

prefix_length.py — Prefix Length Feature Collector (Full Technical Documentation)

(Based strictly on script: `prefix_length.py`)

1. Purpose & Role

`prefix_length.py` computes and maintains the **prefix_length** field for every (`asn`, `prefix`) row in the `prefix_data` table.

In the context of the platform, `prefix_length` is a foundational structural feature used for:

- understanding the **granularity** of announced space (e.g. /16 vs /24)
- modeling **subprefix-hijack risk** (short vs long prefixes)
- feeding the **prefix vulnerability ML model** as a core input feature
- interpreting routing behavior in combination with ROA maxLength and VRP coverage

This script does **not** depend on any external APIs. It deterministically derives prefix length from the prefix string itself and keeps the database field up to date.

2. High-Level Behavior

At a high level, the script:

1. Connects to a SQLite database (`asn_data.db` under `database/`).
2. Ensures that the `prefix_data` table has all required columns for this feature.
3. Selects all (`asn`, `prefix`) rows where `prefix_length` is missing (or all rows when `--force` is used).

4. For each prefix, parses the CIDR and extracts the numeric length (e.g. `/24` → `24`).
5. Updates the database with:
 - `prefix_length`
 - timestamps (`prefix_checked_at`, `updated_at`)
 - error metadata (`prefix_error_last` on failure)
6. Repeats the process continuously, with a sleep interval between rounds.

There is no network I/O: all logic is local and CPU-bound.

3. Metrics Produced

3.1 `prefix_length` (INTEGER)

- Derived directly from the CIDR notation, e.g.:
 - `192.0.2.0/24` → `24`
 - `2001:db8::/32` → `32`

If parsing fails (malformed input), `calc_prefix_length()` returns 0. Malformed prefixes are not treated as hard errors; they simply get a `prefix_length` of 0.

3.2 `prefix_checked_at` (TEXT)

- ISO8601 timestamp of the last attempt to process this prefix.

3.3 `prefix_error_last` (TEXT)

- Contains the last unexpected error message encountered while processing this row (e.g., DB or runtime errors). Parsing failures are not treated as hard errors; malformed prefixes simply receive `prefix_length = 0` and `prefix_error_last` remains NULL.

- Set to `NULL` on successful computation.

3.4 updated_at (TEXT)

- ISO8601 timestamp for the last update to this row by this script.
-

4. Database Contract

4.1 Required input schema

The script expects a table named:

- `prefix_data`

with at least the following columns present:

- `asn INTEGER`
- `prefix TEXT`

It **automatically adds** the following columns if they are missing:

- `prefix_length INTEGER`
- `prefix_checked_at TEXT`
- `prefix_error_last TEXT`
- `updated_at TEXT`

This is implemented by `ensure_columns()` via `PRAGMA table_info()` and `ALTER TABLE` statements.

4.2 Update behavior

Updates are done in-place, not via inserts.

On success, `bulk_update_success()` executes:

```
UPDATE prefix_data
SET prefix_length = ?,
    prefix_checked_at = ?,
    prefix_error_last = NULL,
    updated_at = ?
WHERE asn = ? AND prefix = ?;
```

On failure, `bulk_update_error()` executes:

```
UPDATE prefix_data
SET prefix_error_last = ?,
    prefix_checked_at = ?,
    updated_at = ?
WHERE asn = ? AND prefix = ?;
```

This guarantees that each `(asn, prefix)` pair gets one authoritative `prefix_length` value and proper audit metadata.

5. Data Flow Overview

5.1 Target selection

`load_targets()`:

- Builds a base query:
`SELECT asn, prefix FROM prefix_data`
- If `--force` is **not** set, it adds:
`WHERE prefix_length IS NULL`
- Iterates asynchronously through results and collects a list of `(asn, prefix)` tuples.

This ensures the script only recomputes what is necessary, unless explicitly forced.

5.2 Prefix length computation

For each `(asn, prefix)`:

- `calc_prefix_length(prefix)`:
 - splits on `'/'`
 - converts the part after `'/'` to `int`
 - returns the value if successful
 - returns `0` on any parsing error

This is a simple, deterministic, $O(1)$ operation per prefix.

5.3 Batch updates

The script accumulates results in memory:

- `batch_ok` for successful computations
- `batch_err` for failures

When `len(batch_ok) >= db_batch`, it calls `bulk_update_success()` and commits.

When `len(batch_err)` exceeds a threshold (`max(200, db_batch // 4)`), it calls `bulk_update_error()` and commits.

Finally, any remaining batches are flushed at the end of `run_once()`.

6. Performance & Concurrency

Concurrency model

- A global `asyncio.Semaphore(args.concurrency)` limits how many prefixes are processed concurrently.

- A worker coroutine is created per `(asn, prefix)` in batched chunks.

Although the computation itself is extremely fast (string split + int), this architecture:

- aligns with the rest of the platform's async design
- allows scaling to millions of rows without blocking
- keeps the structure similar to network-bound collectors

Batching

- `batch_size = 100` controls how many prefixes are processed per batch of tasks.
- `db_batch` controls how many successful results are grouped per `executemany()` call.

This balances DB write overhead and memory usage.

Database efficiency

Using batched `executemany()` updates:

- reduces transaction overhead
- minimizes lock contention in SQLite
- keeps the script efficient even for large tables

7. Control Loop Behavior

The runtime model is:

- `run_once(args)`:
 - Ensures columns exist
 - Loads targets

- Processes all selected prefixes once
 - Commits any remaining updates
- `run_forever(args)`:
 - Logs a start message with timestamp
 - Calls `run_once()`
 - Logs any round-level error
 - Sleeps `DEF_SLEEP_BETWEEN_ROUNDS` seconds (3600s / 60min)
 - Repeats indefinitely

This makes the collector fully autonomous:

- it periodically reconverges the database toward a complete, correct `prefix_length` coverage.

8. CLI Arguments

Available CLI options (via `argparse`):

Argument	Description	Default
<code>--db</code>	SQLite DB path	resolved to <code>database/asn_data.db</code> if relative
<code>--force</code>	Recompute for all rows, ignoring existing <code>prefix_length</code>	<code>False</code>
<code>--concurrency</code>	Maximum concurrent prefix workers	<code>800</code>
<code>--db-batch</code>	Number of rows per DB update batch (success path)	<code>1</code>

Behavior notes

- If `--db` is not an absolute path, the script overrides it to use the standard project DB under `database/asn_data.db`.
 - `--force` is important for recomputing after code changes or DB migrations.
-

9. Usage Examples

Compute `prefix_length` for missing rows:

```
python3 prefix_length.py --db database/asn_data.db
```

Force recomputation for all prefixes:

```
python3 prefix_length.py --db database/asn_data.db --force
```

Tune concurrency and batch size:

```
python3 prefix_length.py --concurrency 400 --db-batch 50
```

Run continuously (as a background service):

```
python3 prefix_length.py
```

The script will:

- compute `prefix_length` for all missing rows on the first round
 - sleep 60 minutes
 - run again to catch any new prefixes added by other collectors
-

10. Integration with the Rest of the Platform

`prefix_length` is consumed by:

- the **prefix vulnerability ML model**, as a direct feature
- logic that interprets:
 - ROA `maxLength` vs actual prefix length
 - likelihood of subprefix hijacks
 - routing behavior for short vs long prefixes

It also plays a supporting role in:

- risk surface computation
- AS+prefix attack-surface modeling
- reporting and analytics on prefix size distributions

Since it is cheap and deterministic, this collector can safely run frequently.