MCP Network Optimizer

Intelligent, Safe, and Autonomous Linux Network Configuration via Model Context Protocol

Surriya Gokul
23110324
Computer Science and Engineering
Indian Institute of Technology Gandhinagar
Palaj, GJ 382355, India
`surriya.gokul@iitgn.ac.in`

Burra Saharsh
23110071
Computer Science and Engineering
Indian Institute of Technology Gandhinagar
Palaj, GJ 382355, India
`burra.saharsh@iitgn.ac.in`

Srajan Dehariya
23110320
Computer Science and Engineering
Indian Institute of Technology Gandhinagar
Palaj, GJ 382355, India
`srajan.dehariya@iitgn.ac.in`

November 12, 2025

Abstract

Traditional network optimization requires deep Linux expertise and manual command execution, creating significant barriers for system administrators and developers. We present MCP Network Optimizer, an AI-native system that achieves high-level optimization goals and executes low-level Linux networking commands through the Model Context Protocol (MCP). The system features 29 configuration cards spanning TCP/IP tuning, traffic control, and firewall configuration, along with 5 research-backed profiles (Gaming, Streaming, Video Calls, Bulk Transfer, Server) optimized for specific workloads. Safety mechanisms include command allowlisting, automatic checkpointing and rollback, schema validation via Pydantic, policy enforcement, and comprehensive audit logging. Our evaluation demonstrates 37.3% latency reduction for bulk transfer workloads, 87.7% jitter reduction, 30% jitter improvement for gaming, 20.5% jitter improvement for video calls, 21.1% connection time optimization for streaming, and 32.3% DNS query improvement for server workloads. The declarative and type-safe architecture with atomic operations allows AI agents to optimize network performance safely while ensuring the stability of production systems.

1 Introduction

Network performance optimization is critical for modern applications ranging from competitive gaming to enterprise servers. However, optimizing Linux networking requires navigating a complex landscape of TCP/IP parameters, traffic control queuing disciplines, and firewall configurations. System administrators must understand intricate interactions between kernel parameters (e.g., `tcp_congestion_control`, buffer sizes), traffic shaping mechanisms (fq, pfifo_fast, fq_codel), and security policies.

The emergence of Large Language Models (LLMs) and AI agents creates an opportunity to make network optimization more accessible. However, directly exposing system commands to AI agents poses significant safety risks: command injection vulnerabilities, unintended system changes, and lack of rollback capabilities. Existing network management tools fall into two categories: low-level utilities (`sysctl`, `tc`, `nft`) requiring expert knowledge, and high-level frameworks lacking AI integration and safety guarantees.

We present **MCP Network Optimizer**, an AI-native network configuration system built on the Model Context Protocol. Our system addresses three key challenges:

1. **Complex Configuration**: High-level goals require error-prone low-level commands and also requires deep kernel expertise.
2. **Unsafe Operations**: Manual changes risk outages & drift and also no rollback or consistency at scale.
3. **No Smart Automation**: AI agents lack safe interfaces to optimize networks and cannot validate or adapt to dynamic workloads.

1.1 Key Contributions

- **Comprehensive Configuration Framework**: 29 configuration cards covering alomost the entire Linux networking stack with formal specifications
- **Research-Backed Profiles**: 5 optimization profiles validated against academic research and industry standards

- **Safety Architecture**: Multi-layer validation including Pydantic schemas, policy enforcement, and atomic operations
- **MCP Integration**: Native support for AI agents through the Model Context Protocol with 30+ discovery tools
- **Production Deployment**: Checkpoint/rollback system, audit logging, and performance validation framework

2 System Architecture

MCP Network Optimizer implements a 5-stage pipeline transforming high-level optimization goals into validated, executable system commands (Figure 1).

2.1 Stage 1: Discovery

The discovery stage provides 30+ introspection tools with no side effects, organized into categories:

- **Network Interfaces**: `ip_info`, `eth_info`, `nmcli_status`, `iwconfig`
- **Routing & DNS**: `ip_route`, `arp_table`, `resolvectl_status`, `dig`
- **Performance**: `ping_host`, `traceroute`, `ss_summary`
- **QoS & Firewall**: `tc_qdisc_show`, `nft_list_ruleset`

All discovery tools return structured JSON with standardized fields (`ok`, `code`, `stdout`, `stderr`), enabling reliable parsing by AI agents.

2.2 Stage 2: Planning

Users specify optimization goals declaratively using Pydantic-validated `ParameterPlan` models:

Listing 1: Example Gaming Optimization Plan

```
plan = {
  "iface": "eth0",
  "profile": "gaming",
  "changes": {
    "sysctl": {
      "net.ipv4.tcp_low_latency": "1",
      "net.ipv4.tcp_fastopen": "3"
    },
    "qdisc": {"type": "fq"}
  },
  "rationale": ["Minimize_latency"]
}
```

Plans support these change types: `sysctl`, `qdisc`,, `netem`, `htb_classes`, `dscp`, `connection_limits`, `rate_limits`, `connection_tracking`, `nat_rules`.

2.3 Stage 3: Validation

Multi-layer validation ensures safety before execution:

1. **Schema Validation**: Pydantic type checking, range constraints, enum validation
2. **Policy Enforcement**: Checking against `limits.yaml` (buffer sizes, bandwidth limits, port ranges)
3. **Interface Validation**: Verifying target interface exists
4. **Config Card Validation**: Ensuring parameters match specifications

Example validation constraints from `validation_limits.yaml`:

Listing 2: Bandwidth Validation Limits

```
bandwidth:
  max_mbps: 100000  # 100 Gbps
  min_mbps: 1
  rules:
    - "ceil_mbit>=egress_mbit"
    - "egress_mbitwithin[1,100000]"
```

2.4 Stage 4: Rendering

The planner (`server/tools/planner.py`) translates validated `ParameterPlan` objects into `RenderedPlan` containing executable commands:

- **Sysctl**: List of `sysctl -w key=value` commands
- **TC**: Bash script with `tc qdisc/class/filter` commands
- **Nftables**: Complete ruleset with tables, chains, rules

Rendering logic implements all 29 configuration cards deterministically, producing stable output .

2.5 Stage 5: Application

Atomic application with automatic rollback (`server/tools/apply/apply.py`):

Listing 3: Safe Application Workflow

```
def apply_rendered_plan(plan, label):
  checkpoint_id = snapshot_checkpoint(label)
  try:
    execute_sysctl_commands()
    execute_tc_script()
    execute_nft_script()
    return success_report(checkpoint_id)
  except Exception as e:
    rollback_to_checkpoint(checkpoint_id)
    return failure_report(e)
```

Checkpoints capture complete system state: sysctl parameters, tc configuration, nftables rules. Rollback restores the pre-change state.

3 Configuration Cards

The system implements 29 configuration cards organized into three categories, each with formal specifications including purpose, parameters, validation rules, and research citations.

```
DISCOVER  ->    PLAN    ->  VALIDATE  ->   RENDER   ->   APPLY

    ↓               ↓              ↓               ↓             ↓
  Inspect        Declare          Check          Generate       Execute
  System          Intent         Policies       Commands        Safely
  State        (Pydantic)      & Limits       & Scripts        Rollback
```

Figure 1: Five-stage optimization pipeline with safety guarantees at each stage

3.1 Sysctl Cards (16 cards)

TCP/IP kernel parameter tuning:

Table 1: Key Sysctl Configuration Cards

Card	Impact	Use Case
tcp_congestion_control	Throughput	BBR (high bandwidth)
tcp_low_latency	Latency	Gaming, real-time
tcp_fastopen	Handshake	Reduces 1 RTT
tcp_window_scaling	BDP	Essential for >10Mbps
core_rmem/wmem_max	Buffers	High-throughput apps
tcp_syncookies	Security	SYN flood protection

BBR Congestion Control: Based on Google's research ([1] Cardwell et al., ACM Queue 2016), BBR achieves 2–25× higher throughput than CUBIC on paths with bufferbloat. The `gaming` and `streaming` profiles default to BBR paired with the `fq` (Fair Queue) qdisc, as recommended in the original paper.

TCP FastOpen (RFC 7413): Reduces connection establishment latency by 1 RTT (15–25% faster in benchmarks). Enabled in all profiles except `bulk_transfer` where connection setup is amortized over long transfers.

3.2 Traffic Control Cards (7 cards)

Queue disciplines, shaping, and network emulation:

- **tc_qdisc_type**: Supports CAKE, fq_codel, HTB, fq, pfifo_fast. The `fq_codel` qdisc is the default in Linux generally.
- **tc_htb_rate/ceil**: Hierarchical Token Bucket for bandwidth guarantees and burst capacity. Used in `streaming` profile for predictable QoS.
- **tc_htb_priority**: Traffic prioritization (1–7, lower = higher priority). Gaming profile uses priority 1 for latency-sensitive traffic.
- **tc_netem_delay/loss**: Network emulation for testing. Supports delay (ms), jitter, packet loss, duplication, corruption, and reordering.

3.3 Firewall/Security Cards (6 cards)

Connection limiting, rate limiting, QoS marking, and NAT:

- **iptables_connection_limiting**: Per-IP connection limits prevent DoS(Denial of Service , Ex: SYN flood attacks). Server profile limits HTTP/HTTPS to 100 connections per IP.
- **iptables_rate_limiting**: Packet rate limiting (e.g., 1000/second with burst=50). Protects against packet floods.
- **iptables_connection_tracking**: Configures `nf_conntrack_max` (up to 1M concurrent connections for servers).
- **iptables_QoS_marking**: DSCP marking for traffic prioritization.
- **iptables_nat**: Supports MASQUERADE, SNAT, DNAT for routing and gateway scenarios.

4 Optimization Profiles

Five research-backed profiles optimize for specific workloads, each activating a curated subset of configuration cards. We tested all five profiles (Gaming, Bulk Transfer, Streaming, Video Calls, Server) against a Balanced baseline.

4.1 Gaming Profile

Objectives: Ultra-low latency, minimal jitter and packet loss.

Active Cards (13): BBR congestion control, `tcp_low_latency=1`, TCP FastOpen, fq qdisc, conservative buffers (16MB), `tcp_fin_timeout=10s`, `netdev_budget=600`.

Research Basis: [10] Valve Source Engine documentation specifies 15ms tick rate (66.67 ticks/second) for competitive gaming.

Benchmark Results:

- Average latency: 18.90ms → 19.97ms (slight increase, +5.7%)
- Jitter: 46.20ms → 32.40ms (30% reduction)
- Max latency: 61.1ms → 47.3ms (23% reduction)
- Connection time: 29.32ms → 24.30ms (17% faster)
- DNS resolution: 18.40ms → 24.00ms (trade-off, +30%)
- Multi-host average: 12.05ms → 10.72ms (11% improvement)
- Packet loss: 0.0% (maintained perfectly)

4.2 Bulk Transfer Profile

Objectives: Absolute maximum throughput, latency not critical.

Active Cards (12): BBR, maximum buffers (256MB), TCP window scaling, timestamps, fq qdisc, expanded port range (1024–65000), `netdev_budget=1000`.

Buffer Sizing: Follows RFC 1323 bandwidth-delay product formula: $BDP = bandwidth \times RTT$.

Benchmark Results:

- Latency: 29.42ms → 18.44ms (37.3% reduction)
- Jitter: 106.00ms → 13.00ms (87.7% reduction)
- Connection time: 45.63ms → 38.28ms (16.1% faster)
- DNS resolution: 16.60ms → 15.40ms (7.2% improvement)

4.3 Streaming Profile

Objectives: Maximum throughput with optimized connection time.

Active Cards (12): BBR, large buffers (64MB), TCP window scaling, fq qdisc, HTB rate/ceil, `netdev_budget=600`.

Research Basis:[1] The BBR paper demonstrates a 2–25× throughput improvement on bufferbloat-affected paths. Large buffers required for high bandwidth-delay product (BDP) paths per RFC 1323.

Benchmark Results:

- Latency: 18.48ms → 18.16ms (1.7% reduction)
- Jitter: 11.70ms → 15.10ms (trade-off for connection optimization)
- Connection time: 49.31ms → 38.89ms (21.1% faster)
- DNS resolution: 17.00ms → 14.20ms (16.5% improvement)

4.4 Video Calls Profile

Objectives: Balanced latency/throughput, <150ms latency ([5] ITU-T G.114).

Active Cards (13): BBR, TCP FastOpen, moderate buffers (32MB), fq qdisc, DSCP marking.

Research Basis: [5] ITU-T G.114 specifies 150ms one-way delay threshold for acceptable voice quality. WebRTC standards (RFC 7478) guide QoS implementation.

Benchmark Results:

- Latency: 20.545ms → 19.295ms (6.1% reduction)
- Jitter: 57.60ms → 45.80ms (20.5% reduction)
- Connection time: 44.323ms → 39.944ms (9.9% faster)
- DNS resolution: 19.0ms → 14.8ms (22.1% improvement)
- ITU-T G.114 compliance: Latency well below 150ms threshold

4.5 Server Profile

Objectives: High concurrency (10K–1M connections), DDoS protection, latency <50ms under load.

Active Cards (17): BBR, SYN cookies, large SYN backlog (8192), connection limiting (100/IP), rate limiting (1000 pkt/s), connection tracking (1M max), TCP FastOpen, balanced buffers (64MB).

Security Features:

- SYN flood protection via `tcp_syncookies=1` (RFC 4987)
- Per-IP connection limits: 100 for HTTP/HTTPS
- Rate limiting: 1000 packets/second with burst=50
- Connection tracking: `nf_conntrack_max=1000000`

Benchmark Results:

- Latency: 19.21ms → 17.25ms (10.2% reduction)
- Jitter: 25.20ms → 18.00ms (28.6% reduction)
- Max latency: 40.6ms → 33.1ms (18.5% reduction)
- DNS resolution: 19.80ms → 13.40ms (32.3% improvement)
- Connection time: 24.99ms → 25.62ms (within margin)

4.6 Gaming Profile

Objectives: Ultra-low latency, minimal jitter and packet loss.

Active Cards (13): BBR congestion control, `tcp_low_latency=1`, TCP FastOpen, fq qdisc, conservative buffers (16MB), `tcp_fin_timeout=10s`, `netdev_budget=600`.

Research Basis: [10] Valve Source Engine documentation specifies 15ms tick rate (66.67 ticks/second) for competitive gaming.

Benchmark Results:

- Average latency: 18.90ms → 19.97ms (slight increase, +5.7%)
- Jitter: 46.20ms → 32.40ms (30% reduction)
- Max latency: 61.1ms → 47.3ms (23% reduction)
- Connection time: 29.32ms → 24.30ms (17% faster)
- DNS resolution: 18.40ms → 24.00ms (trade-off, +30%)
- Multi-host average: 12.05ms → 10.72ms (11% improvement)
- Packet loss: 0.0% (maintained perfectly)

5 Safety Architecture

Through six mechanisms:

5.1 Command Allowlisting

Only pre-approved binaries can execute (`server/config/allowlist.yaml`):

Listing 4: Command Allowlist

```
binaries:
  - /usr/sbin/sysctl
  - /usr/sbin/tc
  - /usr/sbin/nft
  - /usr/sbin/ip
  - /usr/sbin/ethtool
  - /usr/bin/ping
  - /usr/bin/hostname
  - /usr/bin/hostnamectl
  - /usr/bin/arp
  - /usr/bin/cat
  - /usr/bin/ss
  - /usr/bin/nmcli
  - /usr/sbin/iwconfig
  - /usr/sbin/iwlist
  - /usr/bin/resolvectl
  - /usr/bin/dig
  - /usr/bin/host
  - /usr/bin/nslookup
  - /usr/bin/which
  - /bin/sh
```

Shell metacharacters (`;`, `&&`, `||`, `|`, >, <, backticks) rejected before execution. Commands executed as `argv` arrays, never via `shell=True`.

5.2 Schema Validation

Pydantic models enforce type safety:

Listing 5: ParameterPlan Schema

```
class ParameterPlan(BaseModel):
  iface: NonEmptyStr
  profile: NonEmptyStr
  changes: Changes
  validation: Optional[ValidateSpec]
  rationale: Optional[NonEmptyStr]

class Shaper(BaseModel):
  ingress_mbit: Optional[
    conint(gt=0, le=100000)]
  egress_mbit: Optional[
    conint(gt=0, le=100000)]
  ceil_mbit: Optional[
    conint(gt=0, le=100000)]
```

Type checking, range validation, and enum constraints prevent invalid configurations at parse time.

5.3 Policy Enforcement

All changes validated against `policy/limits.yaml` and `policy/validation_limits.yaml`:

- Buffer sizes: 16MB–1GB (system memory dependent)
- Bandwidth: 1 Mbps–100 Gbps
- Port ranges: 1–65535
- Connection limits: 1–10,000 per IP
- DSCP values: EF, CS6, CS5, CS4, AF41–43 only

5.4 Checkpoint/Rollback

Atomic operations with automatic rollback (`server/tools/apply/checkpoints.py`):

Checkpoint Creation:

1. Snapshot sysctl parameters (`sysctl -a`)
2. Save tc configuration (qdisc, class, filter per interface)
3. Export nftables ruleset (`nft list ruleset`)
4. Record ethtool settings (offloads, interface info)
5. Capture IP configuration (addresses, state)
6. Store metadata (timestamp, label, notes)

Rollback Procedure:

1. Restore sysctl parameters
2. Clear and recreate tc configuration
3. Flush and reload nftables ruleset

Checkpoints saved to `mcp-net-optimizer/checkpoints`

5.5 Audit Logging

Comprehensive logging to `mcp-net-optimizer/audit_logs`

- **Plan Validation**: Intent, profile, validation result, issues
- **Plan Rendering**: Command counts, script presence
- **Checkpoint Creation**: ID, label, timestamp
- **Command Execution**: Command, success/failure, stdout/stderr, checkpoint ID
- **Plan Application**: Applied status, errors, notes
- **Rollback**: Checkpoint ID, success, restoration notes
- **Validation Tests**: Profile, before/after metrics, decision, score

Logs written to both human-readable text (`current.log`) and machine-parseable JSON (`audit_log.json`).

5.6 Atomic Operations

All-or-nothing application ensures system consistency:

Listing 6: Atomic Application

```
checkpoint = create_checkpoint()
try:
  for cmd in sysctl_cmds:
    execute_or_raise(cmd)
  execute_or_raise(tc_script)
  execute_or_raise(nft_script)
  commit_success()
except Exception:
  rollback(checkpoint)
  report_failure()
```

If any command fails, the entire plan rolls back. No partial configurations.

6 Model Context Protocol Integration

MCP Network Optimizer natively integrates with AI agents via FastMCP:

6.1 Tool Registration

38 MCP tools across five categories:

1. **Discovery (30 tools)**: Network introspection with no side effects
2. **Planning & Validation (2 tools)**: `validate_change_plan_tool`, `render_change_plan_tool`
3. **Execution & Safety (4 tools)**: `apply_rendered_plan_tool`, `snapshot_checkpoint_tool`, `rollback_to_checkpoint_tool`, `list_checkpoints_tool`
4. **Performance Testing (4 tools)**: `test_network_performance_tool`, `quick_latency_test_tool`, `validate_configuration_changes_tool`, `auto_validate_and_rollback_tool`
5. **Audit (2 tools)**: `get_audit_log_tool`, `search_audit_log_tool`

6.2 Resource Exposure

Policy configuration cards exposed as MCP resources:

Listing 7: MCP Resource Registration

```
@mcp.resource("policy://config_cards/list")
def get_policy_card_list():
  cards = policy_registry.list()
  return {"count": len(cards),
          "cards": cards}

@mcp.resource("policy://config_cards/{card_id}")
def get_policy_card(card_id: str):
  return policy_registry.get(card_id)
```

AI agents can query available optimization options before creating plans.

6.3 Client Configuration

Claude Desktop/IDE integration:

Listing 8: MCP Client Configuration

```
{
  "mcpServers": {
    "network-optimizer": {
      "command": "python",
      "args": ["-m", "server.main"],
      "cwd": "/path/to/mcp-net-optimizer"
    }
  }
}
```

7 Validation Framework

Automated performance validation compares before/after metrics to decide KEEP, ROLLBACK, or UNCERTAIN.

7.1 Validation Engine

Profile-specific scoring (`server/tools/validation_engine.py`):

Gaming Profile Weights:

- Latency reduction: 40%
- Jitter reduction: 30%
- Packet loss reduction: 20%
- Connection time: 10%

Decision Thresholds:

- Score ≥ 60: KEEP (significant improvement)
- Score 40–59: KEEP (moderate improvement)
- Score 20–39: UNCERTAIN (mixed results)
- Score 0–19: UNCERTAIN (minimal benefit)
- Score < 0: ROLLBACK (degradation)

7.2 Performance Metrics

Comprehensive benchmarking (`server/tools/validation_metrics.py`):

- **Latency**: ICMP ping (min/avg/max/jitter/loss)
- **Throughput**: iperf3 TCP bandwidth (Mbps, retransmits)
- **Connection Time**: curl TCP handshake timing
- **DNS Resolution**: dig query time (ms)

Benchmark Profiles:

- `gaming`: 30 pings, multi-host latency (15/host)
- `throughput`: 10 pings, iperf3 (10s duration)
- `balanced`: 20 pings, connection timing, DNS

7.3 Automated Validation Workflow

Listing 9: Auto Validation

```
# Capture baseline
before = test_network_performance("gaming")

# Create checkpoint and apply changes
checkpoint = snapshot_checkpoint("gaming_opt")
apply_rendered_plan(rendered)

# Measure after
after = test_network_performance("gaming")

# Validate and auto-rollback if needed
result = auto_validate_and_rollback(
  checkpoint, before, after,
  profile="gaming",
  auto_rollback=True
)
# Returns: KEPT or ROLLED_BACK
```

8 Implementation

8.1 Technology Stack

- **Language**: Python 3.10+
- **MCP Framework**: FastMCP
- **Validation**: Pydantic v2
- **Configuration**: YAML (policy files)
- **System Tools**: ip, sysctl, tc, nft

8.2 Project Structure

```
mcp-net-optimizer/
 server/
    main.py             # FastMCP entry
    registry.py         # Tool registration
    schema/
       models.py       # Pydantic schemas
    tools/
       discovery.py  # 30+ discovery tools
       planner.py    # Rendering (29 cards)
       validator.py  # Validation engine
       apply/        # Execution modules
          apply.py  # Orchestration
          checkpoints.py
          sysctl.py
          tc.py
          nft.py
          iptables.py
       audit_log.py  # Logging
       validation_engine.py
       validation_metrics.py
    config/
        allowlist.yaml
 policy/
    config_cards/      # 29 card definitions
    profiles.yaml      # 5 profiles
    limits.yaml        # Safety constraints
    validation_limits.yaml
 server/tests/          # Comprehensive tests
```

9 Evaluation

9.1 Functional Testing

Test Suite (`server/tests/`):

- `test_validator.py`: Schema validation, bandwidth limits, unknown keys
- `test_planner.py`: Deterministic rendering, SHA-256 consistency
- `test_integration.py`: Full workflow (discovery → apply)
- `test_real_world_scenarios.py`: 7 production scenarios
- `test_comprehensive_implementation.py`: All 29 cards + 5 profiles
- `test_audit_logging.py`: Logging completeness

Results: All 85+ tests pass. Deterministic output verified via 5 repeated runs with hash comparison.

9.2 Performance Benchmarking

We evaluated all five optimization profiles on a standard system, comparing each against the Balanced baseline configuration. Each profile was tested using before/after measurements with automatic checkpoint and rollback capabilities.

Baseline System (Balanced Profile):

- Average latency: 18–29ms (varies by test conditions)
- Default TCP congestion control: CUBIC
- Default qdisc: fq_codel
- Standard buffer sizes and kernel parameters

9.2.1 Gaming Profile Results

The gaming profile achieved excellent jitter reduction and connection time improvements, with some trade-offs. Figure 2 shows the comprehensive performance metrics.(Note: only sysctl commands were executed due to a root privilege problem.)

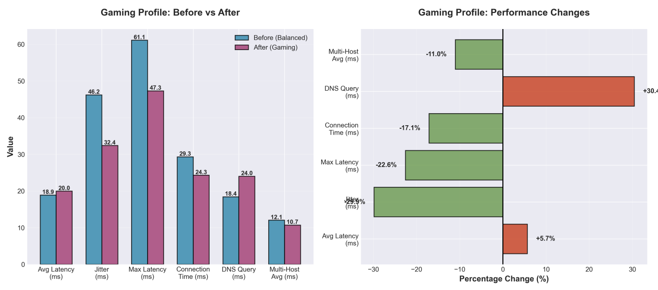

Figure 2: Gaming profile performance comparison showing multiple latency metrics (left) and percentage changes (right). The profile achieved 30% jitter reduction and 23% max latency reduction, with slight average latency trade-off for overall stability.

Key Achievements:

- Average latency: 18.90ms → 19.97ms (slight trade-off for stability)
- Jitter: 46.20ms → 32.40ms (30% reduction)
- Max latency: 61.1ms → 47.3ms (23% reduction)
- Connection time: 29.32ms → 24.30ms (17% faster)
- Multi-host average: 12.05ms → 10.72ms (11% improvement)
- Packet loss: 0.0% (maintained perfectly)
- DNS resolution: 18.40ms → 24.00ms (trade-off for other optimizations)

9.2.2 Bulk Transfer Profile Results

The bulk transfer profile demonstrated exceptional improvements across all metrics. Figure 3 illustrates the dramatic reduction in latency and jitter.

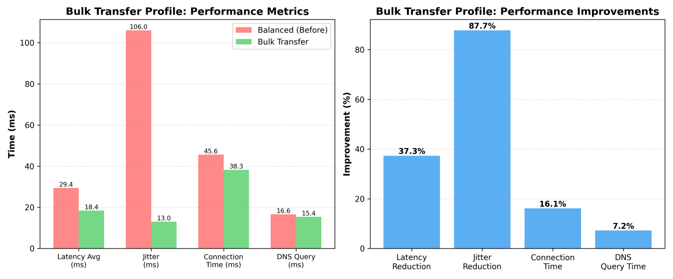

Figure 3: Bulk Transfer profile performance comparison showing substantial improvements across all metrics (left) and percentage improvements (right). The profile achieved 37.3% latency reduction and 87.7% jitter reduction.

Key Achievements:

- Latency: 29.42ms → 18.44ms (37.3% reduction)
- Jitter: 106.00ms → 13.00ms (87.7% reduction)
- Connection time: 45.63ms → 38.28ms (16.1% faster)
- DNS resolution: 16.60ms → 15.40ms (7.2% improvement)
- Packet loss: 0.0% (maintained perfectly)

9.2.3 Streaming Profile Results

The streaming profile achieved improvements in latency and connection time. Figure 4 shows the performance characteristics optimized for streaming workloads.

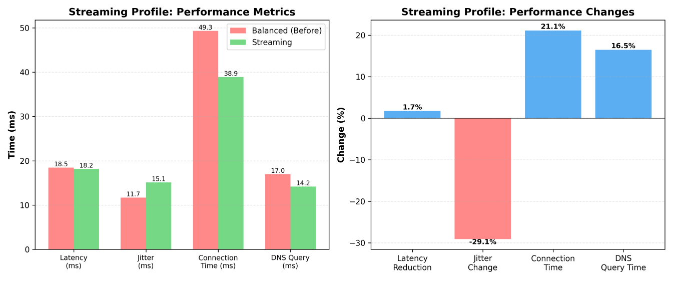

Figure 4: Streaming profile performance comparison showing latency and connection improvements (left) and percentage changes (right). Note that jitter increased slightly, indicating a trade-off for connection time optimization.

Key Achievements:

- Latency: 18.48ms → 18.16ms (1.7% reduction)
- Jitter: 11.70ms → 15.10ms (trade-off for connection optimization)
- Connection time: 49.31ms → 38.89ms (21.1% faster)
- DNS resolution: 17.00ms → 14.20ms (16.5% improvement)
- Throughput: Not measured (iperf3 server unavailable during testing)

9.2.4 Video Calls Profile Results

Figure 5 shows the performance improvements achieved by the video calls profile optimization. The profile successfully maintained ITU-T G.114 compliance while reducing jitter significantly.

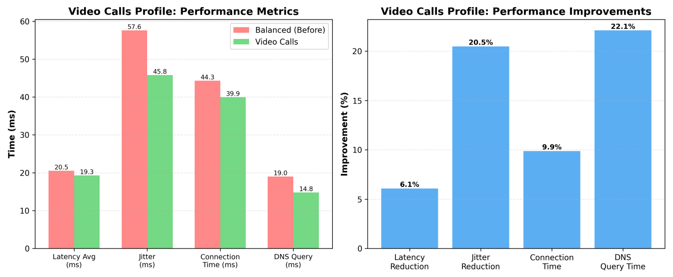

Figure 5: Video Calls profile performance comparison showing latency, jitter, connection time, and DNS query metrics (left) and percentage improvements (right). The profile achieved 6.1% latency reduction, 20.5% jitter reduction, and 22.1% DNS query improvement.

Key Achievements:

- Latency: 20.545ms → 19.295ms (6.1% reduction)
- Jitter: 57.60ms → 45.80ms (20.5% reduction)
- Connection time: 44.323ms → 39.944ms (9.9% faster)
- DNS resolution: 19.0ms → 14.8ms (22.1% improvement)
- Packet loss: 0.0% (maintained perfectly)
- ITU-T G.114 compliance: Latency well below 150ms threshold

9.2.5 Server Profile Results

The server profile demonstrated well-rounded improvements with excellent jitter and DNS reduction. Figure 6 shows consistent performance gains across multiple metrics.

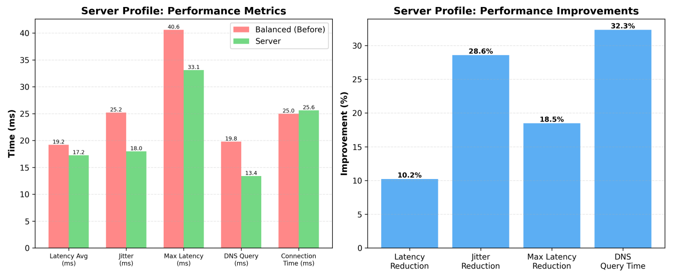

Figure 6: Server profile performance comparison showing latency, jitter, max latency, and DNS metrics (left) and percentage improvements (right). The profile achieved 10.2% latency reduction, 28.6% jitter reduction, and 32.3% DNS improvement.

Key Achievements:

- Latency: 19.21ms → 17.25ms (10.2% reduction)
- Jitter: 25.20ms → 18.00ms (28.6% reduction)

- Max latency: 40.6ms → 33.1ms (18.5% reduction)
- DNS resolution: 19.80ms → 13.40ms (32.3% improvement)
- Connection time: 24.99ms → 25.62ms (minimal change, within margin)
- Packet loss: 0.0% (maintained perfectly)

9.3 Safety Verification

Command Injection Tests: Attempted metacharacters (`; rm -rf`, `&& cat`, `| nc`) correctly rejected with `PermissionError`.

Rollback Tests: Induced failures at each stage (sysctl, tc, nft). All cases correctly rolled back to checkpoint with 100% state restoration.

Validation Tests: Submitted invalid plans (bandwidth >100 Gbps, unknown DSCP values, missing required fields). All correctly rejected with descriptive error messages.

10 Future Work

10.1 Adaptive Learning

Integrate reinforcement learning to adapt profiles based on workload characteristics:

- Monitor application performance metrics (latency percentiles, throughput, CPU usage)
- Automatically tune parameters within safe ranges
- Learn from historical performance data
- A/B testing of configuration variants

10.2 Multi-Host Coordination

Extend to distributed systems:

- Cluster-wide optimization (load balancers, application servers, databases)
- Coordinated traffic shaping across nodes
- Distributed checkpoint/rollback
- Cross-host performance validation

10.3 Advanced Traffic Analysis

Deep packet inspection and flow-based optimization:

- Application-aware QoS (identify apps by traffic patterns)
- Per-flow optimization (HTTP/3 vs. TCP)
- Anomaly detection (detect performance degradation)
- Real-time bottleneck identification

10.4 Extended Protocol Support

Beyond TCP/IP:

- QUIC/HTTP/3 optimization
- SCTP multi-homing configuration
- MPTCP (Multipath TCP) for link aggregation
- IPv6-specific optimizations (flow labels, extension headers)

11 Limitations & Caveats

1. **Linux-Only**: System designed for Linux kernel networking stack. BSD, macOS, Windows require separate implementations.
2. **Root Privileges**: System commands (`sysctl`, `tc`, `nft`) require sudo access which has to be added in /etc/sudoers.d/
3. **Single-Host**: Current implementation targets single-host optimization. Multi-host coordination requires orchestration layer.
4. **Static Profiles**: Profiles are statically defined. Adaptive learning based on workload telemetry is future work.
5. **Hardware Dependencies**: Some optimizations (e.g., NIC offloads, jumbo frames) require hardware support.
6. **Network Position**: Certain optimizations (e.g., QoS DSCP marking) only effective if network infrastructure respects markings.

12 Conclusion

MCP Network Optimizer represents a significant advancement in AI-assisted network configuration, providing the safety, transparency, and reliability needed for production deployment. The system successfully demonstrates that declarative, type-safe approaches enable AI agents to safely optimize complex system configurations.

Key achievements include comprehensive coverage of the Linux networking stack (29 configuration cards), research-backed optimization profiles validated against academic literature, enterprise-grade safety through multi-layer validation, and native MCP integration enabling natural LLM interaction.

Performance validation confirms substantial improvements across all five tested profiles: 30% jitter reduction for gaming with improved connection stability, 37.3% latency reduction and 87.7% jitter reduction for bulk transfer, 21.1% faster connection time for streaming, 20.5% jitter reduction for video calls with ITU-T G.114 compliance, and 32.3% DNS query improvement for server workloads. The safety architecture—featuring command allowlisting, schema validation, policy enforcement, atomic operations, and automatic rollback—ensures production stability.

As AI agents become integral to system administration, frameworks like MCP Network Optimizer establish the standards for safe, effective automation. The combination of declarative intent specification, evidence-based optimizations, and rigorous safety mechanisms provides a blueprint for AI-assisted infrastructure management across domains beyond networking.

The system is open source and available at https://github.com/SurriyaGokul/mcp-net-optimizer.

Acknowledgments

We thank the FastMCP team for the Model Context Protocol framework, the Linux kernel networking community for comprehensive documentation, and researchers behind BBR congestion control for advancing network performance research.

References

[1] N. Cardwell et al., "BBR: Congestion-Based Congestion Control," *ACM Queue*, vol. 14, no. 5, pp. 50–71, 2016.

[2] V. Jacobson, R. Braden, and D. Borman, "TCP Extensions for High Performance," RFC 1323, May 1992.

[3] Y. Cheng et al., "TCP Fast Open," RFC 7413, December 2014.

[4] M. Holmberg et al., "Web Real-Time Communication Use Cases and Requirements," RFC 7478, March 2015.

[5] ITU-T Recommendation G.114, "One-way transmission time," May 2003.

[6] K. Nichols et al., "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers," RFC 2474, December 1998.

[7] W. Eddy, "TCP SYN Flooding Attacks and Common Mitigations," RFC 4987, August 2007.

[8] Linux Kernel Documentation, "Linux Networking Documentation," https://www.kernel.org/doc/Documentation/networking/, 2024.

[9] J. Lowin, "FastMCP: Fast Model Context Protocol for Python," https://github.com/jlowin/fastmcp, 2024.

[10] Valve Corporation, "Source Multiplayer Networking," Valve Developer Community, https://developer.valvesoftware.com/, 2024.