MCP SERVER VALIDATOR

Requirements Specification

A Protocol Conformance and Security Validation Tool

Version	1.0
Date	January 16, 2026
Authors	Paola Di Maio & Claude (Anthropic)
Status	Final Draft
Affiliation	W3C Web AI IG

1. Executive Summary

1.1 Purpose

The MCP Server Validator is a comprehensive tool for verifying that Model Context Protocol (MCP) server implementations comply with protocol specifications, security best practices, and interoperability requirements. It addresses a critical gap in the MCP ecosystem by providing automated quality assurance for server development.

1.2 Background: What MCP Is

Model Context Protocol (MCP) is an emerging open standard defining how applications expose contextual data and tool functionality to large language models (LLMs) in a uniform, secure, extensible way. It grew out of the need to standardize LLM integrations with external systems, much the same way that Language Server Protocol (LSP) standardized tooling for editors.

Key characteristics of MCP:

- Open protocol described with normative RFC-style language (RFC 2119/8174) and formal JSON-RPC schemas
- Communication based on JSON-RPC 2.0 messages over supported transports (stdio, HTTP + SSE)
- Architecture: host ↔ client ↔ server, with servers exposing resources, tools, and prompts
- Rapidly gaining adoption for connecting LLMs to applications, IDEs, databases, and APIs

1.3 Problem Statement

Currently, MCP server developers must manually ensure their implementations comply with protocol message formats (JSON-RPC), tool naming conventions, schema definitions (Zod/Pydantic), security patterns, and best practices. This manual process is error-prone, time-consuming, and creates inconsistent quality across the ecosystem. Academic research has documented security risks including tool poisoning and attack surfaces in real MCP implementations.

1.4 Solution Overview

This validator provides automated conformance checking through static analysis (code inspection without execution) and optional runtime testing (simulated client interactions). It follows the established pattern used by JSON-RPC conformance suites, LSP test harnesses, OpenAPI/Swagger validators, and gRPC compliance tools: formal specification → test harness → validator reporting.

2. Scope & Stakeholders

2.1 In Scope

- Protocol conformance validation (JSON-RPC 2.0, MCP message schemas)
- Feature compliance validation (resources, tools, prompts)
- Connection lifecycle and transport testing (stdio, HTTP + SSE)
- Security pattern detection and warnings
- Naming convention enforcement
- Schema validation (Zod for TypeScript, Pydantic for Python)

2.2 Out of Scope (Version 1.0)

- Business logic semantics validation
- Deep network fuzzing
- Custom rule plugins (planned for v2.0)
- Docker-based sandboxing
- Automatic code fixes (suggest only)
- Integration testing with real LLM clients

2.3 Stakeholders

- **MCP Server Developers:** Primary users validating their implementations
- **MCP Client Developers:** Ensuring interoperability with validated servers
- **QA/Test Engineers:** Integrating validation into CI/CD pipelines
- **Security Reviewers:** Auditing MCP servers for vulnerabilities
- **MCP Ecosystem Maintainers:** Establishing quality baselines

3. Prior Art & Normative References

3.1 Normative References

- **[MCP-SPEC]** Model Context Protocol Specification (modelcontextprotocol.io/specification/latest)
- **[RFC2119]** Key words for use in RFCs to Indicate Requirement Levels
- **[RFC8174]** Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words
- **[JSON-RPC]** JSON-RPC 2.0 Specification

3.2 Informative References

- **[SECURITY-ANALYSIS]** arXiv:2506.02040 - MCP Security Risk Analysis
- **[ECOSYSTEM-HEALTH]** arXiv:2506.13538 - MCP Server Ecosystem Health Study

3.3 Comparable Validators in Other Ecosystems

This validator follows established patterns from:

- **Language Server Protocol (LSP):** Reference implementations with capability negotiation tests
- **OpenAPI/Swagger Validators:** Checking server implementations against specifications
- **gRPC Compliance Tools:** Ensuring RPC interfaces match formal proto descriptions
- **JSON-RPC Conformance Suites:** Protocol-level message format testing

4. Functional Requirements

4.1 Protocol Conformance

REQ-PROTOCOL-001: JSON-RPC Format

The validator MUST confirm that every message received or sent by the MCP server follows JSON-RPC 2.0 format (IDs, structure, error codes).

- Priority: MUST HAVE
- Acceptance: All JSON-RPC messages parse correctly; invalid messages are flagged

REQ-PROTOCOL-002: MCP Schema Compliance

The validator SHOULD validate that server Requests, Responses, and Notifications adhere to schema definitions in the official MCP specification JSON schemas.

- Priority: SHOULD HAVE
- Acceptance: Schema validation against official MCP JSON schemas passes

REQ-PROTOCOL-003: Capability Negotiation

The validator MUST verify capability negotiation behavior: server announces supported features, client handles capability feedback correctly, and negotiation follows the MCP lifecycle.

- Priority: MUST HAVE
- Acceptance: Capability document is well-formed; features match declared capabilities

REQ-PROTOCOL-004: Role-Based Message Flow

The validator SHOULD test role-based message flows (host → client → server patterns) to ensure correct message routing.

- Priority: SHOULD HAVE
- Acceptance: Message flow patterns match MCP architecture specification

4.2 Feature Compliance

REQ-FEATURE-001: Resource Validation

The validator MUST check that server-declared resources match required MCP resource metadata (URI templates, types, descriptions).

- Priority: MUST HAVE
- Acceptance: All resources have valid URI templates and type declarations

REQ-FEATURE-002: Tool Discovery and Invocation

The validator MUST test that tools are discoverable via tools/list and invocable via tools/call with correct parameter handling and error reporting.

- Priority: MUST HAVE

- Acceptance: tools/list returns valid tool array; tools/call executes with proper responses

REQ-FEATURE-003: Prompt Template Validation

The validator SHOULD verify that prompt templates adhere to expected structure and metadata fields per specification.

- Priority: SHOULD HAVE
- Acceptance: Prompt templates have required fields and valid argument schemas

4.3 Static Analysis Requirements

REQ-STATIC-001: Code Parsing

The validator MUST parse server source code without execution:

- Parse Python server files (.py) using AST module
- Parse TypeScript server files (.ts, .tsx) using TypeScript parser
- Extract server metadata (name, version, description)
- Extract all tool, resource, and prompt definitions
- Complete analysis in <5 seconds for typical servers

REQ-STATIC-002: Tool Naming Convention

The validator MUST verify tools follow MCP naming conventions:

- Format: {service}_{action}_{resource} (e.g., github_list_repos)
- Pattern: ^[a-z]+_[a-z]+_[a-z]+$
- Detect overly generic names (e.g., "get_data", "tool1")
- Suggest corrected names for violations

REQ-STATIC-003: Schema Validation

The validator MUST verify input/output schemas:

- Detect presence of input schema for each tool
- Verify schema type (Zod for TypeScript, Pydantic for Python)
- Check all parameters have types defined
- Verify required vs optional fields marked correctly
- Check for descriptive field documentation

REQ-STATIC-004: Security Pattern Detection

The validator MUST detect common security issues:

- Hardcoded credentials (API keys, passwords, tokens)
- Missing authentication patterns
- Dangerous function usage (eval, exec, subprocess without sanitization)
- Path traversal vulnerabilities
- Missing destructiveHint annotations on destructive operations

4.4 Connection Lifecycle & Transport

REQ-CONN-001: Lifecycle Simulation

The validator MUST simulate connection lifecycle (connect, capability negotiation, disconnect) for all declared transports the server supports (stdio, HTTP + SSE).

- Priority: MUST HAVE (for runtime mode)
- Acceptance: Full lifecycle completes without errors on supported transports

REQ-CONN-002: Session Semantics

The validator MUST ensure stateful session semantics are properly maintained (ID reuse, sequence ordering).

- Priority: MUST HAVE
- Acceptance: Session IDs are unique; message ordering is correct

4.5 Error & Edge Behavior

REQ-ERROR-001: Malformed Request Handling

The validator MUST assert that malformed requests receive valid JSON-RPC error replies with correct error codes.

- Priority: MUST HAVE
- Acceptance: Invalid JSON returns -32700; invalid request returns -32600

REQ-ERROR-002: Unsupported Method Handling

The validator MUST assert that unsupported methods return proper JSON-RPC errors with code -32601.

- Priority: MUST HAVE
- Acceptance: Unknown methods return standardized error response

REQ-ERROR-003: Tool Error Responses

The validator MUST verify that tool execution errors include proper error codes in the response structure.

- Priority: MUST HAVE
- Acceptance: Tool errors contain {"error": {"code": ..., "message": ...}}

5. Non-Functional Requirements

5.1 Security

SEC-NFR-001: Sandboxed Execution

The validator MUST be safe to run against untrusted server code. Static analysis uses AST parsing only (never eval/exec). Runtime testing requires explicit opt-in via --allow-runtime flag.

SEC-NFR-002: Configurable Security Policies

The tool SHOULD provide configurable security policies (e.g., require TLS/HTTPS, enforce strict auth, timeout limits).

SEC-NFR-003: Timeout Protection

Runtime tests MUST have timeout protection (30 seconds per test by default) to prevent hanging on malicious or buggy servers.

5.2 Performance

PERF-NFR-001: CI/CD Integration

Validator tests SHOULD be parallelizable to support CI/CD integration (e.g., GitHub Actions, GitLab CI).

PERF-NFR-002: Analysis Speed

Static analysis MUST complete in <5 seconds for 95% of typical MCP servers.

5.3 Usability

USE-NFR-001: Machine-Readable Output

Outputs MUST be available in machine-readable formats (JSON, YAML) for automation integration.

USE-NFR-002: Human-Readable Output

Outputs MUST also be available in human-readable formats (console text, HTML report) with actionable suggestions.

6. Interfaces & Reporting

6.1 Command Line Interface

The validator MUST provide a CLI with the following capabilities:

- Basic validation: mcp-validator validate server.py
- Analysis level: --level=static|runtime|both
- Output format: --format=console|json|html
- Severity filter: --min-severity=critical|error|warning|info
- Rule selection: --rules=naming,security or --exclude=runtime
- Config file: --config=.mcp-validator.yaml
- Exit codes: 0 (pass), 1 (fail), 2 (error)

6.2 Report Structure

Reports MUST include:

- **Metadata:** file, language, validator version, timestamp
- **Summary:** total checks, passed, failed, warnings, score, compliance level
- **Issues:** severity, category, rule ID, message, location (file/line/column), suggestion, example fix
- **Statistics:** tools analyzed, tools compliant, security issues, best practice violations

6.3 Compliance Levels

- **EXCELLENT (90-100%):** Full specification compliance, robust error handling
- **GOOD (70-89%):** Strong compliance with minor issues
- **MODERATE (50-69%):** Functional but with notable deviations
- **POOR (25-49%):** Significant compliance issues
- **CRITICAL (<25%):** Major protocol violations or security risks

7. Security Considerations

MCP servers may expose powerful operations. Based on academic research documenting security risks in real MCP implementations (arXiv:2506.02040), the validator SHOULD check for:

7.1 Excessive Functionality Exposure

Servers exposing too much functionality without capability limits. The validator should warn when servers declare many tools without appropriate scoping.

7.2 Missing Privilege Separation

Tool calls that lack privilege separation. Destructive operations (delete, modify, execute) should have appropriate annotations and confirmation requirements.

7.3 Unauthorized Access Patterns

Servers lacking authentication requirements where expected. The validator should flag servers that access sensitive resources without auth configuration.

7.4 Tool Poisoning Vectors

Patterns that could enable tool poisoning attacks, including tools with overly permissive input schemas or insufficient output sanitization.

8. Test Case Taxonomy

Test Category	Core Checks
JSON-RPC Schema	Message shapes follow JSON-RPC 2.0 specification
Capability Negotiation	Supported server features match MCP spec
Resource Discovery	Server returns correct resource lists with valid URIs
Tool Invocation	Tools callable and return expected typed results
Tool Naming	Names follow service_action_resource convention
Schema Validation	Input schemas properly typed (Zod/Pydantic)
Error Handling	Invalid messages return compliant JSON-RPC errors
Security Patterns	No hardcoded secrets; proper auth patterns

9. Technical Requirements

9.1 Language Support

Python Server Validation (Priority 1):

- Parse Python AST using built-in ast module
- Detect FastMCP usage patterns
- Validate Pydantic models
- Support Python 3.8+

TypeScript Server Validation (Priority 2):

- Parse TypeScript AST using @typescript-eslint/parser
- Detect MCP SDK usage patterns
- Validate Zod schemas
- Support TypeScript 4.5+

9.2 Dependencies

Python Dependencies:

- ast (built-in) - Python parsing
- pyyaml - Configuration and rule files
- click - CLI framework
- colorama - Colored output
- jinja2 - HTML report generation

9.3 Rule Definition Format

Rules MUST be defined in YAML with the following structure: rule ID, name, severity (CRITICAL/ERROR/WARNING/INFO), category, description, patterns, examples (valid/invalid), and suggestions.

10. Success Criteria

10.1 Adoption Metrics

- Used by 50+ MCP server developers within 3 months
- Integrated into 10+ CI/CD pipelines
- Referenced in MCP official documentation
- 100+ stars on GitHub

10.2 Quality Metrics

- 90%+ accuracy on known-good servers (minimal false positives)
- 95%+ detection rate on known-bad servers (minimal false negatives)
- <5% false positive rate on security checks
- User satisfaction >4.5/5 in surveys

10.3 Technical Metrics

- 100% test coverage on core validation logic
- <5 second static analysis for 95% of servers
- Zero crashes on fuzzing with 1000 random inputs
- All CLI tests passing on Linux, macOS, Windows

11. Version Roadmap

11.1 Version 1.0 (Current Scope)

- Static analysis for Python MCP servers
- Tool naming convention validation
- Schema validation (Pydantic)
- Security pattern detection
- CLI interface with JSON/console output
- Basic runtime testing (opt-in)

11.2 Version 1.5 (Near-term)

- TypeScript server support
- HTML report generation
- Resource and prompt validation
- GitHub Actions integration

11.3 Version 2.0 (Future)

- Custom rule plugin system
- Automatic code fixes
- VS Code extension
- Web-based validator service
- Docker-based sandboxing for runtime tests

Appendix A: MCP Naming Convention Rules

Format: {service}_{action}_{resource}

- **service:** lowercase, the external service (github, slack, etc.)
- **action:** lowercase verb (list, create, update, delete, get, search)
- **resource:** lowercase noun (repos, messages, users, issues)

Valid Examples:

- github_list_repos
- slack_send_message
- jira_create_issue
- stripe_get_customer

Invalid Examples:

- listRepos (camelCase)
- get_data (too generic)
- tool1 (not descriptive)
- list_github_repos (wrong order)

Appendix B: Approval and Sign-off

Reviewed by	Paola Di Maio
Co-authored by	Claude (Anthropic)
Approved for development	[ ] Yes [ ] No
Date	_______________

END OF REQUIREMENTS DOCUMENT