**The plan for the building this system**

**Step 1: understanding**

Once I thoroughly read the business case requirement and specification, I will summarize the main points to myself and explain to myself in my own way the TASK.

My notes:

I have to come up with a link redirect mechanism that allows me to cloak params.

With the following primary goals:

* Encapsulation: to hide original traffic source params from the affiliate network
* Consistency: the input should be the same our\_param unless explicitly refreshed
* Reversibility: we must have the capability to retrieve original param from our\_param
* Refreshability: Should allow forcing a new our\_param for an existing triplet
* Performance: to handle 1 million requests per day and prepare for scaling.

**Step 2: R&D:**

Once I understand the task, I will start to research possible solution that will fit.

As is the case for most thing I will divide the solution into 3 categories:

* Ready made solution that can be purchased or has a limited licence
* Open-source solution (MIT licence or limited)
* Implement the solution

My notes:

Binom - is a direct to task, out-of-the-box functionality and scalability, it is robust for high traffic environments and offers advanced analytics and speedy redirects

Shlink - has an open-source that offers more control and opportunities for customizations with a good initial structure.

Both you can customise but does not meet all of the requirements

I still do not have a global background to this case such as the existing infrastructure and tools and also consideration that this may be done in a short time period, I can only choose to Implement the solution.

**Step 3: Architecture and DD:**

Once I decided Implement the task I will jump into the more technical details

The Core Technology Stack

* Programming Language: TypeScript for maintaining much better code quality and reducing error or bugs
* Runtime Environment: Node.js because of its asynchronous and non-blocking I/O model, and because of its large NPM ecosystem for existing libraries to use
* Web Framework: Express.js - minimal, flexible, large community
* Data Storage: Can use Redis - an in-memory data structure store, used as a database, cache. Speed, Data Structures, Persistence and Scalability (can be clustered)
* Docker for consistent environments and easier deployment
* Logging: Winston - A very flexible logging library for Node.js, allowing multiple transports (console, file, etc.) and log levels
* Unique ID Generation: nanoid - For generating short, unique, URL-friendly ids for our\_param, UUID's are an option but be longer than required

**System Components**

1. **Entry Point (src/server.ts):**
   * Initializes the HTTP server.
   * Initializes and manages the Redis client instance and its lifecycle.
   * Instantiates the main Express application (via createApp from src/app.ts).
   * Sets up process event listeners for graceful shutdown (potentially via a shutdown.ts module).
   * Starts the HTTP server to listen for incoming requests.
2. **Application Setup (src/app.ts):**
   * Exports a createApp function that takes the redisClient as a dependency.
   * Creates and configures the main Express application instance.
   * Applies global middleware:
     + Body parsers (express.json, express.urlencoded).
     + Global rate limiting.
     + Request logging (requestLogger).
   * Mounts the main application router (from src/routes/index.ts).
   * Defines root path handler, 404 handlers, and the global error handler (errorHandler).
   * Returns the configured Express app instance.
3. **Routing Layer (src/routes/):**
   * Main Router (src/routes/index.ts):
     + Exports a createAppRouter function that takes the redisClient as a dependency.
     + Initializes and mounts specific feature routers (e.g., traffic.routes.ts).
     + Defines a health check endpoint (/health).
   * Traffic Routes (src/routes/traffic.routes.ts):
     + Exports a createTrafficRouter function that takes the redisClient.
     + Instantiates the ParameterMappingService (passing it the redisClient).
     + Instantiates the TrafficController (passing it the parameterMappingService).
     + Defines API routes (/redirect, /refresh, /retrieve\_original).
     + Applies route-specific middleware (e.g., validateTripletParams, specific rate limiters like refreshLimiter).
     + Delegates request handling to methods of the TrafficController.
4. **Controller Layer (src/controllers/traffic.controller.ts):**
   * Class-based (TrafficController).
   * Receives the ParameterMappingService instance via constructor dependency injection.
   * Contains methods (handleRedirect, handleRefresh, handleRetrieveOriginal) that correspond to API endpoints.
   * Responsible for:
     + Extracting relevant data from the Express Request object (often aided by validation middleware).
     + Calling appropriate methods on the ParameterMappingService to perform business logic.
     + Constructing the Express Response (status codes, JSON bodies, redirects).
     + Calling next(error) to pass errors to the global error handler.
5. **Service Layer (src/services/ParameterMappingService.ts):**
   * Class-based (ParameterMappingService).
   * Receives the redisClient instance via constructor dependency injection.
   * Encapsulates the core business logic:
     + Generating tripletId from (keyword, src, creative).
     + Generating unique our\_param values.
     + Interacting with Redis (via AdvancedRetrySystem) to store and retrieve mappings (ttp: and ptk: keys).
     + Handling the refresh mechanism.
   * Throws custom errors (NotFoundError, BadRequestError, InternalServerError) for issues it encounters, or allows RetryError/CircuitBreakerOpenError from the resilience layer to propagate.
6. **Resilience Library (src/lib/resilience/):**
   * AdvancedRetrySystem.ts: Orchestrates retry and circuit breaker.
   * ExponentialBackoffRetry.ts: Implements retry logic.
   * CircuitBreaker.ts: Implements circuit breaker logic.
   * errors.ts: Defines custom error classes used throughout the application.
7. **Middleware (src/middleware/):**
   * requestLogger.ts: Logs incoming requests and their outcomes.
   * errorHandler.ts: Centralized error handling for all API routes.
   * validateTripletParams.ts: Validates the presence and basic format of keyword, src, creative parameters.
   * Rate limiting middleware (configured using express-rate-limit in app.ts and traffic.routes.ts).
8. **Utilities (src/utils/):**
   * logger.ts: Configures and exports the Winston logger instance.
9. **Configuration (src/config/index.ts):**
   * Loads environment variables using dotenv.
   * Defines and exports a typed configuration object (AppConfig) providing defaults for all settings (ports, URLs, Redis settings, rate limits, resilience parameters, etc.).
10. **Data Storage (Redis):**
    * Stores two primary types of mappings using specific key prefixes (ttp:, ptk:).
    * Accessed via the ioredis client, wrapped with retry/circuit breaker logic in the ParameterMappingService.

**Request Flow (Redirect)**

1. **Incoming Request:** A client (e.g., user's browser, another service) sends a GET request to the server:  
   GET https://ourserver.com/api/v1/redirect?keyword=shoes&src=google&creative=1234
2. **Express Application (src/app.ts):**
   * The request hits the Express application.
   * Global middleware runs:
     + express.json() / express.urlencoded() (not strictly needed for this GET, but generally present).
     + Global API rate limiter (apiLimiter in app.ts) checks the request against defined limits. If exceeded, a 429 response is sent.
     + requestLogger logs the incoming request details.
3. **Routing (src/routes/index.ts -> src/routes/traffic.routes.ts):**
   * The main router (src/routes/index.ts) directs the request (based on /api/v1 prefix) to the trafficRouter.
   * The trafficRouter (src/routes/traffic.routes.ts) matches the /redirect path.
4. **Route-Specific Middleware (src/middleware/validateTripletParams.ts):**
   * The validateTripletParams middleware executes.
   * It extracts keyword, src, and creative from req.query.
   * It validates their presence and basic format.
     + If validation fails, it creates a BadRequestError and calls next(error), short-circuiting to the global errorHandler.
   * If validation succeeds, it attaches the validated parameters (e.g., as req.tripletParams) to the Request object and calls next().
5. **Controller (src/controllers/traffic.controller.ts - handleRedirect method):**
   * The trafficRouter delegates the request to the TrafficController.handleRedirect method.
   * The controller method retrieves keyword, src, creative (e.g., from req.tripletParams).
   * It calls this.parameterMappingService.getOrGenerateOurParam({ keyword, src, creative }).
6. **Service Layer (src/services/ParameterMappingService.ts - getOrGenerateOurParam method):**
   * This method is wrapped by the AdvancedRetrySystem for resilience against Redis issues.
   * **a. Generate Triplet ID:** Creates a canonical, hashed representation of the (keyword, src, creative) triplet (e.g., SHA256(keyword:shoes|src:google|creative:1234)). This is the tripletId.
   * **b. Query Redis (via AdvancedRetrySystem -> ioredis):**
     + Attempts GET ttp:<tripletId> from Redis.
   * **c. If our\_param exists in Redis:**
     + Logs that an existing parameter was found.
     + Returns the retrieved our\_param\_value.
   * **d. If our\_param does not exist (new combination):**
     + Logs that a new parameter is being generated.
     + Generates a new unique new\_our\_param\_value (using nanoid).
     + Constructs the JSON string for the original triplet: {"keyword":"shoes","src":"google","creative":"1234"}.
     + Uses a Redis pipeline (again, via AdvancedRetrySystem -> ioredis) to atomically:
       - SET ttp:<tripletId> <new\_our\_param\_value>
       - SET ptk:<new\_our\_param\_value> '{"keyword":"shoes","src":"google","creative":"1234"}'
     + Logs the successful storage.
     + Returns the new\_our\_param\_value.
   * **e. Error Handling:** If any Redis operation fails after retries or the circuit breaker is open, an error (e.g., RetryError, CircuitBreakerOpenError, or InternalServerError if pipeline fails) is thrown.
7. **Controller (Resuming handleRedirect):**
   * **If getOrGenerateOurParam returns successfully:**
     + The controller receives the our\_param\_value.
     + It constructs the affiliate redirect URL using appConfig.affiliateBaseUrl and the our\_param\_value: https://affiliate-network.com?our\_param=<our\_param\_value>.
     + Logs the redirect action.
     + It calls res.redirect(302, redirectUrl.toString()).
   * **If getOrGenerateOurParam throws an error:**
     + The catch block in handleRedirect catches the error.
     + It logs the error.
     + It calls next(error), passing the error to the global error handler.
8. **Express Response / Error Handling:**
   * **Successful Redirect:** Express sends a 302 Found HTTP response to the client with the Location header set to the affiliate URL. The requestLogger logs the 302 status.
   * **Error Occurred:** If next(error) was called at any stage:
     + The global errorHandler (src/middleware/errorHandler.ts) middleware executes.
     + It logs the error in detail.
     + It determines an appropriate HTTP status code (e.g., 400, 500, 503).
     + It sends a JSON error response to the client (e.g., {"status":"error", "statusCode":503, "message":"Service temporarily unavailable."}). The requestLogger logs this error status code.

**Data Structures and Storage Mechanism (Redis)**

The system leverages Redis as a high-throughput, in-memory key-value store for persisting and managing the mappings between traffic source parameters and the internally generated our\_param. All interactions with Redis are performed via the ioredis client, typically wrapped within the AdvancedRetrySystem for resilience, from the ParameterMappingService.

**Two primary key structures are used in Redis:**

1. Triplet ID to our\_param Mapping (Forward Mapping):
   * Purpose: To quickly and consistently retrieve the *currently active* our\_param for a given combination of (keyword, src, creative). This is essential for generating redirect links and for the refresh mechanism.
   * Redis Key Pattern: ttp:<tripletId>
     + ttp:: A static prefix denoting "Triplet To Parameter." (Defined as TTP\_PREFIX in ParameterMappingService).
     + <tripletId>: A deterministic, case-insensitive SHA256 hash generated from a canonical string representation of the keyword, src, and creative values (e.g., keyword:value|src:value|creative:value). This ensures that the same input triplet always produces the same tripletId.
   * Redis Value: A string representing the currently active our\_param (e.g., abc123xyz).
   * Redis Data Type: String.
   * Primary Redis Commands Used: GET, SET (often within a PIPELINE).
2. our\_param to Original Triplet Data Mapping (Reverse Mapping):
   * Purpose: To retrieve the original (keyword, src, creative) values when given an our\_param. This is crucial for the /retrieve\_original API endpoint.
   * Redis Key Pattern: ptk:<our\_param\_value>
     + ptk:: A static prefix denoting "Parameter To Key (details)." (Defined as PTK\_PREFIX in ParameterMappingService).
     + <our\_param\_value>: The actual unique our\_param string (e.g., abc123xyz) generated by nanoid.
   * Redis Value: A JSON string containing the original traffic parameters. The structure is defined by the StoredTripletData interface (which is TrafficParams in ParameterMappingService.ts).  
     Example: {"keyword":"shoes","src":"google","creative":"1234"}
   * Redis Data Type: String (storing JSON).
   * Primary Redis Commands Used: GET, SET (often within a PIPELINE).

**Atomicity:**When creating or updating mappings (especially during the initial generation for a new triplet or during a refresh), both the forward (ttp:) and reverse (ptk:) mappings are set within a single Redis PIPELINE. This ensures that these related operations are sent to Redis as a batch, reducing network overhead. While PIPELINE doesn't offer true transactional atomicity in the same way MULTI/EXEC , it guarantees that the commands are executed sequentially without interruption from other clients, and ioredis allows checking for errors in the pipeline results. For this use case, pipelining provides sufficient consistency and performance. If strict atomicity (all or nothing) were paramount, MULTI/EXEC could be used, though it adds slight complexity.

**Data Model (Conceptual):**

* TrafficParams interface: { keyword: string; src: string; creative: string; }
* StoredTripletData interface (same as TrafficParams): Defines the structure of the JSON stored in the ptk: mapping.

**Methods for Ensuring Consistency and Reversibility**

The system employs several strategies to maintain consistency and ensure that our\_param values can be reliably reversed to their original traffic source parameters.

Consistency:

* Deterministic tripletId Generation:
  + The \_generateTripletId method in ParameterMappingService creates a canonical string from the keyword, src, and creative parameters (e.g., converting to lowercase, specific ordering of fields like keyword:${k}|src:${s}|creative:${c}).
  + This canonical string is then hashed using SHA256. This process ensures that the exact same set of input parameters will always produce the identical tripletId.
* Authoritative Forward Mapping (ttp:):
  + The ttp:<tripletId> key in Redis acts as the single source of truth for the *current* our\_param associated with a specific triplet.
  + When a request comes in for a triplet, the system first checks this mapping.
    - If found, the existing our\_param is used, ensuring consistent our\_param assignment for that triplet until a refresh occurs.
    - If not found, a new our\_param is generated and this mapping is created.
* Unique our\_param Generation:
  + nanoid is used to generate short, unique, URL-friendly strings for our\_param. The probability of collision is extremely low for a reasonable length. This ensures that each new mapping (or refreshed mapping) gets a distinct identifier for the ptk: key.

**Reversibility:**

* **Dedicated Reverse Mapping (ptk:):**
  + **Whenever a new our\_param is generated (either for a new triplet or during a refresh), a corresponding reverse mapping is created: ptk:<new\_our\_param\_value> -> JSON\_string\_of\_original\_triplet.**
  + **This ptk: mapping directly links an our\_param back to the StoredTripletData (keyword, src, creative) it was generated for.**
* **Data Integrity in Reverse Mapping:**
  + **The original keyword, src, and creative values (as received, after initial validation) are stored in the JSON string. No transformations are applied that would prevent retrieving the exact original (case-preserved, if applicable by requirements, though current implementation normalizes for tripletId generation).**
* **Persistence of Old our\_param Mappings:**
  + **When an our\_param is refreshed for a triplet, the ttp:<tripletId> mapping is updated to point to the *new* our\_param.**
  + **However, the *old* ptk:<old\_our\_param\_value> mapping (linking the previous our\_param to the original triplet) is not deleted by default.**
  + **This means that if an old link containing the old\_our\_param\_value is used with the /retrieve\_original API, it will still correctly resolve to the original triplet data it was associated with at the time of its generation. This provides historical reversibility for previously issued our\_param values.**

**Handling the Refresh Mechanism**

The system provides an API endpoint (POST /api/v1/refresh) to force the assignment of a *new* our\_param value to an existing (keyword, src, creative) combination. This makes the triplet appear as a "new" entity to the affiliate network when subsequent redirect links are generated.

**Refresh Flow:**

1. API Request: A POST request is made to /refresh with the keyword, src, and creative in the request body.
2. Validation: The validateTripletParams middleware and controller-level checks ensure the parameters are valid.
3. Controller (TrafficController.handleRefresh):
   * Delegates the request to ParameterMappingService.refreshOurParam(params).
4. Service (ParameterMappingService.refreshOurParam):
   * This method is wrapped by the AdvancedRetrySystem.
   * a. Generate tripletId: The tripletId is generated from the provided (keyword, src, creative) parameters, same as in the redirect flow.
   * b. (Optional Log) Retrieve Old our\_param: The service may optionally GET ttp:<tripletId> to log the old\_our\_param\_value if it exists, for audit purposes.
   * c. Generate New our\_param: A *new*, unique new\_our\_param\_value is generated using nanoid.
   * d. Update/Create Mappings (Atomic Batch via Redis Pipeline):
     1. Update Forward Mapping: SET ttp:<tripletId> <new\_our\_param\_value>
        + This command overwrites any existing our\_param associated with this tripletId, effectively making <new\_our\_param\_value> the active parameter for this triplet.
     2. Create New Reverse Mapping: SET ptk:<new\_our\_param\_value> '{"keyword":"...","src":"...","creative":"..."}'
        + This creates the reverse lookup entry for the newly generated new\_our\_param\_value.
   * e. Logging: The refresh event, including the old (if any) and new our\_param values and the triplet details, is logged.
   * f. Return New our\_param: The new\_our\_param\_value is returned.
5. Controller Response: The TrafficController sends a JSON response to the client, typically including a success message and the new\_our\_param\_value.

**Consistency After Refresh:**

* New Redirects Use New our\_param: Any subsequent request to /redirect for the same (keyword, src, creative) triplet will now use the ttp:<tripletId> mapping, which points to the new\_our\_param\_value. Thus, the affiliate link will contain this new parameter.
* Old our\_param Still Reversible: As mentioned in section 4.2, the ptk:<old\_our\_param\_value> mapping for the *previously active* our\_param is typically preserved. This means:
  + If a third party or system still holds a link with the old\_our\_param\_value, an API call to /retrieve\_original?our\_param=<old\_our\_param\_value> will still successfully return the original (keyword, src, creative) data.
  + This behavior is generally desirable for data integrity and historical tracking, but could be changed (e.g., by deleting the old ptk: entry during refresh) if business rules required old our\_param values to become immediately invalid for retrieval. The current implementation prioritizes full historical reversibility.
* No Data Loss: The original (keyword, src, creative) data itself is not lost or altered; only its association with the *active* our\_param changes for new link generation.

**Security Considerations**

* **Input Validation:**

Strictly validate keyword, src, creative, and our\_param for type, length, and potentially allowed characters to prevent injection attacks or malformed data.

**HTTPS:** The entire system should operate over HTTPS to protect data in transit.

* **Rate Limiting:** Implement rate limiting on all endpoints (especially /redirect and /refresh) to prevent abuse and DoS attacks. express-rate-limit is a good choice.
* **Redis Security:**
  + Configure Redis with a strong password (requirepass).
  + Bind Redis to localhost or a private network interface if the Node.js app and Redis are on the same machine or within a trusted network.
  + If Redis is exposed, use firewalls to restrict access.
* **Endpoint Protection for /refresh and /retrieve\_original:** These endpoints might need authentication/authorization (e.g., API key, IP whitelisting) if they are not intended for public use, to prevent unauthorized data manipulation or retrieval. The current problem description doesn't specify, but in a real-world scenario, this would be crucial.
* **Sensitive Data:** Ensure no overly sensitive data is inadvertently logged.
* **Dependency Security:** Regularly scan NPM dependencies for vulnerabilities .

**Performance Optimizations**

* **Redis Connection Pooling:** Use a Redis client library that supports connection pooling (e.g., ioredis) to efficiently manage connections.
* **Asynchronous Operations:** Leverage Node.js's async nature. All Redis operations should be asynchronous.
* **Efficient Keying in Redis:** The chosen keying scheme is direct and avoids complex scans.
* **Redis Pipelining:** For operations involving multiple Redis commands (like setting both ttp: and ptk: mappings), use Redis pipelining or transactions (MULTI/EXEC) to reduce network round-trip latency.
* **Payload Size:** Keep JSON payloads for ptk: mapping as concise as possible.
* **Node.js Clustering:** For CPU-bound tasks or to better utilize multi-core processors, Node.js cluster module can be used to fork multiple worker processes. However, for this primarily I/O-bound task, a single process might be sufficient initially, with horizontal scaling (multiple instances behind a load balancer) being a more common approach.
* **Caching (Application Level):**the application itself could further reduce Redis lookups for ttp: mapping. This adds complexity and should be considered only if Redis becomes a bottleneck.
* **Logging Strategy:** Asynchronous logging to files or external services to avoid blocking the event loop. Avoid excessive logging in hot paths.

Real-world project aspects

If this were a real-world project beyond a test task, several aspects would be improved or extended:

**Proactive Issue Detection:**

Implementing a sophisticated asynchronous link health checker that samples a percentage of the generated affiliate links by choosing links from Redis ptk**:**keys, dedicated log, store of generated links, synthetically generate a known good mapping and so on.

**Robustness and Scalability**

* **Containerization & Orchestration:**
  + Use Kubernetes or a similar orchestrator for managing deployments, scaling, and ensuring high availability.
* **Load Balancing:** Place multiple instances of the application behind a load balancer (e.g., Nginx, HAProxy, or cloud provider's LB) to distribute traffic.
* **Redis Clustering/Sentinel:** For high availability and scalability of Redis, implement Redis Cluster or Redis Sentinel for failover.
* **Advanced Input Validation & Sanitization:** Use a dedicated validation library like Joi or class-validator for more comprehensive and declarative validation rules on request parameters and bodies.
* **More Sophisticated Rate Limiting:** Implement more granular rate-limiting strategies, potentially based on IP, API keys, or user sessions, and consider distributed rate limiting if running multiple instances.
* **Circuit Breaker Pattern:** For calls to Redis (or other external services), implement a circuit breaker (e.g., using opossum) to prevent cascading failures if Redis becomes temporarily unavailable.
* **Configuration Management:** Use a dedicated configuration service (e.g., HashiCorp Consul, AWS Parameter Store) for managing secrets and configurations in production environments.

**Monitoring and Alerting**

* **Comprehensive Monitoring:** Integrate with monitoring tools like Prometheus/Grafana, Datadog, or New Relic to track:
  + Request latency and throughput for each endpoint.
  + Error rates.
  + Node.js process metrics (CPU, memory, event loop lag).
  + Redis performance metrics.
* **Alerting:** Set up alerts for critical issues (e.g., high error rates, service unavailability, Redis down, high latency).
* **Distributed Tracing:** Implement distributed tracing (e.g., OpenTelemetry, Jaeger) to follow requests across services if the system grows more complex.

**Enhanced Security**

* **Authentication/Authorization for Management Endpoints:** Secure /refresh and /retrieve\_original (if it's not public) with API keys, OAuth2, or JWT-based authentication. Implement role-based access control (RBAC).
* **Web Application Firewall (WAF):** Deploy a WAF to protect against common web exploits (XSS, SQLi - though less relevant for Redis, still good practice).
* **Security Audits:** Conduct regular security audits and penetration testing.
* **Data Encryption at Rest for Redis:** If Redis persistence is enabled and contains sensitive (even if pseudonymized) data, consider options for encrypting data at rest.

**Operational Excellence & Developer Experience**

* **CI/CD Pipeline:** Implement a full CI/CD pipeline (e.g., Jenkins, GitLab CI, GitHub Actions) for automated testing, building, and deployment.
* **Automated Testing:**
  + **Unit Tests:** For individual functions and methods in services.
  + **Integration Tests:** Testing interactions between components (e.g., API routes and the service layer, service layer and Redis).
  + **End-to-End Tests:** Testing the full request flow.
* **API Documentation:** Auto-generate interactive API documentation using tools like Swagger/OpenAPI (can integrate with Express using swagger-ui-express and JSDoc comments or OpenAPI spec files). *I've added basic JSDoc for Swagger in the POC.*
* **Idempotency for /refresh:** Ensure that if the /refresh operation is called multiple times (e.g., due to network retries), it behaves predictably, ideally by returning the same "new" our\_param if the inputs haven't changed since the first successful refresh in a short window. This is more complex.
* **Data Archival/Cleanup Strategy:**
  + Over time, the ptk:<our\_param\_value> mappings could grow very large, especially if our\_param values are refreshed frequently.
  + Consider a strategy for archiving or deleting very old, unused ptk: mappings if they are no longer needed for retrieval. This would depend on business requirements for how long original data needs to be reversible.
  + This might involve tracking last access times or creation timestamps.

**Additional Features**

* **Analytics Dashboard:**
  + Track how many times each triplet is requested.
  + Track frequency of refreshes.
  + Identify popular keywords, sources, creatives.
  + This might involve streaming data to an analytics platform or using Redis data structures like sorted sets for counters.
* **Bulk Operations:** API endpoints for bulk refreshing or bulk retrieval of mappings.
* **UI for Management:** A simple web interface for internal users to:
  + View existing mappings.
  + Manually trigger refreshes.
  + Search for mappings.
* **Versioning of our\_param:** If there's a need to distinguish between different "versions" of our\_param generated for the same triplet over time (e.g., for historical analysis connected to a specific our\_param), the system could explicitly version them. The current design implicitly handles this by old our\_params still being retrievable.
* **Customizable our\_param Format/Length:** Allow configuration of the our\_param generation (e.g., length, character set) if different affiliate networks have different requirements.
* **Support for Expiry on Mappings:** Allow our\_param values or mappings to expire after a certain period, useful if links are temporary. Redis has built-in TTL (Time To Live) features.
* **Parameter Aliasing/Grouping:** More advanced: Allow grouping multiple keywords, src, creative combinations under a single our\_param if they should be treated identically by the affiliate.