ChatGPT

Product Requirements Document (Custom HTTP Proxy)

Purpose and Scope

A *custom HTTP proxy* will act as an intermediary between clients (such as browsers or apps) and backend servers, forwarding HTTP/HTTPS requests and returning responses. This proxy is designed to improve performance, security, and flexibility. For example, forward proxies can cache content (like DNS results or web pages) to reduce bandwidth usage [1], while reverse proxies can distribute load across servers, cache static assets, and compress responses for faster delivery [2]. Our custom proxy will also provide security benefits: inspecting traffic to block malicious content [3], enforcing access rules, and hiding client IPs (anonymity) [4]. These features support corporate use cases such as enterprise security filtering, content restrictions, and bandwidth optimization [5] [1].

The proxy must handle all standard HTTP methods and features. It will accept HTTP/1.1 (and ideally HTTP/2/3) requests, support the `CONNECT` method for tunneling HTTPS, and manage persistent connections and chunked transfer encoding [6] [7]. In practice, **clients connect to the proxy**, which decrypts SSL (if configured), applies routing/filtering rules, forwards requests to the appropriate backend or CLI process, and returns the (possibly cached or compressed) response to the client [8] [2]. This design hides internal server details and enables centralized policy enforcement. The proxy will run as a standalone service (e.g. containerized) that can be scaled horizontally behind a load balancer.

Goals and Objectives

- **High Performance:** Introduce minimal latency, support high throughput (thousands of requests/sec), and use persistent connections. Implement optional response caching and compression to speed delivery [2].
- **Robust Security:** Enforce TLS encryption (terminate/re-encrypt or tunnel HTTPS via `CONNECT` [6]), validate input, and filter malicious traffic [3]. Hide or log original client IPs securely [4]. Support access control (IP whitelists/blacklists, domain-based blocking) so that requests matching policy are blocked or altered.
- **Streaming Support:** Handle large/real-time data by streaming. Use HTTP chunked encoding to send data incrementally [7]. For example, if the proxy invokes a CLI tool that generates output (such as a video transcoder), it must stream that tool's stdout/stderr to the HTTP client in real time [7] [9].
- **Flexibility & Customization:** Allow customizable routing and filtering rules. Administrators should be able to configure which requests are forwarded, blocked, or modified. The proxy should support rewriting headers and URLs, injecting custom headers, and defining rules (e.g. block example.com/*, or only allow certain domains). As with other proxies, requests can include `X-Forwarded-*` headers to pass original client info [10].
- **Reliability & Observability:** Target high availability (e.g. 99.9% uptime). Log all requests and errors for auditing. Provide metrics (requests/sec, latencies, error rates) for monitoring. The system should

fail gracefully: for example, return meaningful HTTP error codes if the backend or CLI fails, and retry transient errors when appropriate.

- **Scalability:** Design the proxy to be stateless so multiple instances can run in parallel. Use a load balancer to distribute client connections. Employ connection pooling to backends. Ensure it can handle spikes by scaling out (e.g. container orchestration).
- **Maintainability:** Write clean, modular code with configuration via files or environment variables. Include automated tests. Use industry-standard libraries (e.g. Node.js `http-proxy` or similar) for core HTTP handling to speed development and ensure security best practices.

Functional Requirements

- **HTTP Request/Response Handling:**
- *Protocol Support:* Accept all standard HTTP/HTTPS methods (GET, POST, PUT, DELETE, OPTIONS, etc.) and HTTP versions. Implement HTTP/1.1 fully (including keep-alive and chunked transfer encoding [7]). For HTTPS, support the `CONNECT` method to tunnel through the proxy [6]; optionally, also allow TLS termination at the proxy (decrypting client TLS and optionally re-encrypting to backends) to enable content inspection and compression [8].
- *Routing:* Forward incoming requests to designated backend servers or services. Support configurable routing rules based on URL path or host header. For example, requests to `/api/*` might go to one server, and `/static/*` to another.
- *Header Management:* Correctly preserve or rewrite headers. Add or forward `X-Forwarded-For/Proto/Host` as needed to identify original client info [10]. Allow injecting custom headers (for authentication tokens, client IDs, etc.) and support removal of sensitive headers (e.g. stripping client IP or cookies if needed).
- *Response Processing:* Forward backend responses to the client. If caching or compression is enabled, the proxy may modify the response body or headers (e.g. gzip the payload). Responses should be streamed to the client as received; large or infinite streams must not be buffered fully. Use HTTP chunked encoding for streaming data [7].
- **Content Filtering & Security Policies:**
- *Malicious Content Inspection:* Analyze request URLs and response content. Block or sanitize requests/responses that match malicious patterns (SQL injection signatures, malware payloads, etc.), similar to a lightweight web application firewall. For example, as noted by Fortinet, proxies can “examine web traffic to identify and block malicious activity or content” [3].
- *Access Control:* Support whitelist/blacklist rules. Administrators can specify domains, URL patterns, or file types to block. For instance, block downloads of `.exe` files or requests to disallowed domains. Also enforce authentication if required (e.g. require an API key in requests, or client TLS certificates for validation). As with content restrictions described in Fortinet, filters may be based on domain, pathname, or file extension [11].
- *Anonymity:* By default, the proxy should hide backend server identities from clients (i.e. client sees only the proxy’s IP). It may optionally mask or omit client IPs from logs, providing anonymity for clients if needed [4].
- **Caching & Compression:**
- *Static Caching:* Optionally cache frequent static responses (images, CSS/JS, etc.) in memory or a local cache store. Cached content should be served without re-querying the backend, reducing load and bandwidth [2] [5]. Implement cache eviction (e.g. LRU policy) and respect HTTP cache headers.
- *Response Compression:* Optionally compress responses (e.g. GZIP/Brotli) to clients when supported, reducing transfer size and improving load times [2].

- **Logging & Monitoring:**
- *Access Logs:* Record every request and response with details: timestamp, client IP, target URL, response code, response size, and latency. Include any applied rule (e.g. "blocked by filter") in logs. Logs should be written to a file or logging system for later analysis.
- *Metrics:* Collect and expose runtime metrics: number of active connections, requests per second, average latency, error rates, cache hit/miss rates, etc. Provide an endpoint or integration (e.g. Prometheus) for monitoring these metrics.
- **CLI Integration & Streaming:**
- *Process Execution:* The proxy must be able to invoke external command-line tools (e.g. video transcoder, data processor) as part of handling a request. For example, a request to `/video/stream` could cause the proxy to run `ffmpeg` to transcode on-the-fly.
- *Real-time Streaming:* Data produced by CLI processes must be streamed to the HTTP client immediately (without waiting for process completion). As with HTTP streaming, use chunked transfer encoding. The proxy should forward both `stdout` and `stderr` of the process as HTTP data, or log them, so that real-time output (progress logs or errors) can be seen by the client or operator. For example, tools like Streamlink provide a mode to "serve stream data through HTTP" for external clients [9]. Similarly, our proxy will continuously read the process output and flush it to the client as it arrives, ensuring minimal delay.
- **Configuration & Management:**
- *Configuration:* Define proxy behavior via a configuration file or environment variables. This includes routing rules, filter rules, TLS certificates/keys, logging destinations, etc. The proxy should reload or update configuration without downtime if possible.
- *Administration Interface:* Provide a CLI or management API for operational tasks (e.g. graceful shutdown, flush cache, view current connections). A full web UI is **not** required (out of scope).

Non-Functional Requirements

- **Performance:** The proxy must introduce minimal overhead. Aim for end-to-end latency overhead of only a few milliseconds. It should handle at least *X* concurrent connections (e.g. thousands of RPS on moderate hardware) without degradation. Use asynchronous I/O and connection pooling to backend servers. Employ efficient buffering to avoid blocking (for example, adjustable proxy buffer sizes).
- **Scalability & Availability:** Design for horizontal scaling. Multiple instances of the proxy can run behind a load balancer. The service should have no single point of failure: if one proxy instance dies, others should continue serving traffic. Support graceful restarts (draining existing connections). Aim for 99.9% availability (enterprise-grade).
- **Reliability:** The proxy should handle errors gracefully. For backend timeouts or failures, return appropriate HTTP error codes. If a CLI process crashes, capture its error output and return HTTP 5xx to the client. Implement retry logic for transient network errors when forwarding requests. Ensure the proxy itself does not crash on malformed input.
- **Security:**
- *Transport Security:* All client-proxy communications should support TLS 1.2+ (strong ciphers) by default. The proxy may act as an HTTPS server with its own certificate, or work behind a TLS-terminating load balancer. For backend connections, either re-encrypt or use plain HTTP depending on requirements.
- *Input Validation:* Sanitize all parts of incoming requests to prevent header injection, path traversal, or buffer overflow attacks. Use well-tested libraries for parsing HTTP. Limit allowed header sizes and URL lengths.

- *Access Control:* Protect the proxy's management interface (if any) with authentication. Disable dangerous features (e.g. shell execution beyond authorized commands). Keep dependencies up to date to avoid known vulnerabilities.
- **Maintainability:** Write modular, well-documented code. Include unit and integration tests covering key scenarios (e.g. routing rules, error handling, streaming). Use CI/CD for automated builds and testing. The codebase should be easily extensible for future features (e.g. adding new filters or protocols).
- **Logging & Auditability:** Logs should include enough detail to audit user actions (e.g. who made a request if authentication is used). Keep sensitive data out of logs (e.g. sanitize passwords).
- **Compliance:** If used in a regulated environment, ensure logs and data handling meet relevant standards (e.g. GDPR, HIPAA) by anonymizing or securing sensitive fields. (Not specified, but allow extension if needed.)

System Architecture

The proxy will be a standalone network service (likely containerized) that listens on one or more public ports. Clients connect to the proxy's endpoint (over HTTP or HTTPS). Internally, the proxy operates in these stages: *receive request* → *apply security/filter rules* → *route to backend/CLI* → *stream response back* → *log/metrics* [8] [3]. A simplified architecture is:

- **Client:** Web browser or app issuing HTTP(S) requests.
- **Proxy Server:** Our custom service. It may consist of modules such as:
- *Listener*: accepts TCP connections (HTTP/HTTPS).
- *Request Processor*: parses incoming HTTP requests, authenticates (if needed), applies filter rules.
- *Router*: determines target (backend server URL or local CLI command) based on configuration.
- *Handler*: forwards request to the target. For backend servers, it opens a connection and proxies data. For CLI, it spawns a process and hooks its stdin/stdout to the HTTP stream.
- *Response Streamer*: reads data from backend or process and writes to client using appropriate HTTP encoding (chunked if streaming).
- *Logger/Monitor*: records request/response details and updates metrics.
- **Backends/Processes:** Actual services or commands that perform the work. Could be databases, application servers, or CLI tools. These are outside the proxy code. The proxy must manage connections to these, and for CLI processes, manage their lifecycle (timeout, kill on disconnect).
- **Config Store:** Configuration for routes, filters, TLS certs, etc., stored in a file or centralized config service. The proxy should reload or pick up changes without full restart if possible.

Key architectural choices: use an event-driven runtime (e.g. Node.js, Go) for high concurrency. Ensure the proxy is *stateless* wherever possible (no in-memory session data) so it can scale out. Health-check endpoints should be provided so orchestrators (Kubernetes, etc.) can monitor proxy health and restart if needed.

Use Cases

1. **Basic Forwarding:** A user's browser requests `http://proxy.company.com/page.html`. The proxy receives this GET, applies no filters (or an allowlist passes it), and forwards it to `http://backend1.internal/page.html`. When the backend responds, the proxy streams the HTML back to the browser. The client sees only the proxy's address in the URL bar. *Key functions:* request parsing, header forwarding, response streaming, logging.

2. **HTTPS Tunneling:** The user browses `https://secure.example.com` via the proxy. The browser sends `CONNECT secure.example.com:443 HTTP/1.1`. The proxy opens a TCP tunnel to `secure.example.com` and relays encrypted bytes between client and server. No filtering is done inside (raw TLS), but the proxy logs the CONNECT request. *Key functions:* handle `CONNECT`, TCP tunneling.
3. **Domain Blocking:** An administrator has configured the proxy to block `example-bad.com/*`. A user requests `http://example-bad.com/malware`. The proxy matches the URL against its block list and immediately returns a 403 Forbidden without contacting the backend. *Key functions:* URL filtering, policy enforcement, error response.
4. **Caching Static Content:** A client requests a static image `/images/logo.png`. The first time, the proxy fetches it from the backend and caches the result. Subsequent requests for `/images/logo.png` are served directly from cache, reducing backend load. *Key functions:* cache lookup, conditional request if needed, cache invalidation.
5. **CLI Data Streaming:** A client makes a GET to `/transcode?file=bigvideo.mp4`. The proxy spawns a CLI tool (e.g. `ffmpeg`) to transcode the video in real time. As `ffmpeg` outputs video data, the proxy immediately sends these bytes to the client using HTTP chunked encoding, so the client can start playing before transcoding finishes. Errors from `ffmpeg` are also streamed (or cause a 5xx). This is analogous to tools like Streamlink, which can "serve stream data through HTTP" from the CLI [9] . *Key functions:* process execution, real-time stdout/stderr streaming, chunked transfer.
6. **Monitoring and Metrics:** Operations staff hit an admin endpoint on the proxy (e.g. `/metrics`) to retrieve current statistics (RPS, errors). They also check logs to verify that a recent suspicious request was blocked as intended. *Key functions:* metrics collection, logging, admin API (read-only).

Success Criteria

- **Functionality:** All features listed above are fully implemented and tested. The proxy passes integration tests for routing, filtering, caching, and streaming.
- **Performance:** The proxy introduces negligible additional latency (e.g. <50ms) and supports the target load (e.g. 1000+ RPS) with <1% error rate. Chunked streaming responses start delivering data within a second of request.
- **Reliability:** The proxy achieves the uptime target (e.g. 99.9%) under load testing. It recovers gracefully from failures (e.g. auto-restarts on crashes, no resource leaks).
- **Security:** No critical vulnerabilities (OWASP Top 10) remain. The proxy successfully enforces all filtering rules (verified by tests). TLS termination is correctly implemented.
- **Observability:** Metrics and logs provide sufficient insight. For example, if a filter blocks traffic, the admin can see which rule triggered. Logs contain full request/response summaries.
- **Maintainability:** Code coverage meets the threshold (e.g. >80%). Documentation is complete. New developers can understand and extend the proxy without major confusion.

Out of Scope

- **Non-HTTP Protocols:** The proxy will **not** support WebSockets, raw TCP other than HTTP CONNECT tunneling, or higher-level protocols like gRPC natively. It focuses on HTTP/HTTPS traffic only.
- **User Interface:** No graphical UI/dashboard for users or admins; management is via config files, CLI commands, or scripts.

- **Persistent Data Storage:** The proxy will not include a database for storing application data. Logs may be written to external systems, but the proxy itself does not retain request/response bodies beyond ephemeral caching.
- **Advanced WAF Features:** Deep-packet inspection or custom application-layer firewalls (beyond rule-based filtering of URLs/headers) are not provided in version 1.
- **Third-Party Integrations:** Integration with external authentication services (OAuth servers, LDAP) or telemetry systems (beyond basic metrics) is not required initially.

Assumptions and Dependencies

- **Platform:** The proxy will run on Linux (or Linux containers). It can rely on a POSIX environment to spawn CLI processes.
- **Libraries/Frameworks:** We assume availability of robust HTTP libraries (e.g. Node.js `http`/ `http2`, Go's `net/http`, or similar). Using a well-known proxy library (like `node-http-proxy`) is acceptable. For metrics, libraries like Prometheus client may be used.
- **Network Environment:** Clients must be able to direct traffic to the proxy (e.g. via DNS or corporate proxy settings). If behind a load balancer, the proxy's health-check endpoint should be reachable. The proxy has outbound internet or intranet access to backends.
- **TLS Certificates:** The proxy can use provided TLS certificates/keys for HTTPS, or be placed behind an existing TLS terminator.
- **CLI Tools:** Any external CLI that the proxy will invoke (e.g. `ffmpeg`, `streamlink`, etc.) must be installed on the host system and accessible in the proxy's execution environment. It is assumed these tools can produce output quickly enough to meet streaming requirements.
- **Resource Limits:** The deployment will allocate sufficient CPU/RAM for expected load. For high-load scenarios, the proxy instances will be replicated.
- **Security Requirements:** The proxy code will run with least-privilege (no root needed unless port 80/443 are privileged). It will not execute untrusted code except as configured by admins.

Sources: The above requirements are informed by standard proxy usage and best practices. For example, proxies commonly enable caching to "save vast amounts of bandwidth" [5], distribute load and optimize content [2], and filter malicious traffic [3]. Supporting streaming (e.g. chunked encoding) is important for real-time data [7]. Tools like Streamlink demonstrate how a CLI can serve HTTP streams [9]. These principles guided our requirements.

---

[1] [2] [6] [10] Proxy servers and tunneling - HTTP | MDN
https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Proxy_servers_and_tunneling

[3] [4] [5] [11] What is HTTP Proxy? | Fortinet
https://www.fortinet.com/resources/cyberglossary/http-proxy

[7] What Is HTTP Chunked Encoding? How Is It Used?
https://bunny.net/academy/http/what-is-chunked-encoding/

[8] Day 16: System Design Concept: Reverse Proxy | by Shivani Mutke | Medium
https://medium.com/@shivanimutke2501/reverse-proxy-a933d8745608

[9] Command-Line Interface - Streamlink 8.0.0 documentation
https://streamlink.github.io/cli.html