Developer Documentation

1) Executive Summary

The Network Topology Simulator ingests device configuration files, auto-builds a hierarchical network topology, validates configuration and capacity health, proposes optimization recommendations, and simulates Day-1/Day-2 behaviors including link failures. This directly addresses the VIP 2025 Networking problem statement requirements such as **topology generation from configs**, **bandwidth awareness**, **load balancing advice**, **validation of duplicates/VLAN/MTU/loops**, **Day-1 discovery and failure simulation**, and **multithreaded + IPC implementation**.

2) Requirements Mapping (VIP 2025)

- **Automatic topology generation from device configs** → `config_parser.py`, `topology_builder.py`
- **Bandwidth awareness & capacity checks** → `topology_builder.py` (link bw), `optimizer.py` (peak load vs link capacity)
- **Load balancing recommendations** → `optimizer.py` (secondary path / shaping recommendations)
- **Validation** (duplicate IPs, wrong VLAN labels, incorrect gateways, MTU mismatch, loops, missing switch configs) → `validator.py` (duplicates, MTU, loops, missing components)
- **Protocol recommendations (BGP vs OSPF)** → `optimizer.py`
- **Day-1 discovery & Day-2 link failure simulation; impact analysis** → `simulator.py` (HELLO/ARP/OSPF logs, fault injection & affected nodes)
- **Architecture**: multithreading per device, IPC (FIFO/TCP semantics), stats/logs → `simulator.py`, `ipc.py`

3) Repository Layout

```
src/
  __init__.py
  config_parser.py       # Parse configs → Device graph
  topology_builder.py    # Build Topology + link properties (VLAN, MTU, bw)
  validator.py           # Duplicate IPs, MTU mismatches, loops, missing devices
  optimizer.py           # Load/capacity checks, LB, protocol, aggregation
  simulator.py           # Day-1/Day-2 simulation with threads + IPC
  ipc.py                 # Simple FIFO-like IPC abstraction
  main.py                # CLI orchestration, outputs (logs, report, topology.png)

tests/
  test_parser.py
  test_validator.py
  test_simulator.py
  sample_configs/ (R1.dump, R2.dump, SW1.dump, SW2.dump)

outputs/                # Generated artifacts
requirements.txt
```

4) Data Model (used across modules)

Defined in `config_parser.py` using dataclasses

- `Interface`: `name`, `ip`(CIDR), `vlan`(int), `mtu`(int), `bw_mbps`(int), `neighbor`(str|None)
- `Endpoint`: `name`, `app_type`, `peak_mbps`, `avg_mbps`

- `Device`: `hostname`, `role` (`router|switch`), `interfaces: [Interface]`, `endpoints: [Endpoint]`, `gateway: Optional[str]`, `ospf_enabled: bool`

These types ensure a clean, strongly-typed boundary between parse/build/validate/simulate stages.

5) Module-by-Module Deep Dive

5.1 `config_parser.py`

Purpose: Parse `*.dump` files into `Device` objects.

Key functions

- `parse_config_file(path: Path) -> Device`
    - Uses regex to extract:
        - `hostname`, `role`
        - `interface ... ip <cidr> vlan <id> mtu <n> bw_mbps <n> neighbor <host>` lines → `Interface` entries
        - `endpoint app type <t> peak_mbps <n> avg_mbps <n>` → `Endpoint` entries
        - `gateway <ip>` (optional)
        - `ospf enabled` (flag)
- `load_directory(conf_dir: str) -> Dict[str, Device]`
    - Reads all `*.dump` and returns `{hostname: Device}` map.

Why this design?
A minimal grammar keeps parsing robust and testable. You can extend the regexes to support vendor-specific CLI outputs later (Cisco IOS/JunOS).

Typical pitfalls & tips

- Ensure each device file has a unique `hostname`.
- Keep the grammar incremental—add new fields as needed without breaking existing patterns.

5.2 `topology_builder.py`

Purpose: Transform `{hostname: Device}` into a `Topology` graph with `Link` metadata.

Data structures

- `Link`: `a`, `b`, `vlan`, `bw_mbps`, `mtu_a`, `mtu_b`
- `Topology`: `nodes: [str]`, `links: [Link]`

Key function

- `build_topology(devices) -> Topology`
    - Iterates interfaces; when `neighbor` is present and found among devices, it creates a single de-duplicated `Link (A–B)` per VLAN.
    - **Bandwidth**: stores the min of both ends (conservative capacity).
    - **MTU**: retains MTU from each end to enable mismatch validation.

Rationale
The tool must be **bandwidth aware** and **MTU aware** to support capacity checks and mismatch detection.

5.3 **`validator.py`**

Purpose: Detect health and consistency issues.

Functions

- `find_duplicate_ips(devices) -> List[str]`
  - Groups interfaces by VLAN; flags IPs that appear >1 within same VLAN.
- `find_mtu_mismatches(topo) -> List[str]`
  - Scans links for `mtu_a != mtu_b`.
- `detect_loops(topo) -> List[str]`
  - Simple cycle detection using BFS/DFS over the undirected graph.
- `find_missing_components(devices) -> List[str]`
  - Reports neighbors referenced by interfaces that don't exist as parsed devices (missing config file).

Why it matters
The brief requires detection of config flaws (duplicate IPs, incorrect VLAN/gateways, MTU mismatches, loops, missing switch configs). The current validator covers duplicates, MTU, loops, and missing components; VLAN/gateway sanity checks can be added following the same pattern.

5.4 **`optimizer.py`**

Purpose: Provide recommendations for capacity and design.

Core logic

- `summarize_endpoint_loads(devices)`
  - Aggregates **peak Mbps** per node (to model worst-case).
- `recommend_load_balancing(devices, topo) -> List[str]`
  - For each link, compares adjacent node peak loads vs link bandwidth; if load > capacity, recommends secondary paths/traffic shaping.
- `suggest_protocols(devices) -> List[str]`
  - Heuristic: if multiple routers and peering complexity grows, **suggest BGP for inter-domain** and **OSPF intra-domain** (as per requirement to recommend BGP vs OSPF).
- `node_aggregation(topo) -> List[str]`
  - Suggests collapsing single-degree access switches into upstream routers to reduce node count (when policy allows).

Rationale
This addresses **load balancing**, **protocol recommendations**, and **node aggregation** requirements.

5.5 **`ipc.py`**

Purpose: Lightweight, FIFO-like IPC for inter-thread messaging.

Class

- `InMemoryIPC`

- Thread-safe internal queue registry keyed by destination name (e.g., hostname).
- `send(target, msg)` and `recv(target, timeout)` APIs.

Why this design?
The brief allows FIFO/TCP/IP for metadata exchange; this in-memory queue models FIFO semantics and can be swapped for sockets later.

5.6 `simulator.py`

Purpose: Multithreaded simulation of devices and events.

Core types & flow

- `DeviceThread(Thread)`
  - Each device is a thread. On start, sends "hello" to neighbors (Day-1 presence).
  - Receives from IPC, appends structured log entries.
  - If `ospf_enabled`, logs OSPF neighbor discovery. Always logs ARP priming.
- `run_simulation(devices, topo, link_fail: str | None)`
  - Spins up one thread per device.
  - If `link_fail="A-B"`, neighbor relations on that link are omitted, and the affected endpoints are reported.
  - Returns `(logs, affected_nodes)` for report collation.

Why this design?
It satisfies the **Day-1/Day-2 simulation** requirement with multithreading and IPC. Shows **which endpoints/nodes are affected by link failures** and logs **discovery events**.

5.7 `main.py`

Purpose: CLI orchestration & output generation.

Flow

1. Load devices (`load_directory`)
2. Build topology (`build_topology`)
3. Validate (duplicates, MTU mismatches, loops, missing components)
4. Optimize (LB, protocol suggestions, aggregation)
5. Simulate (`run_simulation`) with optional `--link-fail A-B`
6. Write artifacts:
   - `outputs/logs.txt` (device/thread logs)
   - `outputs/simulation_report.txt` (validations + recommendations + failure impact)
   - `outputs/topology.png` (basic diagram with routers/switches)

6) End-to-End Execution Flow

```
Configs → Parse Devices → Build Topology → Validate → Optimize
→ Simulate → Write Artifacts
```

Mermaid (Architecture)

```
flowchart TB
    subgraph CLI["Orchestrator (CLI) — main.py"]
        RUN["run(conf_dir, out_dir, link_fail)"]
    end
    subgraph Parser["Configuration Parser — config_parser.py"]
        P1["Load *.dump files"]
        P2["Parse hostnames, roles, interfaces, endpoints, OSPF flags"]
        P3["Emit {hostname→Device}"]
    end
    subgraph Topology["Topology Builder — topology_builder.py"]
        T1["Join neighbors"]
        T2["Annotate Links (VLAN, MTU_a/b, bw=min)"]
        T3["Emit Topology{nodes, links}"]
    end
    subgraph Validator["Validation — validator.py"]
        V1["Duplicate IPs"]
        V2["MTU mismatches"]
        V3["Loop detection"]
        V4["Missing device configs"]
    end
    subgraph Optimizer["optimizer.py"]
        O1["Aggregate peak loads"]
        O2["Recommend load balancing/shaping"]
        O3["Protocol: BGP vs OSPF"]
        O4["Node aggregation"]
    end
    subgraph Simulator["Simulation — simulator.py"]
        direction TB
        S1["DeviceThread per device"]
        S2["InMemory FIFO (ipc.py)"]
        S3["Day-1: HELLO/ARP/OSPF"]
        S4["Day-2: Link failure A-B"]
        S5["Impact analysis"]
    end
    subgraph Outputs["outputs/"]
        G1["topology.png"]
        G2["logs.txt"]
        G3["simulation_report.txt"]
    end
    RUN --> P1 --> P2 --> P3
    P3 --> T1 --> T2 --> T3
    T3 --> V1 & V2 & V3 & V4
    T3 --> O1 --> O2
    O1 --> O3
    O1 --> O4
    P3 --> S1
    T3 --> S1
    S1 --> S2
    S1 --> S3
    S1 --> S4 --> S5
    V1 -. issues .-> G3
    V2 -. issues .-> G3
    V3 -. issues .-> G3
    V4 -. issues .-> G3
    O2 -. recs .-> G3
    O3 -. recs .-> G3
    O4 -. recs .-> G3
    S3 --> G2
    S4 --> G2
    S5 --> G3
    T3 --> G1
```

7) CLI Usage & Configuration Grammar

Run

```
# Install dependencies
pip install -r requirements.txt

# Baseline run (with included samples)
python -m src.main --conf tests/sample_configs --out outputs

# Inject a link failure
python -m src.main --link-fail R1-R2
```

Config grammar (example)

```
hostname R1
role router
interface Gi0/0 ip 10.0.0.1/24 vlan 10 mtu 1500 bw_mbps 100 neighbor R2
endpoint app type web peak_mbps 60 avg_mbps 20
gateway 10.0.0.254
ospf enabled
```

8) Algorithms & Complexity

- **Topology construction**: iterate neighbor relations → **O(E)**
- **Duplicate IPs per VLAN**: aggregate and count → **O(N)** per VLAN
- **Loop detection**: BFS/DFS on undirected graph → **O(V + E)**
- **MTU mismatch scan**: per edge → **O(E)**
- **Load balancing checks**: compute per-node peak, then compare at edges → **O(V + E)**

9) Testing Strategy (pytest)

- `test_parser.py` → verifies device loading and basic interface extraction.
- `test_validator.py` → confirms MTU mismatch detection with provided sample configs.
- `test_simulator.py` → ensures link-failure injection produces expected log/impact signals.

Extending tests

- Add duplicates/VLAN/gateway unit tests.
- Add property tests for parser (e.g., fuzz IP, VLAN, MTU values).
- Add large-topology tests to measure performance.

10) Output Artifacts

- **`outputs/topology.png`**: simple 2-tier layout; annotate edges with VLAN/bandwidth.
- **`outputs/logs.txt`**: per-thread/device discovery and event logs.
- **`outputs/simulation_report.txt`**:
  - **Validations**: duplicates, MTU mismatches, loops, missing device configs
  - **Recommendations**: LB suggestions, protocol recommendations, node aggregation
  - **Failure impact**: affected nodes if `--link-fail` provided

11) Extension Roadmap

- **Parsing**: Add robust parsers for Cisco IOS/JunOS; introduce tokenization or a grammar parser as it grows.
- **IPC**: Swap `InMemoryIPC` for TCP sockets or OS FIFOs to model cross-process capacity and backpressure.
- **Protocol fidelity**: Implement ARP caches, OSPF adjacencies/LSAs, ICMP MTU black-hole detection.
- **Optimization**: Integrate TE policies (ECMP, policy routing) & SLA targets.
- **Visualization**: Use NetworkX/Graphviz for richer graphs and metrics overlays.

12) Common Pitfalls & Debugging

- **No topology image** → ensure `matplotlib` is installed (`pip install -r requirements.txt`).
- **No MTU mismatches detected** → confirm sample configs actually differ in MTU on both ends.
- **Missing device warnings** → verify the `neighbor` names match `hostname` of a provided config.
- **Threads appear idle** → increase simulated runtime or instrument additional logs in `DeviceThread.run`.

Final notes

This documentation is written to read like an internal engineering handbook while explicitly showing how the implementation meets VIP 2025 requirements across **topology creation, validation, optimization, and simulation with multithreading and IPC**