work;

    pthread_mutex_t queue_lock;

};


struct worker_pool {

    struct thread_worker *workers;

    int worker_count;

    int next_worker;  // For round-robin assignment

    pthread_mutex_t assignment_lock;

};


// TODO 1: Implement worker_pool_create that creates N workers (default = CPU count)

//         Each worker gets its own io_uring_engine and connection_table

struct worker_pool *worker_pool_create(int worker_count) {

    if (worker_count <= 0) {

        worker_count = get_nprocs();

    }



    struct worker_pool *pool = calloc(1, sizeof(struct worker_pool));

    pool->workers = calloc(worker_count, sizeof(struct thread_worker));

    pool->worker_count = worker_count;
```

```
    // TODO 2: Initialize each worker with engine, connection table, and event_fd
    for (int i = 0; i < worker_count; i++) {
        struct thread_worker *worker = &pool->workers[i];
        worker->worker_index = i;
        pthread_mutex_init(&worker->queue_lock, NULL);
        worker->event_fd = eventfd(0, EFD_NONBLOCK);

        // Create per-worker engine with appropriate configuration
        struct engine_config cfg = {
            .sq_entries = 4096,
            .cq_entries = 8192,
            .flags = 0,
            .enable_sqpoll = false,
            .sqpoll_cpu = 0
        };
        worker->engine = engine_create(&cfg);

        worker->conn_table = conn_table_create(1024);
        worker->pending_work = work_queue_create();
    }

    return pool;
}

// TODO 3: Implement worker_thread_main function that runs each worker's event loop
static void *worker_thread_main(void *arg) {
    struct thread_worker *worker = (struct thread_worker *)arg;

    while (worker->running) {
        // TODO 4: Check for incoming work from other threads via event_fd
        //         Use io_uring to read from event_fd asynchronously

        // TODO 5: Process any pending work in the worker's queue
        struct work_item *item = work_queue_pop(worker);
        if (item) {
            // TODO 6: Handle different work item types (new connection, migrated connection, etc.)
            handle_work_item(worker, item);
```

```
        }

        // TODO 7: Run the worker's own event loop for its connections
        engine_process_completions(worker->engine);

        // TODO 8: Implement work stealing: if idle, check other workers' queues
        if (worker_idle(worker)) {
            try_steal_work(worker, worker->worker_index);
        }
    }

    return NULL;
}

// TODO 9: Implement assign_connection_to_worker that uses a load balancing strategy
int assign_connection_to_worker(struct worker_pool *pool, int client_fd,
                                struct sockaddr_in *addr) {
    pthread_mutex_lock(&pool->assignment_lock);

    // Simple round-robin assignment
    int worker_idx = pool->next_worker;
    pool->next_worker = (pool->next_worker + 1) % pool->worker_count;

    pthread_mutex_unlock(&pool->assignment_lock);

    // TODO 10: Create connection state and add to worker's connection table
    struct connection_state *conn = conn_create(client_fd, addr);
    conn->worker_id = worker_idx;  // Track affinity

    struct thread_worker *worker = &pool->workers[worker_idx];
    pthread_mutex_lock(&worker->queue_lock);
    conn_register(worker->conn_table, conn);
    pthread_mutex_unlock(&worker->queue_lock);

    // TODO 11: Notify worker about new connection via event_fd
    uint64_t notification = 1;
    write(worker->event_fd, &notification, sizeof(notification));
```

```
    return worker_idx;
}

// TODO 12: Implement work stealing logic when a worker is idle
static void try_steal_work(struct thread_worker *thief, int thief_idx) {
    // Check neighbors first (cache-friendly)
    for (int offset = 1; offset < MAX_WORKERS; offset++) {
        int victim_idx = (thief_idx + offset) % MAX_WORKERS;
        if (victim_idx == thief_idx) continue;

        // TODO 13: Check if victim has pending work
        // TODO 14: Safely migrate a connection from victim to thief
        // TODO 15: Update connection's worker_id and re-register with thief's io_uring
    }
}
```

Language-Specific Hints: C

- **Threading**: Use `pthread_create` with thread affinity (`pthread_setaffinity_np`) to pin workers to specific CPU cores for better cache locality.
- **Atomic Operations**: For performance counters, use GCC builtins like `__sync_fetch_and_add` or C11 `_Atomic` types for lock-free statistics.
- **Memory Barriers**: When implementing work stealing, use `__atomic_thread_fence` to ensure proper memory ordering between threads.
- **File Descriptor Passing**: For connection migration between threads, consider using `SCM_RIGHTS` with `sendmsg` / `recvmsg` to pass file descriptors through Unix domain sockets.
- **TLS Storage**: Use `pthread_key_create` and `pthread_setspecific` for thread-local storage of worker context.

Milestone Checkpoint: HTTP Extension

After implementing basic HTTP support, verify functionality with:

```bash
# Start your HTTP server on port 8080
./build/server --http --port 8080 --root ./www

# Test with curl
curl -v http://localhost:8080/index.html

# Expected output includes proper HTTP headers:
# HTTP/1.1 200 OK
# Content-Type: text/html
# Content-Length: 1234
# Connection: keep-alive
#
# <file contents>

# Benchmark with wrk
wrk -t4 -c100 -d10s http://localhost:8080/index.html

# Compare performance against your raw TCP server
```

Debugging Tips for Extensions

Symptom	Likely Cause	How to Diagnose	Fix
HTTP parser stuck in incomplete state	Buffer doesn't contain complete HTTP request line or headers	Add debug logging showing buffer contents and parser state	Ensure you're accumulating data across multiple reads until request is complete
Multi-threaded server has lower throughput than single-threaded	Lock contention or excessive work stealing overhead	Profile with `perf record -g` and look for `pthread_mutex_lock` hotspots	Reduce lock granularity, use lock-free data structures, or disable work stealing
TLS handshake times out	Blocking OpenSSL calls stalling event loop	Use `strace` to see if thread is stuck in `read`/`write` syscalls	Implement proper async OpenSSL with `SSL_get_error` and `SSL_ERROR_WANT_READ`
Load balancer crashes when backend dies	No error handling for failed backend connections	Add logging when `IORING_OP_CONNECT` or health check fails	Implement circuit breaker pattern, remove unhealthy backends from rotation
Memory leak with WAL extension	WAL records not freed after checkpoint	Use `valgrind --leak-check=full` to identify allocation sites	Ensure WAL records are properly reference counted and freed after durability confirmed
UDP server loses packets under load	Receive buffer overflow or application too slow	Check `/proc/net/udp` for `drops` count, increase socket buffer size	Use `IORING_OP_RECVMSG_MULTISHOT` for higher efficiency, or add receive buffer pooling

Remember that these extensions represent advanced topics; tackle them one at a time after solidifying your understanding of the core io_uring concepts from the main milestones. Each extension not only enhances your server's capabilities but deepens your expertise in system architecture and performance optimization.

Glossary

> **Milestone(s):** All milestones (1-4)

This glossary defines key terms, acronyms, and Linux-specific concepts used throughout this design document. Understanding these terms is essential for working effectively with `io_uring` and the architectural patterns described in this project. Terms are organized alphabetically for reference.

Terms and Definitions

The following table provides comprehensive definitions of technical terms, data structures, functions, and concepts referenced in this design.

Term	Definition	Related Concepts
`adaptive_tune(engine)`	Function that adjusts server parameters (like batch sizes or buffer pool sizes) based on runtime metrics and observed performance.	`engine_stats`, `loop_config`, performance optimization
`allocate_op_context(table)`	Function that allocates and registers a new `io_op_context` structure in the context table, assigning it a unique identifier.	`create_context_table`, `free_op_context`, `lookup_op_context`
Application error	An error category (ERR_CATEGORY_APPLICATION) indicating a problem caused by client behavior or invalid request parameters (e.g., file not found, malformed request). These errors are typically returned to the client.	`ERR_CATEGORY_TRANSIENT`, `ERR_CATEGORY_FATAL`, `handle_io_error`
Assembly Line with a Concierge Desk	A mental model for the network echo server with linked operations: the Concierge Desk (multishot accept) continuously hands out tickets, and the Assembly Line (linked SQE chain) processes each request through sequential stations (read → process → write).	Linked SQEs, multishot accept, network server
`backend_server`	A structure representing a backend server in a load-balancing configuration, containing host, port, health status, and performance metrics.	`load_balancer`, health checking, `LEAST_CONN`
`benchmark_file_server()`	Function that performs a throughput and latency benchmark of the file server under specified concurrency and request volume, storing results in a `benchmark_result` structure.	`benchmark_result`, performance testing, Milestone 4
`benchmark_result`	A structure containing the results of a performance benchmark, including server type, concurrency level, requests per second, throughput in MB/s, and CPU utilization.	performance metrics, Milestone 4
`buffer_pool`	A structure managing a collection of reusable `io_buffer` objects, with support for both dynamically allocated and fixed (pre-registered) buffers.	`io_buffer`, `create_buffer_pool`, `acquire_buffer`, `release_buffer`
Buffer pool load factor	The ratio of buffers currently in use to the total number of buffers in the pool. Used to determine when to expand the pool or throttle submissions.	`buffer_pool`, resource management
Buffer rotation	A technique of alternating between two or more buffers for a single connection to prevent read/write	`connection_state`, double buffering, pipeline parallelism

Term	Definition	Related Concepts
	conflicts, allowing data processing on one buffer while the next I/O operation uses another.	
Busy-wait	A polling strategy where the application actively checks for completions in a tight loop without blocking, trading higher CPU usage for lower latency. Configured via `loop_config.busy_wait`.	`loop_config`, `IORING_ENTER_GETEVENTS`, latency vs. throughput
`classify_error(error_code)`	Function that categorizes a system error code (like `EAGAIN`, `ECONNRESET`) into one of the defined error categories (application, transient, fatal, resource).	`handle_io_error`, error recovery strategy
Chain atomicity	The property of linked SQE chains where the entire chain succeeds or fails as a unit; if any operation in the chain fails, subsequent linked operations are not executed.	`IOSQE_IO_LINK`, linked operations, error handling
`compare_with_sync_baseline()`	Function that compares the performance of the `io_uring`-based file server against a synchronous I/O baseline, typically using tools like `dd` or a simple synchronous server.	`benchmark_file_server`, Milestone 2
Completion ordering	The sequence in which completed operations become available in the CQ. `io_uring` does **not** guarantee CQEs will appear in the same order as SQEs were submitted, except when using `IOSQE_IO_LINK` or `IOSQE_IO_DRAIN`.	`io_uring_cqe`, out-of-order completion, `IOSQE_IO_DRAIN`
`conn_create(fd, addr)`	Function that creates a new `connection_state` structure for a newly accepted socket, initializing its state to `CONN_ACCEPTING`.	`conn_destroy`, `conn_register`, `connection_state`
`conn_destroy(table, conn)`	Function that closes the socket, releases associated buffers, and frees a `connection_state` structure, removing it from the connection table.	`conn_create`, connection lifecycle, resource cleanup
`conn_lookup(table, user_data)`	Function that looks up a connection by decoding the `user_data` token and performing a hash table lookup in the connection table.	`conn_register`, `conn_unregister`, user_data correlation
`conn_register(table, conn)`	Function that inserts a `connection_state` into the connection table, assigning it a unique identifier that can be encoded into `user_data`.	`conn_table_create`, `conn_unregister`, hash table

Term	Definition	Related Concepts
`conn_table_create(size)`	Function that creates a connection table (a hash table) with a power-of-two capacity for efficient lookup of connections by their `user_data` token.	`conn_table_destroy`, `connection_table`, linear probing
`conn_unregister(table, conn)`	Function that removes a connection from the connection table, typically called during connection cleanup.	`conn_register`, `conn_destroy`
`CONN_ACCEPTING`	A connection state constant indicating the connection has been accepted but is waiting for the kernel to complete the accept operation (when using `IORING_OP_ACCEPT`).	`CONN_READING`, `CONN_WRITING`, `CONN_CLOSING`, `CONN_CLOSED`
`CONN_CLOSED`	A connection state constant indicating the connection is fully closed and its resources have been freed.	`CONN_CLOSING`, `conn_destroy`
`CONN_CLOSING`	A connection state constant indicating the connection is in the process of being cleaned up (canceling in-flight operations, closing socket).	`CONN_WRITING`, `CONN_CLOSED`, `IORING_OP_ASYNC_CANCEL`
`CONN_READING`	A connection state constant indicating the connection is waiting for data to be read from the socket.	`CONN_ACCEPTING`, `CONN_WRITING`, `submit_connection_read`
`CONN_WRITING`	A connection state constant indicating the connection is waiting for data to be written to the socket.	`CONN_READING`, `CONN_CLOSING`, `submit_connection_write`
Connection affinity	A load-balancing strategy where related connections (e.g., from the same client IP) are routed to the same worker thread to improve cache locality and simplify state management.	`worker_pool`, `assign_connection_to_worker`, thread-local storage
`connection_state`	The central structure representing a TCP connection, tracking its file descriptor, state machine, buffers, and operation context.	`conn_table_create`, `CONN_ACCEPTING`, `CONN_READING`, `CONN_WRITING`, `CONN_CLOSING`, `CONN_CLOSED`
Connection state machine	A finite state machine model that tracks a TCP connection through defined states: `ACCEPTING`, `READING`, `WRITING`, `CLOSING`, `CLOSED`. Transitions occur upon I/O completion events.	`connection_state`, state machines, network server
`connection_table`	A structure implementing a hash table for efficient lookup of `connection_state` objects by a `user_data` token. Uses linear probing for collision resolution.	`conn_table_create`, `conn_lookup`, hash collision resolution

Term	Definition	Related Concepts
`context_table`	A structure implementing a lookup table for `io_op_context` objects, keyed by the `user_data` token from CQEs. Typically implemented as an array with linear probing.	`create_context_table`, `allocate_op_context`, `lookup_op_context`
Correlation tokens	The `user_data` values embedded in SQEs that are returned in corresponding CQEs, allowing the application to match completions to their originating operation context.	`user_data`, `lookup_op_context`, `decode_user_data`
`create_buffer_pool(buffer_size, buffer_count, register_fixed, ring)`	Function that initializes a pool of buffers, optionally registering them as fixed buffers with the `io_uring` instance to reduce per-operation overhead.	`acquire_buffer`, `release_buffer`, `IORING_REGISTER_BUFFERS`
`create_context_table(size)`	Function that creates and initializes a context table with a specified initial size (rounded to power of two).	`destroy_context_table`, `allocate_op_context`, `context_table`
CQ	Completion Queue – the ring buffer where the kernel places `io_uring_cqe` structures to report the results of completed I/O operations. The application consumes entries from this queue.	`io_uring_cqe`, `io_uring_peek_batch_cqe`, `io_uring_cqe_seen`
CQE	Completion Queue Entry – a `struct io_uring_cqe` that reports the result (`res` field) of a completed I/O operation and includes the `user_data` token from the corresponding SQE.	`io_uring_cqe`, `user_data`, `io_uring_wait_cqe`
CQ overflow	A condition where the completion queue becomes full and the kernel cannot add new CQEs. When this occurs, the kernel sets the `IORING_CQ_F_OVERFLOW` flag and may drop completions, requiring special handling.	`IORING_CQ_F_OVERFLOW`, ring sizing, error recovery
`decode_user_data(user_data, conn_id, op_type, seq, buffer_id)`	Function that decodes the packed `user_data` token into its constituent parts: connection ID, operation type, sequence number, and buffer ID. Used for debugging and context lookup.	`encode_user_data`, `user_data` correlation, debugging
`dump_ring_state(ring)`	Debugging function that prints the current state of the io_uring rings (SQ and CQ head/tail pointers, entries) to help diagnose hangs or throughput issues.	debugging, `event_loop_run_debug`
`EAGAIN`	A Linux error code indicating the operation would block (non-blocking I/O) or that resources are temporarily unavailable. In	transient error, `ERR_CATEGORY_TRANSIENT`, retry logic

Term	Definition	Related Concepts
	`io_uring`, this may occur for socket operations when the socket buffer is full/empty.	
`EBUSY`	A Linux error code returned by `io_uring_enter` when the submission queue is full and the operation cannot proceed without waiting. Indicates the application should process some completions to free SQEs.	`io_uring_enter`, ring sizing, submission strategy
`ECONNRESET`	A Linux error code indicating the TCP connection was reset by the peer. This is a common network error that should result in connection cleanup.	`handle_network_error`, `CONN_CLOSING`, network errors
`EFAULT`	A Linux error code indicating an invalid memory address was accessed. In `io_uring`, this can occur if a buffer passed in an SQE is not accessible or has been freed before I/O completion.	memory management, fixed buffers, buffer lifecycle
`encode_user_data(conn_id, op_type, seq, buffer_id)`	Function that packs connection ID, operation type, sequence number, and buffer ID into a single 64-bit `user_data` token for embedding in an SQE.	`decode_user_data`, `user_data` correlation
`engine_create(cfg)`	Function that initializes an `io_uring_engine` instance, setting up the `io_uring` rings and associated data structures based on the provided `engine_config`.	`engine_destroy`, `io_uring_engine`, `engine_config`
`engine_destroy(eng)`	Function that tears down an `io_uring_engine` instance, unmapping rings, closing the io_uring file descriptor, and freeing allocated resources.	`engine_create`, cleanup, resource management
`engine_get_ring(eng)`	Function that returns a pointer to the underlying `struct io_uring` instance, allowing direct access to ring structures for advanced operations.	`io_uring_engine`, `io_uring`
`engine_get_sqe(eng)`	Function that returns a pointer to a free submission queue entry (SQE) from the engine's io_uring instance, handling internal batch management.	`io_uring_get_sqe`, `engine_submit`, SQE preparation
`engine_get_stats(eng, out)`	Function that retrieves current statistics (submissions, completions, syscall counts) from the engine and writes them to an `engine_stats` structure.	`engine_stats`, performance monitoring

Term	Definition	Related Concepts
`engine_process_completions(eng)`	Function that processes all available completion queue entries (CQEs) from the io_uring instance, invoking appropriate handlers for each operation type.	`io_uring_peek_batch_cqe`, `io_uring_cqe_seen`, event loop
`engine_process_completions_with_zc(eng)`	Extended version of `engine_process_completions` that also handles zero-copy notification CQEs (`IORING_CQE_F_NOTIF`) for buffer release.	`IORING_OP_SEND_ZC`, `IORING_CQE_F_NOTIF`, zero-copy
`engine_stats`	A structure tracking runtime statistics of the io_uring engine, including counts of SQEs submitted, CQEs processed, `io_uring_enter` calls, and batch efficiency.	`engine_get_stats`, performance monitoring
`engine_submit(eng)`	Function that submits all prepared SQEs in the submission queue to the kernel via `io_uring_enter`, without waiting for completions.	`io_uring_submit`, `io_uring_enter`, batched syscalls
`engine_submit_and_wait(eng, min_complete)`	Function that submits SQEs and waits for at least `min_complete` completions to be available, using the `IORING_ENTER_GETEVENTS` flag.	`io_uring_enter`, `IORING_ENTER_GETEVENTS`, blocking operation
`engine_wait_and_process(eng)`	Function that blocks waiting for completions and then processes all available CQEs. Combines waiting and processing in one call.	`engine_submit_and_wait`, `engine_process_completions`
`engine_config`	A structure containing configuration parameters for initializing an `io_uring_engine`, such as SQ/CQ sizes, flags like `IORING_SETUP_SQPOLL`, and CPU affinity for kernel polling.	`engine_create`, `IORING_SETUP_SQPOLL`, ring sizing
`ERR_CATEGORY_APPLICATION`	Error category constant (0) for errors caused by client input or application logic (e.g., file not found, invalid request). Recovery typically involves sending an error response to the client.	`classify_error`, `handle_io_error`, error handling
`ERR_CATEGORY_FATAL`	Error category constant (2) for critical errors indicating the system cannot continue normal operation (e.g., memory corruption, ring setup failure). Recovery typically involves graceful shutdown.	`classify_error`, `handle_io_error`, system integrity
`ERR_CATEGORY_RESOURCE`	Error category constant (3) for errors caused by resource exhaustion (e.g., `ENOBUFS`, `ENOMEM`). Recovery may involve throttling,	`classify_error`, `handle_io_error`, resource management

Term	Definition	Related Concepts
	expanding pools, or failing gracefully.	
`ERR_CATEGORY_TRANSIENT`	Error category constant (1) for temporary errors that may resolve if retried (e.g., `EAGAIN`, `EINTR`). The system may retry the operation after a delay.	`classify_error`, `handle_io_error`, retry logic
Error classification	The process of categorizing system error codes into defined categories (application, transient, fatal, resource) to determine appropriate recovery strategies.	`classify_error`, error handling table
`event_callbacks`	A structure containing function pointers for event loop callbacks: `on_setup`, `on_pre_submit`, `should_stop`, `on_cleanup`. Allows customization of the event loop behavior.	`event_loop_create`, event-driven architecture
Event-driven architecture	An architectural pattern where the flow of the program is determined by events such as I/O completions, timer expirations, or signals, as opposed to a linear control flow.	`event_loop`, `io_uring`, asynchronous I/O
`event_loop`	The central control structure that orchestrates the asynchronous I/O operations: preparing SQEs, submitting them, waiting for completions, and dispatching handlers.	`event_loop_run`, `event_loop_create`, `engine_process_completions`
`event_loop_create(eng, config, cbs, user_data)`	Function that creates an event loop instance, associating it with an io_uring engine, configuration, callbacks, and optional user data.	`event_loop_destroy`, `event_loop_run`
`event_loop_destroy(loop)`	Function that destroys an event loop instance, freeing associated resources and ensuring pending operations are canceled or completed.	`event_loop_create`, cleanup
`event_loop_request_stop(loop)`	Function that signals the event loop to stop after processing current in-flight operations, setting the `stop_requested` flag.	`event_loop_run`, graceful shutdown
`event_loop_run(loop)`	The main function that executes the event loop: repeatedly prepares operations, submits SQEs, waits for completions, and processes CQEs until `stop_requested` is true.	`event_loop`, `engine_process_completions`, main loop
`event_loop_run_debug(loop, dump_interval)`	A debugging version of `event_loop_run` that periodically dumps ring state and statistics at the specified interval (in iterations).	`dump_ring_state`, debugging

Term	Definition	Related Concepts
Fatal error	An error category (ERR_CATEGORY_FATAL) indicating a critical, unrecoverable error that compromises system integrity (e.g., ring corruption, memory allocation failure). Requires termination.	`ERR_CATEGORY_APPLICATION`, `ERR_CATEGORY_TRANSIENT`, graceful shutdown
Fixed buffer registration	The process of pre-registering a set of buffers with the kernel using `io_uring_register(fd, IORING_REGISTER_BUFFERS, ...)`. This allows SQEs to reference buffers by index instead of address, reducing per-operation overhead.	`io_uring_register`, `IORING_REGISTER_BUFFERS`, `buffer_pool`
Fixed buffers	Buffers that have been pre-registered with an io_uring instance, allowing them to be referenced in SQEs by a numeric index rather than a virtual address. This eliminates kernel mapping overhead for each I/O operation.	`io_uring_register`, `IORING_REGISTER_BUFFERS`, zero-copy
`handle_io_error(cqe, ctx, server_context)`	The main error dispatch function that calls appropriate subsystem-specific error handlers (e.g., `handle_network_error`) based on the operation type and error classification.	`classify_error`, `handle_network_error`, error recovery
`handle_linked_chain_completion(server, accept_cqe, read_cqe, write_cqe, conn)`	Function that processes the completion of a three-operation linked chain (accept→read→write), handling success or failure of the entire chain.	`submit_linked_accept_read_write_chain`, linked chain atomicity
`handle_network_error(cqe, ctx, server_context)`	Network-specific error handler invoked by `handle_io_error` for network-related operations (accept, read, write). Handles connection reset, timeout, and other socket errors.	`handle_io_error`, `CONN_CLOSING`, network server
Hash collision resolution	The method used to handle multiple keys mapping to the same hash table slot. In this project, linear probing is used: checking subsequent slots until an empty one is found.	`context_table`, `connection_table`, linear probing
`http_context`	A structure holding the state of an HTTP request/response transaction, including parser state, request/response structures, and byte counts.	`http_parser_t`, `http_request_t`, HTTP server
`HTTP_METHOD_GET`	Constant representing the HTTP GET method. Used in HTTP request parsing and routing.	`http_request_t`, HTTP server

Term	Definition	Related Concepts
`HTTP_METHOD_HEAD`	Constant representing the HTTP HEAD method.	`http_request_t`, HTTP server
`HTTP_METHOD_POST`	Constant representing the HTTP POST method.	`http_request_t`, HTTP server
`HTTP_METHOD_UNKNOWN`	Constant representing an unknown or unsupported HTTP method.	`http_request_t`, HTTP parsing
`HTTP_READING_BODY`	Constant representing the HTTP context state where the server is reading the request body (after headers).	`http_context`, `STATE_BODY`
`HTTP_READING_HEADERS`	Constant representing the HTTP context state where the server is reading the request headers.	`http_context`, `STATE_HEADERS`
`HTTP_SENDING_RESPONSE`	Constant representing the HTTP context state where the server is sending the response.	`http_context`, `STATE_COMPLETE`
`http_parser_feed(parser, data, len)`	Function that incrementally parses HTTP request data, updating the parser state and returning `PARSE_COMPLETE`, `PARSE_INCOMPLETE`, or `PARSE_ERROR`.	`http_parser_init`, `http_parser_reset`, incremental parser
`http_parser_init(parser)`	Function that initializes an HTTP parser state structure for a new request.	`http_parser_feed`, `http_parser_reset`
`http_parser_reset(parser)`	Function that resets an HTTP parser to its initial state, allowing reuse for a new request on the same connection.	`http_parser_init`, keep-alive connections
`http_parser_t`	A structure representing the state of an incremental HTTP parser, including the request being built and the current parsing state.	`http_request_t`, `PARSE_INCOMPLETE`, `PARSE_COMPLETE`
`http_request_t`	A structure representing a parsed HTTP request, containing method, URI, version, headers, and content length.	`http_parser_t`, HTTP server
Incremental parser	A parser that can process partial input and resume when more data arrives, as opposed to requiring the entire message at once. Essential for async I/O where data arrives in chunks.	`http_parser_feed`, `PARSE_INCOMPLETE`, network server
`io_uring`	The Linux kernel interface for asynchronous I/O, centered around two shared-memory ring buffers: the Submission Queue (SQ) and Completion Queue (CQ).	`io_uring_setup`, `io_uring_enter`, `io_uring_register`
`io_uring_cqe`	Completion Queue Entry structure: contains `user_data` (token from	CQE, `io_uring_wait_cqe`, `io_uring_cqe_seen`

Term	Definition	Related Concepts
	SQE) and `res` (result/error code of the operation). May include flags like `IORING_CQE_F_NOTIF`.	
`io_uring_cqe_seen(ring, cqe)`	Helper function that marks a completion queue entry as consumed, advancing the CQ tail pointer. Must be called after processing each CQE to prevent re-processing.	`io_uring_wait_cqe`, `io_uring_peek_batch_cqe`, CQ tail management
`io_uring_enter(fd, to_submit, min_complete, flags, sigmask)`	The system call used to submit SQEs to the kernel (`to_submit`) and/or wait for completions (`min_complete` with `IORING_ENTER_GETEVENTS`).	`io_uring_submit`, `io_uring_wait_cqe`, batched syscalls
`io_uring_get_sqe(ring)`	Helper function that returns a pointer to the next available submission queue entry in the ring. Returns NULL if the SQ is full.	`io_uring_submit`, SQE preparation
`io_uring_peek_batch_cqe(ring, cqes, count)`	Helper function that peeks at up to `count` completion queue entries without consuming them, allowing batch processing before calling `io_uring_cqe_seen`.	`io_uring_wait_cqe`, `io_uring_cqe_seen`, batch processing
`io_uring_register(fd, opcode, arg, nr_args)`	System call used to register resources (like buffers or file descriptors) with an io_uring instance, reducing per-operation overhead. Common opcodes: `IORING_REGISTER_BUFFERS`, `IORING_REGISTER_FILES`.	fixed buffers, `io_uring_setup`, resource registration
`io_uring_setup(entries, params)`	System call that creates a new io_uring instance, returning a file descriptor. The `params` structure specifies ring sizes and features like `IORING_SETUP_SQPOLL`.	`io_uring_enter`, `io_uring_register`, ring initialization
`io_uring_sqe`	Submission Queue Entry structure: describes a single I/O operation with fields like `opcode`, `fd`, `addr`, `len`, `offset`, and `user_data`.	SQE, `io_uring_get_sqe`, `io_uring_submit`
`io_uring_submit(ring)`	Helper function that submits all prepared SQEs in the submission queue to the kernel (calls `io_uring_enter` with the number of SQEs to submit).	`io_uring_get_sqe`, `io_uring_enter`, batched submission
`io_uring_wait_cqe(ring, cqe)`	Helper function that waits for at least one completion to be available, returning a pointer to a CQE. Blocks if no completions are ready.	`io_uring_peek_batch_cqe`, `io_uring_cqe_seen`, blocking wait
`io_uring_ctx`	An internal context structure that wraps the `io_uring` instance	`io_uring_engine`, `engine_config`, `engine_stats`

Term	Definition	Related Concepts
	along with configuration, statistics, and registered resources like fixed buffers.	
`io_uring_engine`	The primary abstraction for the io_uring subsystem, providing a simplified API for SQE submission, completion processing, and statistics collection.	`engine_create`, `engine_submit`, `engine_process_completions`
`io_buffer`	A structure representing a data buffer that can be used for I/O operations. Tracks memory pointer, capacity, usage status, fixed buffer index, and zero-copy reference count.	`buffer_pool`, `acquire_buffer`, `release_buffer`
`io_op_context`	A structure that holds application-specific context for an in-flight I/O operation, referenced via the `user_data` token. Contains operation type, associated connection, buffer, and chain pointers.	`context_table`, `allocate_op_context`, `lookup_op_context`
`IORING_ACCEPT_MULTISHOT`	A flag used with `IORING_OP_ACCEPT` to indicate that the accept operation should automatically re-submit itself after each successful connection acceptance, generating multiple CQEs from a single SQE.	multishot, `submit_accept_multishot`, network server
`IORING_CQ_F_OVERFLOW`	A flag set in the completion queue ring when the CQ has overflowed (kernel couldn't add a CQE because the ring was full). The application must handle this condition by resizing rings or reducing load.	CQ overflow, error handling, ring sizing
`IORING_CQE_F_NOTIF`	A flag set in a CQE to indicate that this completion is a notification for a zero-copy operation, signaling that the kernel has released its reference to the buffer.	`IORING_OP_SEND_ZC`, zero-copy, `release_buffer_after_zc`
`IORING_ENTER_GETEVENTS`	A flag passed to `io_uring_enter` to indicate that the call should wait for at least `min_complete` completions if they are not immediately available. Enables blocking behavior.	`io_uring_enter`, `engine_submit_and_wait`, event loop
`IORING_OP_ACCEPT`	io_uring operation code for asynchronously accepting incoming TCP connections on a listening socket.	`IORING_ACCEPT_MULTISHOT`, `submit_accept_multishot`, network server
`IORING_OP_ASYNC_CANCEL`	io_uring operation code for canceling in-flight operations, useful for cleaning up pending reads/writes	`conn_destroy`, `CONN_CLOSING`, operation cancellation

Term	Definition	Related Concepts
	when a connection is closed prematurely.	
`IORING_OP_READ`	io_uring operation code for asynchronous read operations from a file descriptor (file or socket).	`prepare_async_file_read`, `submit_connection_read`, file I/O
`IORING_OP_SEND_ZC`	io_uring operation code for zero-copy send operations on a socket. The kernel sends data directly from the provided buffer without copying, but the buffer must remain valid until a notification CQE arrives.	zero-copy, `IORING_CQE_F_NOTIF`, `submit_zero_copy_send`
`IORING_OP_WRITE`	io_uring operation code for asynchronous write operations to a file descriptor.	`submit_connection_write`, file I/O, network I/O
`IORING_REGISTER_BUFFERS`	Opcode for `io_uring_register` to register a set of buffers as fixed buffers, allowing SQEs to reference them by index.	fixed buffers, `create_buffer_pool`, `io_uring_register`
`IORING_SETUP_SQPOLL`	A flag for `io_uring_setup` that enables kernel-side submission queue polling: a kernel thread periodically checks the SQ for new entries, reducing the need for `io_uring_enter` syscalls.	`engine_config`, kernel polling, performance optimization
`IOSQE_IO_DRAIN`	A flag placed on an SQE to enforce ordering: all previously submitted SQEs must complete before this SQE starts execution. Use sparingly as it serializes execution.	ordering guarantees, serialization, performance impact
`IOSQE_IO_LINK`	A flag placed on an SQE to link it to the next SQE in the submission ring, creating a chain of operations that execute sequentially. If any operation fails, the chain stops.	linked operations, chain atomicity, `submit_linked_echo_chain`
`LEAST_CONN`	Load balancing algorithm constant that assigns new connections to the backend server with the fewest active connections.	`load_balancer`, `backend_server`, health checking
Library Book Retrieval System	A mental model for asynchronous file I/O: submitting a read request is like giving a book request slip to a librarian and continuing other work until notified the book is ready at the pickup counter.	async file I/O, `io_uring`, mental model
Liburing	A user-space helper library provided by the Linux kernel community that simplifies `io_uring` setup and operations with convenient wrapper functions.	`io_uring_setup`, `io_uring_enter`, `io_uring_get_sqe`
Linked chain partial failure	A scenario in a linked SQE chain where an operation in the middle	`IOSQE_IO_LINK`, chain atomicity, error handling

Term	Definition	Related Concepts
	fails, causing the remaining operations in the chain not to be executed. The application must handle cleanup of any partially completed chain.	
Linked operations	A sequence of SQEs marked with `IOSQE_IO_LINK` that execute sequentially, with the output of one potentially feeding into the next. The chain succeeds or fails as a unit.	`IOSQE_IO_LINK`, `submit_linked_echo_chain`, chain atomicity
`load_balancer`	A structure representing a load balancer that distributes incoming connections across multiple backend servers using algorithms like round-robin or least-connections.	`backend_server`, `RR`, `LEAST_CONN`
`loop_config`	Configuration structure for the event loop, controlling batch sizes, wait behavior (`busy_wait`), and flags for tuning performance.	`event_loop_create`, `event_loop_run`, busy-wait
`LATENCY`	Load balancing algorithm constant that assigns new connections to the backend server with the lowest average latency.	`load_balancer`, `backend_server`, health checking
`lookup_op_context(table, user_data)`	Function that finds the `io_op_context` associated with a given `user_data` token by performing a lookup in the context table.	`allocate_op_context`, `free_op_context`, `context_table`
Memory barriers	CPU instructions that enforce ordering constraints on memory operations, ensuring that reads and writes to the shared ring buffers between the application and kernel are properly synchronized.	`io_uring` rings, synchronization, multi-threading
Multishot	A mode for certain `io_uring` operations (like accept and receive) where a single SQE can generate multiple CQEs over time, automatically re-submitting itself after each completion to reduce submission overhead.	`IORING_ACCEPT_MULTISHOT`, `submit_accept_multishot`, network server
Multishot accept	An accept operation configured with the `IORING_ACCEPT_MULTISHOT` flag, which continuously accepts incoming connections and produces a CQE for each, without requiring a new SQE per connection.	`IORING_OP_ACCEPT`, `IORING_ACCEPT_MULTISHOT`, network server
`network_server`	The top-level structure for the asynchronous TCP server, containing the io_uring engine, connection table, listening socket, and run state.	`network_server_create`, `network_server_run`, `connection_table`

Term	Definition	Related Concepts
`network_server_create(port, engine)`	Function that creates a network server instance, binding a listening socket on the specified port and initializing the connection table.	`network_server_run`, `network_server`, `conn_table_create`
`network_server_run(server)`	The main network event processing loop that submits initial operations (multishot accept) and then runs the event loop to handle I/O completions.	`event_loop_run`, `submit_accept_multishot`, network server
Notification CQE	A special completion queue entry marked with `IORING_CQE_F_NOTIF` that signals the kernel has released its reference to a buffer used in a zero-copy send operation.	`IORING_OP_SEND_ZC`, `IORING_CQE_F_NOTIF`, zero-copy
`OP_CONTEXT_MAGIC`	A magic number (`0xDEADC0DE`) embedded in `io_op_context` structures to detect memory corruption or use-after-free bugs during debugging.	debugging, memory safety, `io_op_context`
`OP_TYPE_ACCEPT_MULTI`	Internal constant representing a multishot accept operation, used in the `op_type` field of `io_op_context` for routing completions.	`io_op_context`, operation type, completion handling
`OP_TYPE_READ`	Internal constant representing a file read operation.	`io_op_context`, `prepare_async_file_read`, file I/O
`OP_TYPE_READ_LINKED`	Internal constant representing a read operation that is part of a linked chain.	`io_op_context`, `IOSQE_IO_LINK`, linked operations
`OP_TYPE_WRITE_LINKED`	Internal constant representing a write operation that is part of a linked chain.	`io_op_context`, `IOSQE_IO_LINK`, linked operations
Object pooling	A performance optimization technique where frequently allocated and freed objects (like `io_op_context` or `io_buffer`) are reused from a pool rather than being allocated from the heap each time.	`context_table`, `buffer_pool`, `acquire_buffer`
Opportunistic batching	A submission strategy where SQEs are prepared over multiple iterations of the event loop but only submitted to the kernel when a batch reaches a certain size or when the loop is about to wait for completions.	`loop_config.max_batch_size`, `engine_submit`, performance
`PARSE_COMPLETE`	Parser status constant indicating that a complete HTTP request has been successfully parsed.	`http_parser_feed`, `STATE_COMPLETE`, HTTP parsing
`PARSE_ERROR`	Parser status constant indicating that the parser encountered an error	`http_parser_feed`, error handling, HTTP parsing

Term	Definition	Related Concepts
	(e.g., malformed request).	
`PARSE_INCOMPLETE`	Parser status constant indicating that more data is needed to complete parsing the HTTP request.	`http_parser_feed`, `STATE_HEADERS`, incremental parser
`prepare_async_file_read(ring, file_fd, buf, offset, len, user_data)`	Function that prepares an SQE for an asynchronous file read operation, setting `opcode` to `IORING_OP_READ` and filling in file descriptor, buffer, offset, and user_data.	`IORING_OP_READ`, file I/O, SQE preparation
`RR`	Load balancing algorithm constant for round-robin assignment, where connections are assigned to backend servers in a cyclic order.	`load_balancer`, `backend_server`, `rr_index`
Reference counting	A technique for tracking the number of active references to a resource (like a buffer used in zero-copy send) to determine when it can be safely reused or freed.	`io_buffer.zc_refcount`, `acquire_buffer_zc`, `release_buffer_after_zc`
Refcount bugs	Errors in reference counting logic that can lead to use-after-free (if a buffer is released while still referenced) or memory leaks (if references are never released).	`io_buffer.zc_refcount`, zero-copy, debugging
`release_buffer(pool, buffer)`	Function that returns a buffer to the pool, marking it as not in use and making it available for future acquisitions.	`acquire_buffer`, `buffer_pool`, buffer lifecycle
`release_buffer_after_zc(pool, buf, is_notification)`	Specialized buffer release function for zero-copy operations: decrements the kernel reference count and only returns the buffer to the pool when the count reaches zero.	`acquire_buffer_zc`, `IORING_OP_SEND_ZC`, zero-copy
Single-threaded	An execution model where the server runs in a single thread, avoiding locking overhead and complexity but limited to a single CPU core. Can be extended with a worker pool for multi-core scalability.	event loop, concurrency, `worker_pool`
`STATE_BODY`	HTTP parser state constant: reading the request body (after headers).	`http_parser_t`, `PARSE_INCOMPLETE`, HTTP parsing
`STATE_COMPLETE`	HTTP parser state constant: request parsing is complete.	`http_parser_t`, `PARSE_COMPLETE`, HTTP parsing
`STATE_HEADERS`	HTTP parser state constant: parsing HTTP headers.	`http_parser_t`, `PARSE_INCOMPLETE`, HTTP parsing
`STATE_START_LINE`	HTTP parser state constant: parsing the HTTP request line (method, URI, version).	`http_parser_t`, `PARSE_INCOMPLETE`, HTTP parsing

Term	Definition	Related Concepts
State machines	Models that track the progression of an entity (like a connection or parser) through a defined set of states, with transitions triggered by events. Used extensively for connection and protocol management.	`connection_state`, `http_parser_t`, `conn_state_t`
`submit_accept_multishot(server)`	Function that submits a multishot accept SQE to the io_uring instance, which will generate a CQE for each incoming connection without needing re-submission.	`IORING_OP_ACCEPT`, `IORING_ACCEPT_MULTISHOT`, network server
`submit_connection_read(server, conn)`	Function that prepares and submits an SQE to read data from a connection's socket into its read buffer.	`CONN_READING`, `io_uring_sqe`, network I/O
`submit_connection_write(server, conn, data, len)`	Function that prepares and submits an SQE to write data to a connection's socket from the provided buffer.	`CONN_WRITING`, `io_uring_sqe`, network I/O
`submit_linked_accept_read_write_chain(server, accept_sqe, read_sqe, write_sqe, conn)`	Function that sets up a linked chain of three SQEs: accept → read → write, for handling an echo server request in a single atomic submission.	`IOSQE_IO_LINK`, linked operations, network server
`submit_linked_echo_chain(server, conn)`	Function that submits a linked READ→WRITE chain for echoing received data back to the client, a common pattern for echo servers.	`IOSQE_IO_LINK`, `handle_linked_chain_completion`, network server
`submit_zero_copy_send(ring, sock_fd, buf, len, user_data)`	Function that submits a zero-copy send operation, incrementing the buffer's reference count and preparing an SQE with `IORING_OP_SEND_ZC`.	`IORING_OP_SEND_ZC`, zero-copy, `io_uring_sqe`
Submission Queue (SQ)	The ring buffer where the application places `io_uring_sqe` structures describing I/O operations to be performed. The kernel consumes entries from this queue.	`io_uring_sqe`, `io_uring_get_sqe`, `io_uring_submit`
SQ	Abbreviation for Submission Queue.	`io_uring_sqe`, `io_uring_submit`
SQE	Submission Queue Entry – a `struct io_uring_sqe` that describes a single I/O operation to be performed by the kernel.	`io_uring_sqe`, `io_uring_get_sqe`, `io_uring_submit`
`test_expectation`	A structure representing an expected completion in a test, containing the `user_data` token, expected result code, and expected buffer contents.	`test_validator`, testing, Milestone 1
`test_validator`	A structure used in unit tests to track expected completions and verify that	`test_validator_create`, `test_validator_record_completion`, testing

Term	Definition	Related Concepts
	actual completions match expectations.	
`test_validator_add_expectation(validator, index, user_data, expected_result, expected_buffer, expected_len)`	Function that adds an expected completion to a validator for later verification.	`test_validator_create`, `test_validator_record_completion`
`test_validator_all_completed(validator)`	Function that checks whether all expected completions have been recorded.	`test_validator_create`, test completion
`test_validator_create(count)`	Function that creates a validator with capacity for a specified number of expected completions.	`test_validator_destroy`, testing
`test_validator_destroy(validator)`	Function that frees a validator and its resources.	`test_validator_create`, cleanup
`test_validator_record_completion(validator, user_data, actual_result, actual_buffer, actual_len)`	Function that records an actual completion, comparing it against the expected completion with the matching `user_data`.	`test_validator_add_expectation`, verification
`thread_worker`	A structure representing a worker thread in a thread pool, containing its thread ID, assigned io_uring engine, connection table, and work queue.	`worker_pool`, `worker_thread_main`, multi-threading
Thread-local storage	A mechanism for storing data that is unique to each thread, not shared between threads. Useful for per-worker statistics or caches.	`thread_worker`, multi-threading, performance
Tight loops	Code loops that run without yielding (e.g., busy-waiting for completions) and can consume excessive CPU, potentially starving other processes. Must be used judiciously.	busy-wait, `loop_config.busy_wait`, performance tuning
Transient error	An error category (ERR_CATEGORY_TRANSIENT) indicating a temporary condition that may resolve if the operation is retried (e.g., `EAGAIN`, `EINTR`).	`classify_error`, `handle_io_error`, retry logic
`try_steal_work(thief, thief_idx)`	Function called by an idle worker thread to attempt to steal pending connections or operations from other workers' queues, improving load balancing.	work stealing, `worker_pool`, load balancing
`udp_session`	A structure representing a UDP communication session, tracking the remote address, packet counts, and protocol-specific context.	UDP server, stateless protocol

Term	Definition	Related Concepts
User_data	A 64-bit user-defined token included in each SQE that is copied to the corresponding CQE upon completion, allowing the application to correlate requests with completions.	`io_uring_sqe.user_data`, `io_uring_cqe.user_data`, correlation tokens
User_data correlation	The process of matching completion entries to their originating operations by comparing the `user_data` token in the CQE with tokens stored in application context structures.	`lookup_op_context`, `conn_lookup`, `decode_user_data`
User_data token	The unique identifier encoded into the `user_data` field, typically combining connection ID, operation type, sequence number, and buffer ID for efficient lookup.	`encode_user_data`, `decode_user_data`, correlation
`wal`	Write-Ahead Log structure: a file where operations are logged before execution to ensure durability and allow recovery after a crash.	`wal_record`, durability, crash recovery
`wal_record`	A structure representing a single entry in the write-ahead log, containing sequence number, operation type, timestamp, operation details, and checksum.	`wal`, durability, crash recovery
`worker_pool`	A structure managing a pool of worker threads, each with its own io_uring instance and connection table, for scaling across multiple CPU cores.	`thread_worker`, `assign_connection_to_worker`, multi-threading
`worker_pool_create(worker_count)`	Function that creates and starts a pool of worker threads. If `worker_count` is 0, defaults to the number of CPU cores.	`worker_thread_main`, `worker_pool`, multi-threading
`worker_thread_main(arg)`	The main function executed by each worker thread, running an event loop on its private io_uring engine and connection table.	`thread_worker`, `event_loop_run`, multi-threading
Work stealing	A load balancing technique where idle threads actively take work (e.g., pending connections) from busy threads' queues, improving overall utilization.	`try_steal_work`, `worker_pool`, multi-threading
Zero-copy	A data transfer technique where data is moved between kernel and user space or between file descriptors without copying the data, improving performance by reducing CPU usage and memory bandwidth.	`IORING_OP_SEND_ZC`, `IORING_CQE_F_NOTIF`, `acquire_buffer_zc`
Zero-copy notification	A special completion entry (`IORING_CQE_F_NOTIF`) that signals the kernel has released its	`IORING_OP_SEND_ZC`, `IORING_CQE_F_NOTIF`, buffer lifecycle

Term	Definition	Related Concepts
	reference to a buffer used in a zero-copy send operation, allowing the buffer to be reused.	
Zero-copy reference counting	The mechanism for tracking kernel references to buffers used in zero-copy operations via the `zc_refcount` field in `io_buffer`. The buffer can only be released when the count reaches zero.	`io_buffer.zc_refcount`, `acquire_buffer_zc`, `release_buffer_after_zc`