ChatGPT

BRRTRouter as a Unified Proxy Gateway with AuthZ for Microservices

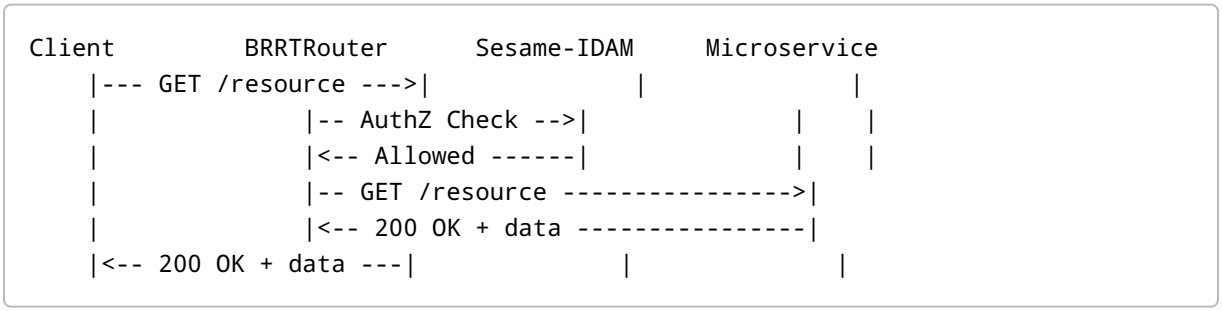
BRRTRouter is envisioned to serve as a high-performance gateway, routing requests to downstream microservices across HTTP, gRPC, and WebSocket protocols. A core requirement is that **BRRTRouter performs authorization (authZ) checks** before relaying requests, using an external service like **Sesame-IDAM** – a Rust-based identity and access management system [1]. In this report, we provide a deep technical analysis of this architecture, including sequence flows, authZ mechanism comparisons, infrastructure concerns, integration recommendations for Sesame-IDAM, and code structure suggestions.

1. Request Flows & Sequence Diagrams

Below we examine typical request flows through BRRTRouter acting as a reverse proxy, including how authorization is enforced and how errors are handled. Each diagram shows the interaction between the **Client**, **BRRTRouter**, **Sesame-IDAM** (auth service), and the **Downstream Microservice**.

1.1 HTTP Proxying (GET/POST Requests)

For HTTP requests (e.g. RESTful API calls), BRRTRouter intercepts the request, checks authorization via Sesame-IDAM, and then proxies the request to the appropriate microservice if allowed. The sequence diagram below illustrates a **successful** HTTP GET flow with authZ and forwarding:



1. **Client → Router:** A client sends an HTTP request (e.g. `GET /resource`). BRRTRouter parses the request and identifies the target route.
2. **Router → Sesame-IDAM:** Before forwarding, the router extracts the client's credentials (e.g. JWT access token in an `Authorization` header) and calls Sesame-IDAM to **authorize** the request. This might be a REST or gRPC call to an `/authorize` endpoint with the token and request details.
3. **Sesame-IDAM → Router:** Sesame-IDAM validates the token (checking signature, expiration, etc.) and evaluates the user's permissions (e.g. roles or scopes) for the requested resource. It returns an **Allow** or **Deny** decision (and possibly user info or scopes). If denied, BRRTRouter will immediately stop and return an error to the client (see error handling below). If allowed, the router proceeds [2].
4. **Router → Microservice:** The router **proxies the request** to the appropriate microservice. It may translate the URL or host based on routing rules (e.g. `/resource` might map to a specific service

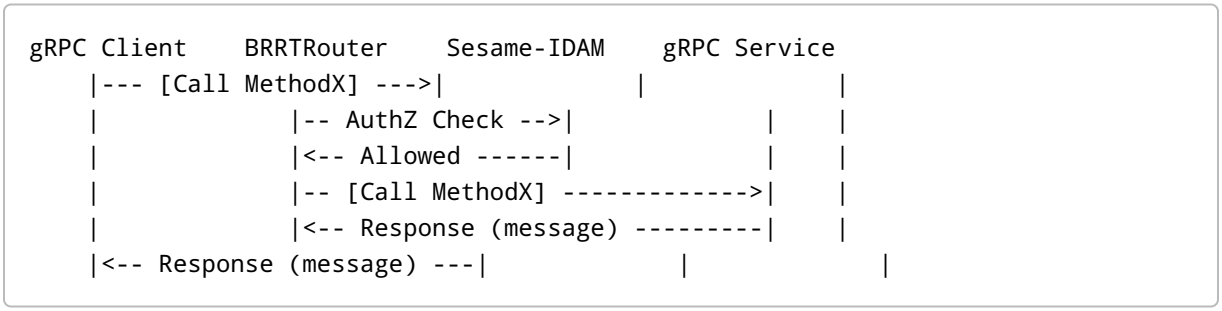
and path). The router opens a connection (keep-alive HTTP connection or uses a connection pool) to the microservice and forwards the HTTP method, path, headers, and body. Hop-by-hop headers like `Connection` are removed, and the router may add identifying headers (e.g. X-Request-ID) or propagate tracing headers.

5. **Microservice → Router:** The downstream service processes the request and sends back a response. This could be a **success** (e.g. 200 OK with JSON data) or an error/status code. BRRTRouter streams the response as it arrives – reading the status, headers, and body without unnecessarily buffering large payloads.
6. **Router → Client:** BRRTRouter forwards the microservice's response back to the client. Status code and body are relayed transparently. In a success case, the client receives the 200 OK and data. If the microservice returned an error (e.g. 404 or 500), the router passes that along (possibly translating certain errors if configured to do so).

POST/PUT requests: The flow for state-changing requests like POST or PUT is very similar. The difference is that BRRTRouter must handle a **request body** (e.g. JSON payload). It will stream the incoming body to the microservice, possibly buffering if needed (for example, if content needs to be inspected or if the microservice is slower to read). The authZ check for POST may also consider the body or content-type if required (though usually authZ is based on headers/roles). After processing, the microservice's response (201 Created, etc.) is relayed back. The router ensures that **all HTTP verbs** are supported (GET, POST, PUT, DELETE, etc.) and that unknown routes return a 404 or 501 as appropriate [3].

1.2 gRPC Proxying with AuthZ

For gRPC traffic, BRRTRouter can act as a **gateway** by terminating the gRPC call from the client and then acting as a gRPC client to the downstream service. The process involves intercepting the call for authZ and then forwarding it. The sequence for a unary gRPC call is as follows:



1. **Client → Router:** A gRPC client invokes a method (e.g. `MethodX`) on BRRTRouter's gRPC endpoint (BRRTRouter would expose the same service interface as the downstream). The call comes over HTTP/2. BRRTRouter's gRPC server (built with a library like Tonic) receives the request.
2. **Router → Sesame-IDAM:** An interceptor or middleware in the router checks the **metadata** (e.g. the `authorization` metadata with a JWT). It calls Sesame-IDAM (possibly via gRPC or REST) to validate the token and permissions for `MethodX`. Sesame-IDAM returns allow/deny as before. (If denied, the router can return a gRPC error code like `UNAUTHENTICATED` or `PERMISSION_DENIED` to the client.)
3. **Router → Service:** If authorized, the router now forwards the gRPC call. This can be done by using a **gRPC client stub** for the downstream service. For instance, the router might use the same protobuf definitions to serialize the request and send it to the microservice's gRPC endpoint. All this happens

within the router's process (it could maintain a pool of gRPC channel connections to downstream services).

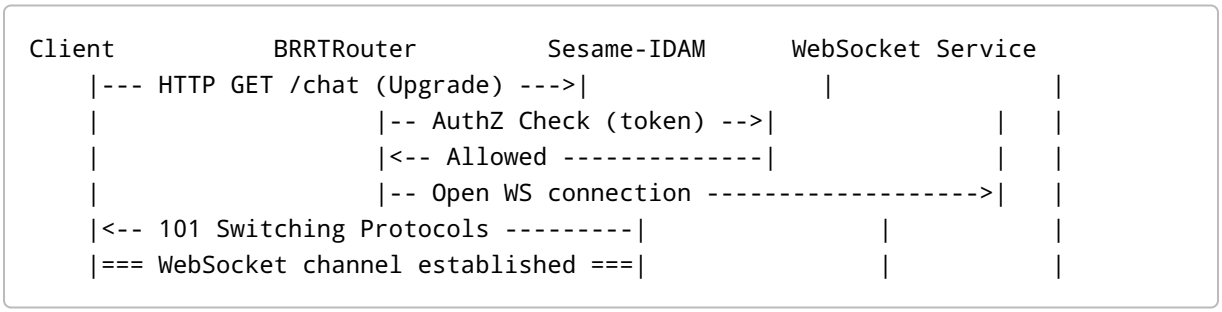
4. **Service → Router:** The microservice handles the gRPC call and returns a response message (or error). With gRPC, this could also be a stream (server-streaming or bidirectional). BRRTRouter receives the response on its client stub.
5. **Router → Client:** The router then sends the response back over the client's gRPC connection. For unary calls, this is a single message and status trailer. For streaming calls, BRRTRouter will forward stream messages incrementally as they arrive, acting as a message proxy. The client gRPC library receives the response as if it came directly from the service.

Throughout this process, the router must preserve gRPC-specific metadata and status codes. For example, if the downstream service returns a gRPC error (with status code and message), BRRTRouter should propagate that to the client. The additional latency introduced by authZ and forwarding is minimal (a few milliseconds) if Sesame-IDAM is fast and the network is local.

1.3 WebSocket Handshake and Message Relaying

BRRTRouter can also proxy **WebSocket** connections, which start as an HTTP Upgrade handshake and then become a persistent bidirectional socket. The router needs to handle the handshake, perform authZ on the initial request (and possibly continuously), and then shuttle messages between the client and microservice.

Handshake flow: The WebSocket handshake is an HTTP GET with `Upgrade: websocket` headers. BRRTRouter will authorize this like any HTTP request, then forward the upgrade to the service. For example:



1. **Client → Router:** Client requests a WebSocket upgrade (e.g. `GET /chat` with appropriate headers).
2. **Router → Sesame-IDAM:** The router checks the auth token (perhaps a query param like `?token=...` or a cookie/header) with Sesame-IDAM before proceeding with the upgrade. If not authorized, the router can reject the handshake (e.g. returning HTTP 401/403 instead of 101 Switching Protocols).
3. **Router → Service:** If authorized, BRRTRouter initiates a WebSocket connection to the target microservice (e.g. connecting to the internal host/port that provides the `/chat` service). This involves sending the Upgrade request to the service.
4. **Service → Router:** The microservice accepts the upgrade and responds with **101 Switching Protocols**, establishing the WebSocket. BRRTRouter relays this 101 response back to the client, completing the handshake. At this point, the router holds two WebSocket connections: one with the client, one with the service. The router can then **tie them together** so that messages flow in both directions.

Message relaying: After the handshake, the router simply forwards WebSocket frames. When the client sends a message, the router receives the frame and forwards it to the service over the backend WS connection. When the service sends a message, the router passes it to the client. This is typically done in a streaming loop with minimal inspection. The router may optionally enforce message size limits or timeouts, but generally it operates at the frame level. The connection remains open until either side closes it. Authorization for individual messages is usually not re-checked for WebSockets (the assumption is that the initial handshake authorized the session), but the router *could* enforce certain policies (for instance, disconnect if a message rate limit is exceeded – see rate limiting later).

1.4 Error Handling Paths

A robust router must handle various error conditions gracefully:

- **Authorization Failure:** If Sesame-IDAM denies a request (e.g. token invalid or user lacks permission), BRRTRouter will **not** contact the microservice at all. It returns an HTTP 403 Forbidden or 401 Unauthorized to the client immediately. For gRPC, it would return an error status (e.g. `PERMISSION_DENIED`). This fail-fast behavior protects the downstream systems from unauthorized calls. (In a sequence diagram, the flow would stop after the authZ service returns "deny". The router might send back a JSON error body or gRPC status detail as needed.)

- **Downstream Timeout:** If a microservice does not respond within a configured timeout, BRRTRouter will abort the request and return a **504 Gateway Timeout** to the client [4]. Internally, the router likely sets a timer per request (or uses a client library timeout). This prevents clients from hanging indefinitely and frees up router resources. The router might log the timeout event and could trigger circuit breaker logic (see §3.2). A partial sequence of a timeout: client request -> authZ allow -> request sent to service -> **no response** -> router cancels the request and replies 504 to client.

- **Downstream Error / Malformed Response:** If the microservice responds with an error status (HTTP 5xx/4xx or gRPC error), BRRTRouter propagates it to the client. A **502 Bad Gateway** is used if the response is so malformed that the router can't even interpret it (e.g. invalid HTTP). For instance, if the service connection drops or returns gibberish, the router sends 502 to indicate an upstream failure [4]. If the service returns a well-formed error (say a 400 or 500 with a JSON error), the router usually relays it as-is. The router thus acts transparently, while adding the appropriate gateway error status when it detects a fundamental proxying failure.

- **WebSocket Errors:** If the backend WebSocket service refuses the connection (e.g. returns HTTP 403 or doesn't support Upgrade), the router will forward that failure to the client (the handshake fails, client gets an HTTP error). If after connection the service goes down or sends an unexpected closure, the router should close the client connection accordingly. The router might attempt to send a WebSocket **Close frame** with an appropriate code. Since WebSocket is stateful, error handling is essentially closing the connection or possibly trying to reconnect if configured.

In all cases, BRRTRouter should **log and monitor** these errors (e.g. counting authZ failures, timeouts, etc.) to aid in debugging and reliability (more in §3.6).

2. Comparative Overview of Authorization Mechanisms

Implementing authorization in a microservices gateway can be done via different mechanisms. We compare a few approaches relevant to BRRTRouter and Sesame-IDAM:

2.1 JWT Verification & Introspection

JWT (JSON Web Token) is a common method for conveying authentication and authorization claims. There are two patterns:

- **Local JWT Verification:** The gateway verifies the JWT's signature and claims locally (using a public key or secret) and makes authorization decisions based on the token's claims (like roles or scopes). This is *fast* and **stateless** – ideal for high throughput microservices, since no external call is needed [5]. BRRTRouter could decode the JWT in-memory and check, for example, that the token is not expired and that it has a claim allowing access to the requested route. Local verification is highly scalable (O(1) per request) [5]. However, any changes in user permissions or revocations won't be known until the token expires. Thus, short token lifetimes or refresh mechanisms are used to mitigate stale permissions.

- **Token Introspection (Opaque Tokens):** Instead of self-contained JWTs, the gateway can use opaque tokens and call an **introspection endpoint** on Sesame-IDAM for each request. Sesame-IDAM would then confirm if the token is active and return the user's scopes/roles. The advantage is **real-time authZ** – tokens can be revoked or permissions changed centrally, and the next request will reflect that. It also avoids exposing token details to the client (better secrecy). The downside is performance: a network call to auth service for every request can become a bottleneck in high-throughput systems [6]. Caching can mitigate this (the router could cache introspection results for a short time) [7], but that reintroduces some staleness. Opaque tokens also add an external dependency (if Sesame-IDAM is down, auth may fail open or closed). In practice, many systems use **hybrid** strategies – e.g. short-lived JWT access tokens for stateless verification and longer-lived opaque refresh tokens for updates [8] [9].

Pros: JWTs offer very low-latency authZ (just signature verify) and work well in microservices where each service can verify the token independently [5]. Introspection offers central control (instant revocation, detailed policy checks by Sesame). JWTs are also technology-neutral (work for HTTP, gRPC, etc. as long as the token is passed along).

Cons: Local JWT verification requires secure key management and doesn't account for real-time revocation easily. Introspection adds **latency and scaling overhead** [6] – a high-volume gateway might overwhelm an auth service if not scaled appropriately or cached. There's also a *security tradeoff:* self-contained JWTs, if stolen, could be used until expiry; introspected tokens can be killed server-side, improving security if an access token leak occurs [10] [11].

2.2 OPA Policy Evaluation

OPA (Open Policy Agent) is a policy engine that can externalize authorization logic from the application. Instead of hard-coding rules or simply checking token claims, BRRTRouter could query OPA with details of the request (user, path, method, etc.) and a policy (written in Rego) would decide allow/deny. This setup

might involve running an OPA sidecar or service that BRRTRouter consults on each request [2] (similar to how Envoy Proxy's external auth filter works [12]).

- **Pros:** OPA enables very **fine-grained and dynamic policies**. For example, you could allow "DELETE /item" only if the user is an admin *and* the item is in a certain state, etc., without hardcoding in the Rust code. Policies can be updated at runtime. OPA's Rego language and engine are general-purpose, making it possible to express RBAC, ABAC (attribute-based), and relationship-based access rules in one system. It also produces decision logs for audit. By decoupling policy from code, OPA allows **central management of authZ** across many services [13].

- **Cons:** Introducing OPA adds a new component with its own performance costs. Each request might incur a call to the OPA engine (though it can be in-process via a library or run as a local sidecar to reduce latency). Policy evaluation is fast (micro-milliseconds) but not free; complex rules or large data sets could slow down decisions [14]. There is also a **complexity overhead**: developers must learn Rego and maintain policies separately from code. OPA itself must be kept in sync with the latest user data (it might pull data from Sesame-IDAM or a database to know user roles, etc.). That said, OPA is designed for high performance and can scale horizontally. Many microservice architectures successfully use OPA for API gateway authZ (e.g. with Envoy/OPA or Traefik plugins) [2] [15].

In BRRTRouter's context, one could envision Sesame-IDAM itself incorporating an OPA engine or BRRTRouter using OPA as a *policy-as-a-service* to which it passes JWT claims and route info. For example, NGINX or Envoy pipelines often send an **authorization check request** to a policy service before forwarding traffic [2], which is analogous to what BRRTRouter would do with Sesame or OPA.

2.3 OAuth2 Token Exchange & Delegation

In complex microservice ecosystems, the end-user's token may not be directly accepted by backend services (due to scope or audience restrictions). **OAuth2 Token Exchange (RFC 8693)** is a mechanism where the gateway can exchange the client's token for a **new token** suited to internal services [16]. This is a form of delegation:

- **Delegation use-case:** The client authenticates to the gateway with a token intended for the gateway (audience = gateway). The gateway, after authZ, contacts the IdP (Identity Provider, possibly Sesame-IDAM if it plays that role) to get a **delegated token** for the downstream service. The exchanged token might have a different scope or audience (targeting the specific microservice) and potentially a shorter lifetime. The gateway then calls the microservice with this internal token [17] [18], rather than the client's original token. This improves security because the microservice only sees tokens that it can validate (audience matches) and with minimal privileges needed for that service.

- **Impersonation vs. delegation:** Token exchange can support both. **Impersonation** means the new token essentially makes the gateway act as the user (the microservice doesn't know the difference – it sees the user's identity). **Delegation** means the new token carries both the original user identity and the fact that it's being used by an intermediary (gateway) – so the microservice knows the request is on behalf of user X via service Y [19] [20]. Delegation is generally preferable for auditing.

Pros: Token exchange adds an extra layer of security and isolation. Backend services can have their own trust domain. It also allows **scope reduction** – the gateway can drop extraneous scopes and only request a narrow scope for the backend, enforcing the principle of least privilege [16]. It can also handle token type conversion (e.g. exchange a JWT for a reference token known internally). If using Sesame-IDAM, it could act as the IdP that issues exchanged tokens.

Cons: The obvious cost is an extra round-trip to the IdP for each exchange (though some gateways cache exchanged tokens per session to avoid doing it every time). It also complicates the architecture: the gateway needs credentials to use the token exchange grant (often it itself needs to be a registered OAuth client with permission to perform exchanges). Implementing this grant type can be non-trivial. However, many modern systems (Keycloak, Auth0, etc.) support token exchange, and libraries exist.

For BRRTRouter, token exchange might be an **optional advanced feature**. Initially, BRRTRouter can simply pass the incoming token through to the microservice (assuming a shared trust in the JWT). But in a zero-trust environment, using token exchange ensures that even internal calls are authenticated with a proper audience-specific token [17]. This pattern is recommended when an API gateway calls multiple downstream APIs each with their own auth requirements [21] [22].

Summary:

- **JWT local authZ** is fastest but requires short token expiry or recheck for revocation.
- **Central introspection** (Sesame-IDAM on every request) gives up-to-date control at the cost of latency, mitigated by caching strategies [23].
- **OPA policies** provide powerful fine-grained control decoupled from code, at the cost of an extra policy layer.
- **OAuth2 delegation** (token exchange) helps enforce proper *end-to-end* security context for microservice calls, trading additional complexity.

In practice, BRRTRouter can combine these: e.g. do local JWT validation for basic integrity and expiration, then call Sesame-IDAM or OPA for fine-grained permission checks (using cached info if available), and possibly use token exchange when forwarding to certain services that require it. The design should remain flexible to accommodate these patterns as needed.

3. Key Infrastructure Concerns

Building BRRTRouter as a production-grade gateway involves addressing several infrastructural and cross-cutting concerns:

3.1 Timeout and Retry Strategies

Timeouts: BRRTRouter should enforce timeouts on external interactions to avoid hanging resources. This includes timeouts for: (a) **AuthZ service calls** (Sesame-IDAM) – if Sesame doesn't respond within, say, 50-100ms, the router might abort the auth check and deny the request or use a fallback decision; (b) **Downstream microservice calls** – depending on the endpoint, a sensible timeout (e.g. a few seconds for typical REST calls) prevents long waits. Timeouts can often be tuned per route (e.g. a report-generation API might allow a longer timeout than a quick GET). If a timeout is hit, the router returns a 504 Gateway Timeout as noted [4].

Retries: In a distributed system, transient failures happen. BRRTRouter can automatically **retry** certain requests to improve reliability. For safety, retries are usually applied to **idempotent** operations (GET, maybe PUT) to avoid duplicate side effects. For example, if a microservice returns a network error or times out, the router might try the request again once (perhaps to a different instance if load balancing). Exponential backoff delays should be used to avoid thundering herds. Also, integrate with the **circuit breaker** (below): if a circuit is open (service is known down), don't bother retrying immediately.

For gRPC, which has built-in deadline propagation, BRRTRouter should set or respect **deadlines** on the incoming call. If the client's deadline is soon, the router may skip retries to avoid exceeding it. For WebSockets, retries are less applicable (they represent a long-lived session; if it fails to connect, the client must retry the session).

Client-side vs internal retries: Sometimes clients will implement their own retries. The gateway should be careful not to *overlap* too many retries (e.g. a client that retries 3 times and a gateway that retries internally 3 times could result in 9 attempts). A best practice is to coordinate via headers or idempotency keys if possible, or simply document the behavior.

3.2 Circuit Breakers and Fallback

A **circuit breaker** is a pattern to avoid continually sending requests to an unhealthy service. BRRTRouter can monitor the error rates or timeouts of each downstream microservice. If a service is consistently failing (e.g. 5 timeouts in a row), the router can "open" the circuit for that service: for a short period, *stop forwarding* requests and instead return an error immediately, or route to an alternative if available [24]. This prevents overload on a struggling service and allows it to recover. The router can periodically "half-open" the circuit to test if the service is back, and then close the circuit when it responds successfully.

Fallback responses: In some cases, it's useful to have a fallback. For example, if a microservice is down, perhaps BRRTRouter could serve a cached response (if available), or a default message. Another possibility is a *static fallback service* – e.g. if the live service is down, route to a simplified service that returns "Please try again later" or some limited functionality [24]. Spring Cloud Gateway and others support routing to a fallback URI when a circuit breaker triggers. In Rust, one could implement this by configuring an alternate handler for that route when the breaker is open.

BRRTRouter would likely implement circuit breaking as part of its internal service client. Libraries like **tower** (Tower is a middleware layer for Rust services) offer ready-made middleware for retries and circuit breaking. The router can maintain counters for each backend and trip circuits when thresholds exceed (errors %, consecutive failures, etc.). When tripped, the router should log the event (and ideally emit a metric or alert).

3.3 Rate Limiting and Quota Enforcement

Rate limiting prevents any single client (or token, or IP) from over-consuming the API. BRRTRouter, being the gateway, is the ideal place to enforce rate limits **per endpoint or per user/API key** [25]. Sesame-IDAM itself lists *Rate limiting* as a feature, likely for API keys and users [26] – integration between router and Sesame can help (Sesame could store policy rules like "user X can do 100 requests/min").

Several types of rate limits to consider:

- **Global limit per API/route:** e.g. at most 500 req/sec to the `/login` endpoint to protect the backend from overload or brute force attacks. The router counts requests and starts rejecting (HTTP 429 Too Many Requests) when limits exceed.
- **Per-User or per-Token limit:** e.g. each user can call `/reports` at most 5 times per minute. The router would key by user ID or token and enforce a sliding window or token-bucket counter [27].
- **Per-IP limit:** a fallback if token info is unavailable – limit by IP to mitigate abuse or DDoS from a single IP.
- **Quota (long-term limits):** e.g. 1000 calls per day for a free tier API key. This requires persistent counters (could be stored in Redis or in Sesame-IDAM if it tracks usage).

Implementing rate limiting in Rust could use a concurrency-friendly structure (like atomic counters or a dashmap) for short intervals, or a dedicated service for more complex policies. The **leaky bucket** or **token bucket** algorithms are common; there are crates like `governor` or `ratelimit` that can be used. For distributed rate limiting beyond one router instance, a centralized store (Redis or Sesame-IDAM service) might be needed so that limits are consistent across a cluster.

Enforcement: When a limit is exceeded, the router should reject the request quickly. HTTP 429 is the usual status. Optionally, it can include a `Retry-After` header to tell the client when to try again. In gRPC, this would translate to a status code like `RESOURCE_EXHAUSTED`. The router should also ensure that the rejection itself doesn't get overly verbose (to avoid aiding an attacker); just a simple error is fine.

Because BRRTRouter will likely integrate with Sesame-IDAM, **sharing rate limit data** is prudent. Sesame might maintain API key usage records [28] and inform the router. For instance, Sesame could include in the authZ response whether the token is within quota. In absence of that, the router does its own counting.

3.4 Caching (AuthZ Results and Responses)

Caching authZ results: To reduce load on Sesame-IDAM, BRRTRouter can cache recent auth decisions. For example, if token XYZ was just checked and allowed for route `/data`, the router could cache that "XYZ is allowed for /data" for a short time (say 1 minute or the token's TTL). Then subsequent requests with the same token and route can skip the Sesame call, as long as no significant change is expected in that short window. This cache must be carefully scoped: include token ID, user or scopes, and the specific route/action in the key. Also, it should honor any **revocation signals** – e.g. if Sesame-IDAM provides a webhook or a timestamp of last revocation, the router should invalidate cache for that token if needed. As noted earlier, systems often **cache introspection results** to mitigate performance costs [23]. This is essentially what the router would be doing. If using JWTs, the router might not need an explicit cache – the token itself carries info and a short expiry. But for fine-grained authZ (like OPA policies), caching "policy allow" decisions for a bit can help.

Caching responses: BRRTRouter can optionally serve as a caching proxy for GET requests to improve performance and reduce load on microservices. If certain endpoints are cacheable (static or public data), the router can store the response in memory (or distributed cache) and return it for subsequent requests. This would involve honoring HTTP cache headers (`Cache-Control`, `ETag`, etc.) and possibly providing a cache invalidation mechanism (maybe tied to events or time-to-live). For instance, a GET /config that rarely changes could be cached for 5 minutes at the router. This reduces latency for clients and offloads backend.

The router must be cautious to cache only safe responses and handle **varying** (e.g. by user, by locale) if applicable.

Given performance goals (millions of req/sec) [29] [30], an in-memory cache within BRRTRouter (using lock-free maps or sharded locks) could serve frequent requests extremely quickly. The cache size and eviction policy (LRU, TTL) should be configurable per route. Also, integration with **circuit breakers**: during an outage, returning cached data (even if slightly stale) might be better than failing – a strategy sometimes called *graceful degradation*.

3.5 Streaming Support and Payload Buffering

Streaming: Both gRPC and WebSocket involve streaming data. BRRTRouter must handle streaming efficiently. This means using non-blocking I/O and backpressure mechanisms. For gRPC (over HTTP/2), Rust's async facilities (Tokio) or the coroutine runtime (May) can manage multiple streaming calls concurrently. The router should **pipeline** data as it comes – read a chunk from the source and write to the destination without waiting for the entire message (this is especially important for large file uploads/downloads or server-sent events). If BRRTRouter uses an async HTTP library or its coroutine `may_minihttp`, it can operate on data frames directly.

Payload buffering: In some cases, buffering is unavoidable. For example, if the router needs to compute a request body's hash or fully parse JSON (perhaps for OPA to use attributes in the body for authZ), it must read the whole body. This requires memory and increases latency. The design should minimize such scenarios for large payloads. Ideally, **pass-through streaming** is used: for instance, reading from the client socket and writing to the microservice socket simultaneously. Rust's memory safety and ownership can help avoid copying data more than necessary (using references or bytes that move between read and write tasks).

For WebSocket frames, the router might buffer a complete frame in memory if required (depending on the WebSocket library) but generally frames are not huge (and can be limited by configuration). It should also handle **ping/pong** frames if the library doesn't automatically respond (keepalive for the connection).

Backpressure: It's critical that BRRTRouter propagates backpressure. If a microservice is slow to consume data, the router's write to that service will back up; the router should then slow reading from the client. In async terms, futures should not be continuously polled if the sink is not ready. This prevents unbounded buffering and memory blow-up. Similarly for responses: if the client is slow (e.g. downloading over a 3G network) but the service produces data fast, the router may need to buffer some data or tell the service to slow down (TCP windowing will naturally do some of this). Configuration of max buffer sizes (per connection) ensures that beyond a point, the router will drop data or terminate the connection if a client can't catch up, to avoid exhaustion.

Chunking and compression: The router should support streaming data with chunked transfer (for HTTP/1.1) and respect content-encoding. If the router is not modifying the response, it can just pass through compressed data. If it needs to inspect or modify, it might have to decompress then recompress, which is expensive, so better to avoid modification in the proxy path.

In summary, streaming support means writing the proxy in an asynchronous, event-driven style. BRRTRouter's coroutine approach is well-suited for this, as each connection can be a coroutine that yields

when it can't read/write, allowing thousands of concurrent streams. The design goal is to handle even large file transfers or high-frequency message streams without significant overhead.

3.6 Observability (Logging, Tracing, Metrics)

Observability is crucial in such an infrastructure component:

- **Logging:** BRRTRouter should produce structured logs for each request. At minimum, logging the method, path, response code, and latency is useful (common log format). For auth, logging whether a request was allowed or denied (and which policy or rule caused a deny) is important for security audits. Logs should include a correlation ID or trace ID if available, to tie into distributed tracing. They should also be configurable in verbosity (e.g. debug logging for headers, etc., vs info level for one-liners). Sensitive data (tokens, personal info) should be redacted in logs. Sesame-IDAM likely also logs authentication events; combining these can help trace a user's journey.

- **Tracing:** To get end-to-end insight, BRRTRouter can propagate distributed tracing headers (such as Zipkin's `X-B3-TraceId` or W3C Trace Context) to microservices. It can also start a trace span for the gateway itself. This allows performance analysis of how much time was spent in the router (auth check duration, forwarding duration) vs the service. Many Rust frameworks integrate with OpenTelemetry. The router can tag spans with information like route name, outcome (success/fail), etc. This is extremely helpful for debugging latency issues across microservices.

- **Metrics:** The router should collect metrics like request counts, error counts, and latency distributions. Examples: QPS (queries per second) per route, number of active WebSocket connections, number of 401/403 responses (auth failures), number of 5xx (failures), etc. Metrics can be exposed via an endpoint (Prometheus format, for instance). Specific metrics like **authZ latency** (time to get response from Sesame) help monitor the auth service's performance. Another metric could be **cache hit rate** for authZ cache. If using a library like `metrics` or `prometheus` in Rust, one can easily increment counters and record histograms around the code.

Having robust observability allows proactive management of the gateway. For example, if metrics show a spike in 429 rate-limited responses, maybe increase quotas or identify abuse. If tracing shows high latency specifically waiting on Sesame-IDAM, perhaps scale that service or cache more aggressively. Logging and metrics also facilitate **SLA enforcement and reliability improvement** [31] – one can set up alerts when error rates go beyond a threshold or when circuits open.

In practice, BRRTRouter can provide hooks or middleware for these concerns. The project roadmap already mentions adding **metrics and tracing hooks** [32]. Using those, one could plug in standard Rust libraries (e.g. `tracing` crate for structured logging/tracing, `opentelemetry` for trace export, and `prometheus` client for metrics). Given performance considerations, writing logs asynchronously (to avoid blocking the request flow) is recommended. Batching log writes or using an async logger will help at high volumes.

4. Integrating Sesame-IDAM for AuthZ in BRRTRouter

Sesame-IDAM is intended to serve as the authN/Z provider for the ecosystem. It provides authentication (user login, JWT issuance) and authorization features like role-based access control and API key

management [33] [26]. Integrating it with BRRTRouter involves treating Sesame as an **external Authorization Service**.

4.1 AuthZ API and Plugin Model

First, BRRTRouter needs a clear interface to talk to Sesame-IDAM. This could be a **REST API** call (e.g. `POST /authz/check` with a JSON containing token and resource) or a gRPC service (with a method like `CheckPermission(Token, Action)` returning allow/deny). Sesame-IDAM might also expose standard OAuth2 introspection endpoints or OPA-compatible policy APIs. For initial integration, assume a simple API: - Request: includes the **user's token or ID**, the **resource or route being accessed**, the **action** (read/write), and maybe contextual info (IP address, etc.). - Response: an **allow/deny** decision, possibly with details like allowed scopes or an error code for why denied.

BRRTRouter can abstract this behind a trait, e.g. an `AuthZService` trait with a method `fn authorize(&self, user_token: &str, route: &RouteInfo) -> Result<Decision, Error>`. A concrete implementation of this trait would call Sesame-IDAM (perhaps using an HTTP client). This abstraction allows plugging in a mock or alternate auth service if needed (useful for testing or if someone uses a different IDP). The call to authZ service should be made *non-blocking* (async/await or a coroutine yield) to not stall the router's event loop.

API assumptions: Sesame-IDAM likely uses JWTs for user sessions [26], so BRRTRouter might not need to send the whole JWT for introspection if the router can validate it. In a basic scenario, BRRTRouter could **verify the JWT's signature and expiry locally** (with Sesame's public key), then call Sesame for fine-grained authZ (passing the user ID or token ID and requested resource). This reduces load, as Sesame then only deals with authorization logic (like checking roles/permissions in its database). Alternatively, Sesame's API might accept the JWT and do both validation and authZ internally. Either design is workable.

One key decision is how tightly to couple Sesame with BRRTRouter. A **loosely coupled** approach treats Sesame like any external microservice – BRRTRouter calls it when needed. A **tighter integration** might involve Sesame pushing policy updates or tokens to the router. Given Sesame-IDAM is a standalone service (with its own database, client SDKs, etc.), loose coupling via HTTP/gRPC is likely best. BRRTRouter could use a local **client library** for Sesame if provided (for example, if Sesame has a Rust SDK to verify tokens or query permissions).

4.2 Failure Tolerance and Fallback

What if Sesame-IDAM is down or slow? This is critical, because if the authZ service is unavailable, the gateway might be unable to authorize requests. There are a few strategies to consider:

- **Fail Closed (Default):** The safest approach is to **deny all requests** when authZ cannot be confirmed. This prevents a situation where, during an auth service outage, protected data might be leaked. BRRTRouter would return 503 Service Unavailable or 500 error indicating auth service failure, or possibly a 403 (to not reveal the internal error). The downside is that an outage of Sesame-IDAM causes a full outage of the gateway's protected routes. If Sesame is highly available (clustered, replicated), this is acceptable.

- **Fail Open (Optional for certain routes):** In some scenarios, it might be acceptable to allow requests to proceed if the auth service can't be reached. This could be a policy choice for less sensitive endpoints or when partial functionality is preferred over none. For example, a read-only public data endpoint might be allowed if authZ times out (assuming the data isn't sensitive). Implementing fail-open is risky and generally not recommended for secure systems, but it could be an option toggled via configuration. If used, it should likely be combined with **auditing** – e.g. log all requests that were allowed due to missing authZ, so they can be reviewed.

- **Cached Decisions:** As mentioned in caching, if Sesame-IDAM is down, the router could rely on **recent cached authZ results** for a short time. E.g., "User123 was allowed to GET /data 5 minutes ago, so allow now if Sesame is unreachable." This provides some continuity. However, it's only feasible if you have such a cache and are willing to accept slightly stale decisions. For critical operations (e.g. financial transactions), this might not be appropriate; for less critical, it might be okay.

BRRTRouter should have a **timeout** on authZ calls (maybe ~100ms). If exceeded, treat it as failure. Combine this with circuit breaking: if Sesame-IDAM is consistently timing out, the router could assume it's down and either open a circuit (fail fast all incoming authZ until Sesame recovers) or switch to a backup strategy (if a secondary auth server is configured).

4.3 Security Considerations

Integrating Sesame-IDAM must be done with strong security practices:

- **Secure Connection:** Communication between BRRTRouter and Sesame-IDAM should be over TLS (even if within the same cluster). This prevents interception of tokens or auth data. If possible, use **mTLS (mutual TLS)** – the router and Sesame authenticate each other with certificates. This ensures only the legitimate router can query Sesame (preventing an attacker who somehow gets into the network from impersonating the router).

- **Authentication and Authorization of the Router:** Sesame-IDAM might require the router to authenticate when calling its APIs. For instance, the router could use a special service account token or client certificate. This prevents rogue services from querying Sesame's authZ API. Sesame could maintain an ACL that only BRRTRouter (identified by client ID or certificate CN) can call certain admin-level introspection queries.

- **Least Privilege for Tokens:** If BRRTRouter passes user JWTs to microservices, ensure those JWTs have appropriate scopes. If using token exchange, the delegated tokens to microservices should have limited scope/audience as discussed. Internally, the router might also include a header like `X-User-ID` for convenience, but relying on the token is better (services can independently verify it). Make sure not to forward any auth info to services that they shouldn't see – e.g. if the router uses an internal admin token to call Sesame, that should never go to end services.

- **Data Handling:** Sesame-IDAM will return sensitive info (user permissions, maybe group memberships). The router should treat this as sensitive: it likely doesn't need to log the full payload, just the decision. If it must log something for audit, ensure it's stored securely. Also, protect in-

memory data structures that store auth info – though in Rust, memory safety is given, still avoid storing sensitive info longer than necessary.

- **Version and Compatibility:** If Sesame-IDAM evolves (new API versions), design the integration to be adaptable. Possibly use feature flags or version negotiation when calling Sesame (e.g. include an `Accept-Version` header or use a versioned endpoint). This prevents a scenario where an update to Sesame breaks the router's expectations.

The **roadmap for BRRTRouter** explicitly calls out adding auth middleware for JWT/OAuth with integration to Sesame-IDAM [32]. This likely means the intention is to have BRRTRouter delegate most auth decisions to Sesame. A good approach is to keep BRRTRouter's own authorization code minimal (just forwarding requests and enforcing allow/deny responses). Sesame-IDAM itself, as seen in its features, supports RBAC and even API key limits [34], so it will be a natural place to centralize complex auth logic. BRRTRouter becomes a **policy enforcement point** (PEP) while Sesame-IDAM is the **policy decision point** (PDP), following standard security architecture.

4.4 Implementation Sketch

As a recommendation, implement BRRTRouter's authZ as **middleware** that wraps the proxy handlers:

- On router startup, configure an AuthZ client (with Sesame's address, credentials, etc.).
- For each incoming request, before routing to a backend, call `authz_client.check(token, route)`:
- If the response is allow, proceed. If deny, short-circuit with an error to client.
- If error contacting auth service, decide fail-closed or open based on config (with sensible default to closed). Possibly use cached decisions here if available.
- Optionally, if Sesame-IDAM also does authentication (validating username/password and issuing tokens), the router might not handle that at all – clients would first obtain a token from Sesame (via a login API) and then present the token to BRRTRouter. So BRRTRouter's focus is validating and authorizing the token's usage.

Security is an ongoing consideration – things like **audit logs**, **IP allow/deny lists**, and **content security (e.g. scanning uploads)** might also come into play, but they can be layered similarly (e.g. another middleware for content scanning).

5. Code Structure and Examples

Finally, let's discuss how BRRTRouter's Rust implementation might be structured to achieve the above. The goal is to maintain high performance (leveraging Rust's zero-cost abstractions and possibly the coroutine runtime) while keeping the code modular for different protocols and concerns.

5.1 Overall Architecture

BRRTRouter can be organized into distinct modules/layers, for example:

- **Router Core:** Responsible for loading the OpenAPI spec and building the routing table (mapping paths + methods to handlers or upstream targets). This likely uses the compiled regex matchers and the spec definitions [35] [36]. It should be agnostic of auth – the routing table just knows where to send a request or which handler function to call.

- **Dispatcher/Handlers:** The router supports registering handlers for routes [37]. In a proxy scenario, these "handlers" will typically perform the forwarding to microservices. Instead of business logic, a handler might be a small piece that calls `proxy_http(request, service_address)` or `proxy_grpc(request, stub)` depending on route config. The existing example uses an `echo_handler` for testing [38] – in real use, one would implement a `proxy_handler`. The dispatcher, running on a coroutine, takes incoming requests and hands them to the appropriate handler function.

- **Auth Middleware:** Rather than duplicating auth checks in every handler, it's cleanest to have a middleware layer. This could be implemented by wrapping the dispatcher or as a pre-processing step when a request arrives. For instance, one could modify the dispatcher to first call `authz_check(request)` before invoking the actual route handler. If using Tower, one could create a Tower Service that does auth then delegates. Since BRRTRouter is custom, a simple approach is:

```
fn handle_request(req: Request) -> Response {
    // Authorization middleware
    if !authorize_request(&req) {
        return Response::forbidden("Unauthorized");
    }
    // Call the actual route handler (which may be a proxy forward)
    dispatch_to_handler(req)
}
```

The `authorize_request` function would extract the token and call Sesame-IDAM (perhaps asynchronously). In a coroutine context, it might perform a blocking call on a separate thread or use async if the runtime permits (if `may` doesn't do async await, one might use its blocking I/O primitives or spawn a small threadpool for outbound HTTP calls).

- **Protocol-specific components:** There might be separate sub-modules for HTTP, gRPC, and WebSocket handling:
- An **HTTP server** (perhaps using `may_minihttp` as noted [39]) to accept HTTP connections. It would parse HTTP, then use the router to dispatch.
- A **gRPC server** – possibly implemented with Tonic or the router's own HTTP/2 handling plus protobuf decoding. If using Tonic, you can generate service trait implementations that call into the router's logic. Alternatively, treat gRPC as just HTTP/2 with a special content type and route those like any other request (though usually you'd need to forward at byte level to preserve things like streaming easily).

- A **WebSocket upgrader** – e.g., using an HTTP upgrade mechanism. Once upgraded, the code managing the WebSocket can be separate from the main request handling loop. Likely a new task/coroutine is spawned to handle each WS connection (reading from client and writing to backend and vice versa).

- **Clients for microservices:** The router will likely include an HTTP client for proxying. In Rust, one could use `hyper::Client` or `reqwest` for simplicity. However, using hyper directly with a custom runtime might require integration (maybe compile hyper on Tokio while the router uses May – that could be complicated). Another approach is to use a simpler HTTP client that can be driven synchronously or with May. There are crates like `ureq` (pure blocking HTTP) that could work inside a coroutine (since coroutines in May are like lightweight threads, blocking IO might be okay as long as it yields – need to confirm integration). For gRPC, one could use Tonic's client which is async (Tokio) – integration might require running a separate async runtime. It may be easier to treat gRPC in a simpler way: if the router doesn't need to inspect messages, it could act as a byte-level proxy – essentially just pipe the HTTP/2 frames. Envoy does this with its generic proxy, but implementing that from scratch is non-trivial. Using Tonic might ultimately be simpler if combining runtimes is feasible (Tokio inside or alongside May, or just using Tokio for both server and client and foregoing May for gRPC part).

- **Async vs Coroutine:** Since BRRTRouter uses the **may** library (which provides Go-like coroutines), the code might look slightly different than typical `async/await`. Coroutines allow writing code in synchronous style that yields at I/O points. For example:

```rust
fn proxy_request(req: HttpRequest, backend_addr: &str) -> HttpResponse {
    // Pseudocode using blocking calls that actually yield under the hood
    let mut stream = TcpStream::connect(backend_addr).unwrap();
    stream.set_write_timeout(Some(REQUEST_TIMEOUT)).unwrap();
    // write request
    stream.write_all(req.serialize()).unwrap();
    // read response
    let resp = HttpResponse::parse_from(&mut stream).unwrap();
    resp
}
```

In a coroutine, `TcpStream::connect` and `write_all` would not block the OS thread, they would yield to other coroutines while waiting. This style is similar to synchronous code, which can be easier to reason about. The advantage for BRRTRouter is high throughput with less overhead than spawning OS threads. The code above would be part of the handler for a route.

5.2 Example: HTTP Proxy Handler with Auth

Below is a conceptual snippet (simplified for clarity) of how a handler might be structured in BRRTRouter to integrate authZ and proxy logic:

```rust
use brrrouter::http::{Request, Response};  // hypothetical modules
use brrrouter::auth::AuthZClient;          // Sesame-IDAM client interface

fn proxy_handler(req: Request, backend_url: &str, authz: &AuthZClient) ->
Response {
    // 1. Authorization middleware
    if let Some(token) = req.headers().get("Authorization") {
        let allowed = authz.check(token, req.method(), req.path());
        if !allowed {
            return Response::forbidden();  // 403 Forbidden
        }
    } else {
        return Response::unauthorized("Missing auth token");
    }

    // 2. Prepare outbound request
    let mut out_req = req.clone_without_body();  // clone headers and meta
    out_req.set_uri(format!("http://{}{}", backend_url,
req.uri().path_and_query()));
    // (Alternatively, backend_url could include path prefix adjustments)

    // 3. Stream request body to backend and get response
    match http_client::send(out_req, req.body()) {
        Ok(mut out_resp) => {
            // 4. Build response to client
            let status = out_resp.status();
            let headers = filter_headers(out_resp.headers());  // drop hop-by-
hop headers, etc.
            let body_stream = out_resp.into_body();
            Response::from_parts(status, headers, body_stream)
        }
        Err(e) => {
            // Map error to appropriate response
            if e.is_timeout() {
                Response::gateway_timeout("Upstream timed out")
            } else {
                Response::bad_gateway("Upstream error")
            }
        }
    }
}
```

In this pseudo-code: - `AuthZClient::check` would synchronously (or async in real impl) call Sesame-IDAM. It could return a bool or perhaps a Result with more info. The request method and path are passed so the auth service can apply route-specific rules. - We build a new `out_req` directed at the backend service. The `backend_url` could be determined by the router's route configuration (maybe stored in the

OpenAPI spec via an extension like `x-backend-url`). We ensure the URI is set to the microservice and copy necessary headers. We remove any headers that should not be forwarded (e.g. `Authorization` might or might not be forwarded depending on whether the backend uses the same token – if using token exchange, we'd replace it with the exchanged token). - `http_client::send` is a placeholder for an HTTP client call that takes a request (and possibly a body stream) and returns a response. In an async context, this would be `.await`ed. - The response from the microservice is then translated to the client. We filter headers (e.g. strip out `Transfer-Encoding: chunked` if we'll re-chunk it, remove any hop-by-hop like `Connection` or internal headers). Then we create a Response with the same status and a body that is essentially a stream from `out_resp`. In a well-designed system, we can **pipe the body stream directly** to the client without reading it fully (this might involve using an `async_stream` or just returning a `Body` that is backed by the upstream response body). This way, large payloads are streamed.

For gRPC, the handler would be similar but using the gRPC stubs: - Use an interceptor to do auth (as shown earlier using Tonic's `with_interceptor` or manual check). - The handler for `MethodX` would call `backend_stub.method_x(request).await` and return the result. Tonic takes care of streaming and error mapping. The router mainly adds the authZ on top. One might structure the gRPC service implementation to use a generic forwarding mechanism (perhaps the microservice address is known from a mapping).

5.3 Handler Registration and Dynamic Dispatch

BRRTRouter's design mentions dynamic handler dispatch driven by the OpenAPI spec [40]. This means when we load the spec, we might attach each operation (GET/POST on a path) to either a function or an upstream target. Code generation could create a static match of routes to handler functions. For example, an OpenAPI extension `x-handler-proxy: "inventory_service"` could indicate that route should be proxied to the inventory service. The startup code would then register that:

```
dispatcher.register_handler("getInventory", proxy_to_inventory_handler);
```

Where `proxy_to_inventory_handler` could be a closure or function that captures the target service's address. Alternatively, the dispatcher could store a mapping of route -> service name, and use a generic handler that looks up the service address and forwards. Pseudocode:

```
let service_map = { "/api/v1/inventory" => "inventory.internal:5000", ... };
...
dispatcher.set_default_handler(|req| {
    let target = service_map.get(req.path()).expect("configured");
    proxy_handler(req, target, &authz_client)
});
```

This uses one generic handler for all proxied routes (looking up the target). If some routes require custom logic, those can still have specialized handlers.

Hot reloading the OpenAPI spec (a planned feature) [41] also plays in: if the spec changes (e.g. new route or different backend), the router might update the routing table on the fly. The code structure should separate the long-running server from the data that can be updated. Perhaps the router holds an `Arc<RouterState>` that can be swapped. Handlers might need to use locks or channels to update their target info safely. This is advanced, but worth noting for code design (use of `RwLock` or lock-free maps for route configs).

5.4 Using Existing Libraries vs Custom Code

Where possible, leverage robust crates: - **Hyper** for HTTP client/server: Hyper is battle-tested and high-performance. BRRTRouter could use Hyper for both listening to HTTP and making outbound requests. The challenge is that Hyper is async/Tokio-based. If the project is already using the **May** coroutine runtime, mixing the two might require some adaptation (maybe running a Tokio runtime on top of a thread – doable but adds complexity). Another crate, like `tiny_http` or the built-in `std::net` with threads, might not give the needed async performance. Given the performance goals (millions of req/sec), using a highly optimized library like Hyper (which uses all sorts of tricks) is prudent. It might be worth exploring a **native coroutine integration** with hyper's futures (for example, using `futures::executor` inside a May coroutine). - **Tonic** for gRPC: Tonic is the go-to Rust gRPC library. It would simplify implementing both the server (to receive client calls) and the client (to forward to microservices). If using Tonic, one might end up running an async runtime (Tokio) for it. It is possible to run multiple runtimes: e.g., let BRRTRouter's main run on May for HTTP, and spawn a Tokio runtime for gRPC tasks. They can communicate via channels if needed. This is complex but manageable. Alternatively, use only Tokio for everything (Hyper + Tonic) and drop the custom coroutine part – but the project seems to be specifically pursuing the coroutine approach for performance and perhaps simplicity of code. Another approach: since gRPC is basically HTTP/2 with Protobuf, one could attempt a minimal implementation. But supporting all gRPC features (flow control, streaming, etc.) is a lot of work – likely not worth redoing when Tonic exists.

- **WebSocket**: There are libraries like `tungstenite` (synchronous) and `tokio-tungstenite` (async) that handle WebSocket protocol framing. BRRTRouter could use tungstenite in coroutine mode – it has a feature for blocking contexts. That could integrate well with May. Essentially, accept the TCP, do the upgrade handshake (tungstenite can do server handshake given the HTTP headers), then yield to an event loop that shuffles messages. Or use Hyper's upgrade feature and then tungstenite for framing. Code structure wise, after upgrading, spawn two coroutines: one reading from client and writing to server, another reading from server and writing to client.

5.5 Example: Simplified Code for Auth + Proxy in Handler

To tie it together, here's a highly simplified, pseudo-Rust snippet demonstrating how an incoming HTTP request might be handled with all pieces (error handling omitted for brevity):

```
fn on_http_request(req: HttpRequest) -> HttpResponse {
    // 0. Route matching (done prior to this, assume we have route info)
    let route = router.match(req.path(), req.method());
    let backend = route.backend;  // e.g. "inventory.internal:8080"
    let auth_required = route.auth;  // e.g. true/false or required scopes

    // 1. Authentication & Authorization
```

```
    if auth_required {
        let token = req.headers().get("Authorization")
            .and_then(|h| h.to_str().ok());
        if token.is_none() {
            return HttpResponse::unauthorized("Missing token");
        }
        let decision = authz_client.check(token.unwrap(), route.id);
        match decision {
            AuthDecision::Allow => { /* ok */ },
            AuthDecision::Deny => {
                return HttpResponse::forbidden("Access denied");
            },
            AuthDecision::Error => {
                // Could decide to fail open/closed; here closed
                return HttpResponse::service_unavailable("Auth service error");
            }
        }
    }

    // 2. Prepare request to backend
    let mut out_req = req.into_outbound(backend);
    // (copy method, uri, headers except few, body stream setup)
    // e.g., adjust host header:
    // out_req.headers().insert("Host", backend.host());
    // Remove `Authorization` if not needed by backend or replace if using
delegation.

    // 3. Forward to backend and wait for response
    let out_resp_result = http_client.execute(out_req);

    // 4. Process response or error
    match out_resp_result {
        Ok(out_resp) => {
            // Build response to client
            let mut resp_builder = HttpResponse::build(out_resp.status());
            copy_headers_except(&out_resp.headers(), &mut resp_builder,
["Connection", ...]);
            // The body can be a stream, we will pipe it directly.
            resp_builder.stream_body(out_resp.into_body());
            resp_builder.finish()
        }
        Err(err) => {
            if err.is_timeout() {
                HttpResponse::gateway_timeout("Upstream timeout")
            } else {
                HttpResponse::bad_gateway("Upstream error")
            }
        }
```

```
    }
}
```

In reality, the code will be more complex (especially handling streaming bodies asynchronously). But this shows the **high-level flow in code**: check auth, then forward. The separation of concerns means our proxy logic doesn't need to know *why* a request might be forbidden – that's handled by the auth client.

5.6 Concluding Thoughts on Code Design

Maintaining a **clean separation** between the routing, auth, and proxy logic will make BRRTRouter easier to extend. For instance, if one wanted to swap Sesame-IDAM for another system or even use an embedded policy engine, you'd implement the same `AuthZClient` trait accordingly. Or if a new protocol (say MQTT or another async API) needs to be proxied, one could add a module for it without tangling with core HTTP code.

Rust's type system can be leveraged to ensure, for example, that an unauthenticated request cannot accidentally hit the proxy logic – by perhaps requiring an `AuthContext` that is produced by the auth middleware and passed to handlers. But that might be overkill for a gateway, where we simply enforce order of operations.

The project's ambitions like **hot reload** and **dynamic spec updates** [40] mean the code might use channels or watchers to update the routing table at runtime. Using a thread-safe structure (Arc<RwLock<RouterState>>) for the routes and service mappings is one approach.

Testing is also important. Code could be structured such that the proxy logic can be tested with a dummy backend (for example, using a local HTTP server in tests, or using dependency injection to provide a fake `http_client`). Similarly, the auth client can be mocked to test allow/deny flows. Unit tests for each piece (auth, route matching, retry logic, etc.) plus integration tests with multiple concurrent requests will help ensure the router works as expected under load.

References:

- BRRTRouter documentation and roadmap for planned features [32].
- Sesame-IDAM description and performance goals [1] [42].
- Permit.io blog on JWT vs opaque token trade-offs [6] [23].
- OPA usage in microservices (InfraCloud blog) [2] [13].
- Medium article on OAuth2 Token Exchange for delegation [17] [18].
- API gateway best practices (rate limiting, logging, etc.) [25] [31].
- Statsig note on gateway error codes [4].
- Circuit breaker pattern description [24].

---

[1] [26] [28] [33] [34] [42] GitHub - microscaler/sesame-idam: simple identity and access management system
https://github.com/microscaler/sesame-idam

2 13 15 How to Implement Microservices Authorization with OPA
https://www.infracloud.io/blogs/opa-microservices-authorization-simplified/

3 29 30 32 35 36 37 38 39 40 41 GitHub - microscaler/BRRTRouter: BRRTRouter is a high-performance, coroutine-powered request router for Rust, driven entirely by an OpenAPI 3.1.0 Specification
https://github.com/microscaler/BRRTRouter

4 502 vs. 504 errors: What's the difference? - Statsig
https://www.statsig.com/perspectives/502-vs-504-errors-difference

5 6 7 8 9 10 11 23 A Guide to Bearer Tokens: JWT vs. Opaque Tokens
https://www.permit.io/blog/a-guide-to-bearer-tokens-jwt-vs-opaque-tokens

12 Envoy External Authorization with Golang GRPC service
https://dev.to/prakash_chokalingam/envoy-external-authorization-with-golang-grpc-service-58h8

14 Best Practices for Microservice Authorization - Permit.io
https://www.permit.io/blog/best-practices-for-authorization-in-microservices

16 17 18 19 20 OAuth2 Token Exchange in Practice | by Sagara Gunathunga | Medium
https://sagarag.medium.com/oauth2-token-exchange-in-practice-5a12a6d2e0d

21 OAuth2 Token Exchange RFC8693 - Tyk API Management
https://tyk.io/blog/res-oauth2-token-exchange-rfc8693/

22 Identity Propagation in an API Gateway Architecture - Google Cloud
https://cloud.google.com/blog/products/api-management/identity-propagation-in-an-api-gateway-architecture

24 24. Implementing Circuit Breaker Pattern in API Gateway - Medium
https://medium.com/@ilakk2023/implementing-circuit-breaker-pattern-in-api-gateway-552a1c9a3780

25 What Is an API Gateway? Core Concepts & Benefits - API7.ai
https://api7.ai/learning-center/api-gateway-guide/what-is-an-api-gateway

27 What is Rate Limiting? Meaning and Definition - Wallarm
https://www.wallarm.com/what/rate-limiting

31 Building Reliable API Gateways with Logging and Monitoring - API7.ai
https://api7.ai/learning-center/api-gateway-guide/api-gateway-logging-monitoring-best-practices