1. Program Overview

1.1 Purpose

This document outlines the technical assessment requirements for our 4–6-week internship evaluation program. Candidates will select ONE project from the four options provided and complete it in phases over the evaluation period.

1.2 Program Structure

- Duration: 4-6 weeks (full-time)
- Evaluation: Phased delivery with weekly checkpoints
- Project Selection: Choose ONE from four options
- Weekly Check-ins: Progress reviews and guidance
- Mentorship: Assigned technical mentor for support

1.3 Evaluation Phases

1.3.1 Phase 1: Core Implementation (Weeks 1-2) - 40 points

- Complete all mandatory core features
- Basic API functionality working end-to-end
- Data models properly designed and implemented
- Error handling and validation
- Basic documentation

1.3.2 Phase 2: Feature Extensions (Weeks 3-4) - 30 points

- Select and implement TWO extension features from provided options
- Maintain code quality standards
- Add unit tests for new features
- Integration with core system

1.3.3 Phase 3: Optimization & Advanced Features (Weeks 5-6) - 20 points

- Choose ONE advanced optimization or feature
- Performance improvements or production-ready enhancements
- Scalability considerations
- Advanced testing or monitoring

1.3.4 Documentation & Presentation - 10 points

- Comprehensive README with setup instructions
- API documentation (Swagger/Postman)
- Code comments and inline documentation
- Architecture diagram
- Final presentation and live demo

1.4 Evaluation Criteria for each phase

Criteria	Percentage (%)	Description
Functionality	20%	All core features working correctly
Code Quality	15%	Clean code, proper structure, SOLID principles
Data Model Design	5%	Well-designed database schema
Extension Features	15%	Two features fully integrated
Testing	5%	Comprehensive test coverage
Optimization	15%	Performance or advanced features
System Design	5%	Scalability and architecture
Documentation	5%	Clear and complete docs
Presentation	10%	Effective demo and explanation
Code Repository	5%	Clean Git history, good commits
TOTAL	**100%**	

1.5 Technology Guidelines

1.5.1 Backend Technologies

- Java: Spring Boot
- Spring Boot with Spring Batch or custom pipeline engine (for applicable assignments)
- Apache Kafka (for applicable assignments)

1.5.2 Database (Choose One):

- PostgreSQL (recommended for most projects)
- Oracle
- MongoDB
- SQLite (for simple persistence layer projects only)

1.5.3 Additional Tools (Optional but Recommended):

- Redis: For caching and queues
- Docker: For containerization
- Git: Version control (mandatory)
- Prometheus: For metrics
- Grafana: For dashboards
- k6 or Apache Bench: For load testing
- Testing: JUnit or equivalent

1.6 Submission Requirements

- GitHub/GitLab repository with complete source code
- README.md with setup and running instructions
- Database schema and migration scripts
- API documentation (Postman collection or Swagger)
- Test cases and test results
- Recorded demo video (5-10 minutes) or live presentation

4 Project Option 3: API Rate Limiter Service

4.1 Project Overview

Build a rate limiting service/middleware that enforces request limits per user, IP, or API key. Prevents API abuse and ensures fair usage. Implements algorithms like Token Bucket or Sliding Window.

4.1.1 Key Features Summary

- Phase 1: Rate limiting algorithm (Token Bucket/Sliding Window), Request validation, HTTP 429 responses, Rate limit headers
- Phase 2 Options: Multiple tiers (Free/Pro/Enterprise), Distributed limiting (Redis), Advanced rules (per-endpoint, time-based), Analytics dashboard
- Phase 3 Options: High performance (10K+ req/sec), Circuit breaker, Whitelist/Blacklist, Cost-based limiting

4.2 Phase 1: Core Implementation (Weeks 1-2)

4.2.1 Rate Limiting Algorithms

You MUST implement ONE of these algorithms:

4.2.2 Algorithm 1: Token Bucket (Recommended)

Concept:

- Bucket holds tokens (max capacity = rate limit)
- Tokens refill at constant rate (e.g., 10 tokens per second)
- Each request consumes 1 token
- If bucket empty, request is rejected (429 Too Many Requests)
- Allows brief bursts above rate limit (if tokens accumulated)

Parameters:

- Capacity: Maximum tokens (e.g., 100)
- Refill Rate: Tokens per second (e.g., 10/sec = 600/min)
- Current Tokens: How many tokens available now
- Last Refill Time: When tokens were last added

Example:

- Capacity: 100 tokens
- Refill Rate: 10 tokens/second
- User makes 120 requests instantly: First 100 succeed, next 20 fail
- After 2 seconds: 20 new tokens available

4.2.3 Algorithm 2: Sliding Window Counter

Concept:

- Track request count in current window
- Window slides with time (not fixed boundaries)
- More accurate than fixed windows

Parameters:

- Limit: Max requests in window (e.g., 100)

- Window: Time period (e.g., 60 seconds)
- Previous Window Count: Requests in previous window
- Current Window Count: Requests in current window

- **Comparison**:
- **Token Bucket:**
  - ✓ Allows bursts (good for spiky traffic)
  - ✓ Simple to implement
  - ✗ Less precise at window boundaries
- **Sliding Window:**
  - ✓ More accurate rate limiting
  - ✓ Smoother throttling
  - ✗ Slightly more complex
-

4.2.4 Mandatory Features

4.2.4.1 Rate Limit Validation

- Check if request is allowed before processing
- Identify user by: user_id, IP address, or API key
- Increment request counter for each call
- Return HTTP 429 if limit exceeded
- Include rate limit headers in response

4.2.4.2 Configurable Limits

- Set limits per identifier type (user, IP, API key)
- Configure max requests and time window
- Support multiple time windows (per second, minute, hour)
- Store configuration in database or config file

4.2.4.3 Rate Limit Reset

- Automatic reset after time window expires
- Manual reset via admin API (for testing)
- Clear reset time in response headers

4.2.4.4 Storage Management

- In-memory storage for Phase 1 (HashMap)
- Efficient lookup: O(1) time complexity
- Automatic cleanup of old entries
- Memory-efficient data structures

-

4.2.5 Phase 1 Deliverables

4.2.5.1 Deliverables Required:

- Rate limiting algorithm implemented (Token Bucket OR Sliding Window)
- Rate limit check API endpoint
- Configuration management (database or config file)
- HTTP 429 responses with proper headers
- In-memory storage working
- Status check endpoint
- Reset endpoint for testing
- Postman collection
- Unit tests (minimum 5 tests)
- README with setup and usage

4.2.5.2 Success Criteria:

- Submit 100 requests → First 100 succeed, 101st gets 429
- Wait for refill → New requests succeed
- Different identifiers tracked separately
- Headers present in all responses
- Reset endpoint clears limits
- System handles concurrent requests correctly
-

4.3 Phase 2: Feature Extensions (Weeks 3-4)

Choose and implement TWO features:

4.3.1 Option A: Multiple Tiers (Free/Pro/Enterprise)

- Implement tiered rate limiting with different limits per subscription level.

4.3.1.1 Requirements:

- Define multiple tiers: FREE, PRO, ENTERPRISE, UNLIMITED
- Each tier has different limits
- API to assign tier to user/API key
- Automatic tier detection during rate limit check
- Upgrade/downgrade capability

4.3.1.2 Testing Scenarios:

- FREE user limited to 100 req/hour
- PRO user gets 1000 req/hour
- Upgrade FREE → PRO immediately increases limits
- Tier expiration downgrades to FREE automatically
-

4.3.2 Option B: Distributed Rate Limiting with Redis

- Scale rate limiting across multiple servers using Redis.

4.3.2.1 Requirements:

- Replace in-memory storage with Redis
- Multiple API servers share same Redis instance
- Atomic operations using Lua scripts or Redis commands
- TTL-based automatic cleanup
- Works correctly with horizontal scaling

4.3.2.2 Redis Data Structure (Token Bucket):

- # Key pattern: rate_limit:{identifier}
  # Value: Hash with fields
  HSET rate_limit:user_123 tokens 95
  HSET rate_limit:user_123 capacity 100
  HSET rate_limit:user_123 refill_rate 0.0278 # 100/3600
  HSET rate_limit:user_123 last_refill 1706871600
  EXPIRE rate_limit:user_123 3600 # Auto-cleanup after window

4.3.2.3 Testing:

- Start 2-3 API instances
- Send requests to different instances
- Verify total requests across all instances don't exceed limit
- Stop Redis, verify graceful degradation
-

4.3.3 Option C: Advanced Rules Engine

- Implement complex rate limiting rules beyond simple per-user limits.

4.3.3.1 Requirements:

- Per-endpoint rate limits (different limits per API path)
- Time-based rules (lower limits during peak hours)
- Combined limits (e.g., 1000/hour AND 10/second)
- IP-based + User-based combined limits
- Rule priority and evaluation order

4.3.3.2 Advanced Rule Examples:

1. Per-Endpoint Limits:

- POST /api/search → 10 requests/second
  POST /api/upload → 5 requests/minute
  GET /api/users → 100 requests/hour
  GET /api/status → Unlimited (no rate limit)

2. Time-Based Rules:

- Peak Hours (9 AM - 5 PM): 50 requests/hour
  Off-Peak (5 PM - 9 AM): 200 requests/hour
  Weekend: Unlimited

3. Burst Allowance:

- Normal: 100 requests/hour
  Burst: Allow up to 120 requests in 5 minutes
  Then: Throttle to hourly limit

4.3.3.3 Testing:

- Create rule: /api/search limited to 10/sec
- Create rule: /api/upload limited to 5/min
- Verify each endpoint enforces its own limit
- Test time-based rules by changing system time
- Test rule priority (higher priority evaluated first)
-

4.3.4 Option D: Analytics Dashboard

- Build real-time analytics to monitor rate limiting activity.

4.3.4.1 Requirements:

- Track all rate limit checks and violations
- Identify top users hitting rate limits
- Show usage trends over time
- Alert on unusual patterns (potential attacks)
- Export data for further analysis

4.3.4.2 Metrics to Display:

- Overview Panel:
- Total requests today
- Rate limited requests (count and percentage)
- Unique identifiers tracked
- Most rate-limited users
- Time Series Charts:
- Requests per minute/hour (line chart)
- Rate limit violations over time
- Success rate percentage
- User Analysis:
- Top 10 users by request count
- Top 10 users by rate limit violations
- User behavior patterns
- Endpoint Analysis:
- Most requested endpoints
- Endpoints with highest rate limit hits

4.3.4.3 Testing:

- Submit 1000 requests from various identifiers
- Verify dashboard updates in real-time
- Check top violators list accuracy
- Export data and verify completeness
-

4.4 Phase 3: Advanced Features (Weeks 5-6)

Choose ONE advanced feature:

4.4.1 Option A: High Performance (10K+ req/sec)

- Optimize for extreme performance: handle 10,000+ requests per second.

4.4.1.1 Requirements:

- Benchmark: Sustain 10,000 requests/second
- Latency: P95 < 10ms, P99 < 50ms
- Optimize algorithm implementation
- Use efficient data structures
- Load testing with k6 or Apache Bench
- Performance profiling and bottleneck analysis

4.4.1.2 Optimization Techniques:

1. Lock-Free Data Structures:

- ```
  // Java: Use Atomic operations
  AtomicLong tokens = new AtomicLong(capacity);

  public boolean tryConsume() {
      while (true) {
          long current = tokens.get();
          if (current < 1) return false;

          if (tokens.compareAndSet(current, current - 1)) {
              return true;
          }
          // Retry if CAS failed
      }
  }
  ```

2. Reduce Redis Round Trips:

  - Use Lua scripts for atomic operations (single round trip)
  - Pipeline multiple commands
  - Connection pooling

3. Local Caching:

- ```
  // Cache rate limit configs locally
  Cache<String, RateLimitConfig> configCache = Caffeine.newBuilder()
      .maximumSize(10000)
      .expireAfterWrite(5, TimeUnit.MINUTES)
      .build();

  // Reduce database queries
  RateLimitConfig config = configCache.get(key,
      k -> configRepo.findByKey(k));
  ```

4. Async Processing:

  - Non-blocking I/O
  - Async logging (don't wait for log writes)
- Batch analytics writes

4.4.1.3 Benchmarking Results to Include:

- Throughput: Requests per second
- Latency: P50, P95, P99, P99.9
- Error rate: Percentage of failed requests
- CPU and memory usage
- Comparison: Before vs after optimization

4.4.1.4 Deliverables:

- Optimized implementation
- Load test scripts
- Performance benchmarks with graphs
- Profiling analysis
- Optimization documentation
-

4.4.2 Option B: Circuit Breaker Pattern

- Automatically block abusive users temporarily, then gradually allow them back.

4.4.2.1 Requirements:

- Detect users hitting rate limit repeatedly
- Auto-block for escalating durations (5min, 15min, 1hr, 24hr)
- Gradual recovery: reduce block time if behavior improves
- Manual override (unblock user)
- Notification when circuit opens

4.4.2.2 Circuit States:

- CLOSED: Normal operation, rate limiting active
- OPEN: Identifier blocked entirely (all requests rejected immediately)
- HALF_OPEN: Testing recovery (allow limited requests)

4.4.2.3 Testing:

- Hit rate limit 5 times → Verify auto-blocked for 5 minutes
- Hit rate limit 10 times → Verify escalation to 15 minutes
- Wait for block to expire → Verify HALF_OPEN state
- Manual unblock via API → Verify immediately accessible
-

4.4.3 Option C: Whitelist/Blacklist Management

- Maintain lists of identifiers that bypass rate limits or are permanently blocked.

4.4.3.1 Requirements:

- Whitelist: Identifiers that bypass all rate limits
- Blacklist: Identifiers that are permanently blocked
- CRUD APIs for list management
- Import/export lists (CSV/JSON)
- Audit log for list changes
- Expiration support (temporary whitelist/blacklist)

4.4.3.2 Testing:

- Add identifier to whitelist → Verify unlimited requests allowed
- Add identifier to blacklist → Verify all requests rejected immediately
- Import CSV with 100 identifiers → Verify all imported correctly
- Export whitelist → Verify CSV format correct
- Set expiration → Wait for expiration → Verify auto-removal
-

4.4.4 Option D: Cost-Based Rate Limiting

- Different operations consume different "costs" from the user's quota.

4.4.4.1 Concept:

- Instead of all operations costing 1 request, assign costs based on resource usage. User has a point budget per time window.

4.4.4.2 Examples:

- GET /api/users/{id} → 1 point (simple read)
- GET /api/search?query=... → 5 points (complex query)
- POST /api/reports/generate → 50 points (expensive operation)
- POST /api/ai/analyze → 100 points (AI processing)
- User Budget: 1000 points per hour
- Can make 1000 simple reads
- OR 200 search queries
- OR 20 report generations
- OR mix of operations

4.4.4.3 Testing:

- User has 1000 points
- Call expensive operation (50 points) 20 times → Uses 1000 points
- 21st call rejected → Insufficient points
- Call cheap operation (1 point) after expensive ones → Verify cost tracked correctly
- Wait for window reset → Verify points restored
-

4.4.4.4 Implementation Guidelines

- Algorithm Choice: Token Bucket for simplicity, Sliding Window for accuracy
- Concurrency: Use atomic operations or locks
- Storage: Redis for distributed, in-memory for single instance
- Headers: Always include X-RateLimit-* headers
- Testing: Unit tests, integration tests, load tests
-