Extending MCP

Writing Custom Protocol Extensions Without Breaking Compatibility

Hello!

- **Google Developer Expert**
  GCP
- **DevOps Ambassador**
  Institute Ambassador and Organizer
- **Community Leader**
  Graph DB Pune & Kong Champion Organizer
- **Multi-Cloud Advocate**
  Calico Big Cats, CDF & open-appsec AMB

Saurabh Mishra

Connect with me

linkedin.com/connectsaurabhmishra
medium.com/@connectsaurabhmishra

Agenda

What is MCP?

Model Context Protocol (MCP)

Definition

An open protocol for connecting AI models with external tools, data sources, and applications, standardizing communication between AI clients and servers.

Key Benefits

- Interoperability & Reusability
- Tool discoverability
- Standardized messaging
- Scalable integrations

Typical Components

• MCP Client & Server
• Tools, Resources & Transport Layer

Why Extend MCP?

Business & Technical Drivers

Common Reasons for Extensions

- Custom authentication workflows
- Organization-specific metadata
- Advanced telemetry
- Proprietary tool capabilities
- Streaming enhancements
- Specialized resource handling

Goal

Add functionality **without breaking** existing clients or servers.

Main Challenge

Maintaining interoperability across mixed versions and implementations.

Extending MCP Safely

Compatibility Challenges

Risks of Poorly Designed Extensions

Common Problems

- Breaking existing clients
- Unsupported message formats
- Version fragmentation
- Vendor lock-in
- Unexpected runtime failures
- Security inconsistencies

Example Failure Scenario

A server adds required fields that older clients cannot parse.

Result

- Connection failures
- Lost functionality
- Reduced ecosystem trust

Core Design Principles

Principles for Safe Protocol Extensions

1. Backward Compatibility
Older clients should continue functioning.

2. Forward Compatibility
Unknown fields should be safely ignored.

3. Loose Coupling
Extensions should remain modular.

4. Discoverability
Capabilities should be explicitly advertised.

5. Graceful Degradation
Fallback behavior must exist.

Extension Architecture

Recommended Extension Model: Namespaced Extensions

Benefits

- Prevents naming collisions
- Improves readability
- Simplifies governance
- Enables modular adoption

Recommendation

Use vendor or domain-based namespaces.

Example Structure

```
"extensions": {
"acme.telemetry": {
"traceId": "12345"
}
}
```

Capability Negotiation

Detecting Supported Features

Best Practice

Advertise capabilities during initialization.

Example Flow

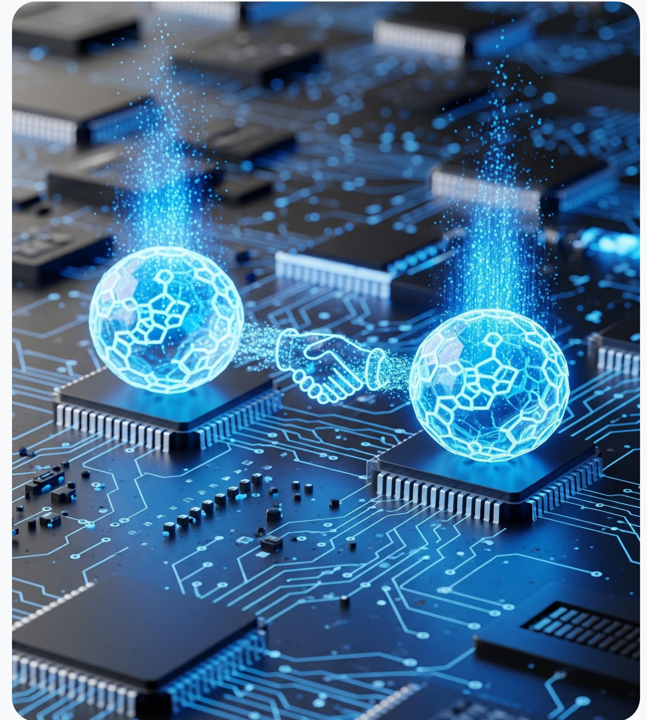
- Client connects
- Server advertises supported extensions
- Client enables compatible features
- Unsupported features are skipped

Sample Capability Object

```
{
  "capabilities": {
    "acme.telemetry": true,
    "streaming.v2": false
  }
}
```

Key Benefit

Avoids hard failures.



Versioning Strategies

Managing Change Safely

Recommended Approaches

- Semantic Versioning (SemVer)
- Feature Flags
- Extension-Specific Versions
- Deprecation Windows

Example Configuration

```
"extensions": {
"acme.telemetry": {
"version": "1.2.0"
}
}
```

Avoid

- Breaking changes in minor releases
- Silent protocol changes

Graceful Fallback Design

Ensuring Reliable Behavior

Strategy

If extension is unsupported:

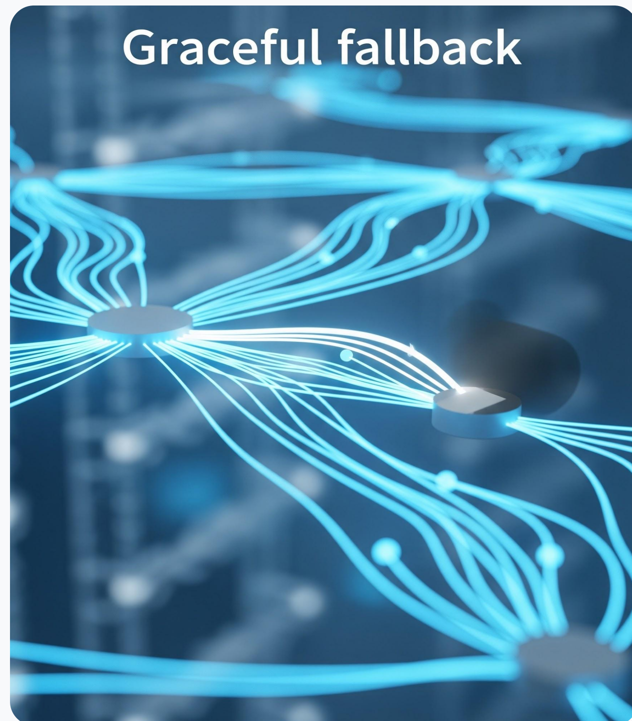
- Ignore optional fields
- Use default protocol behavior
- Return warnings instead of failures

Example

```
{
"warning": "Telemetry extension unsupported"
}
```

Outcome

System remains functional even with partial support.



Error Handling Best Practices

Build Predictable Protocol Behavior

Recommendations

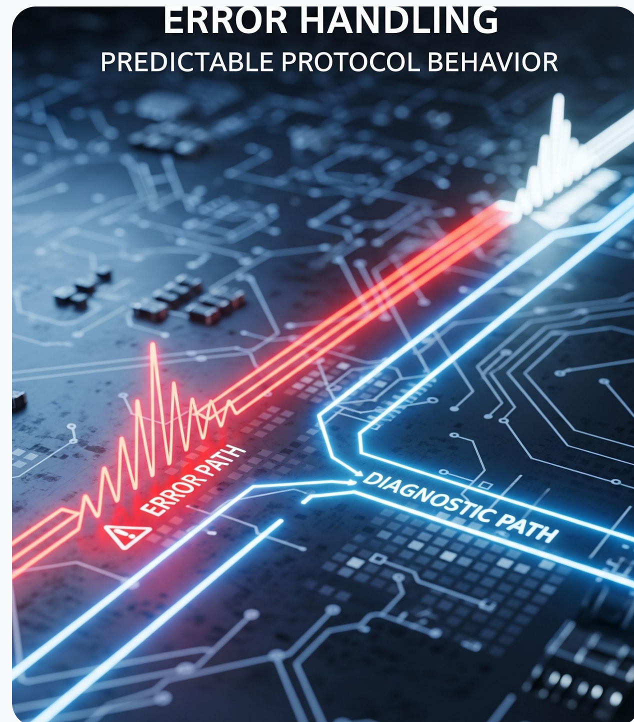
- Standardized error codes
- Human-readable messages
- Retry guidance
- Extension-specific diagnostics

Example Error

```
{
"error": {
"code": "EXT_UNSUPPORTED",
"message": "Extension not supported"
}
}
```

Goal

Reduce debugging complexity.



Security Considerations

Secure Extension Development

Risks Introduced by Extensions

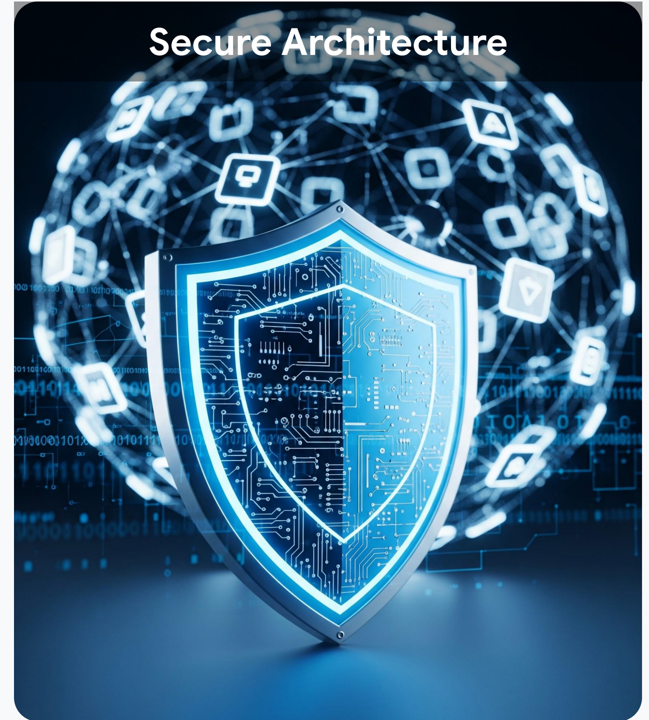
- Expanded attack surface
- Unauthorized capabilities
- Data leakage
- Injection vulnerabilities

Mitigation Strategies

- Validate all extension payloads
- Apply schema enforcement
- Authenticate capability requests
- Limit privileged operations
- Audit extension activity

Principle

"Security must evolve with extensibility."



Testing & Validation

Ensuring Cross-Version Stability

Recommended Testing Layers

- Unit Testing
- Contract Testing
- Schema Validation
- Compatibility Testing
- Fuzz Testing
- Integration Testing

Real-World Example

MCP Extensions in Enterprise Platforms

Example 1 — Neo4j Graph Extensions

Using MCP with Neo4j graph databases.

Use Cases & Benefits

- AI agents querying knowledge graphs
- Relationship-aware retrieval & enrichment
- Advanced semantic search

```
"neo4j.graph": {
"queryLanguage": "Cypher",
"graphTraversal": true
}
```

Example 2 — Google Workspace

Integration with Google document & communication services.

Capabilities

- Drive, Gmail, and Calendar context
- Real-time collaborative AI workflows
- Safe fallback for older clients

```
"capabilities": {
"google.workspace": true,
"google.docs.context": true
}
```

Result: Agents gain deep enterprise productivity context.

Neo4j & Google Cloud Integration

Comprehensive Agent Memory & Cloud Services

Integration Features

- **Vertex AI Embeddings:** Generate high-quality embeddings using Google's models.
- **Google ADK Support:** Seamless integration with the Agent Development Kit.
- **MCP Server:** Dedicated server for Cloud API Registry connectivity.
- **Agent Memory:** Provides relationship-aware retrieval and storage.

Technical Overview

Neo4j Agent Memory enables AI agents to utilize Google Cloud's infrastructure for enhanced knowledge graph querying and contextual reasoning.

Managed MCP Overview

Enterprise-Grade Model Context Protocol for Google Cloud

Key Platform Benefits

- **No Infrastructure:** Fully managed service with no local servers to host or scale.
- **Unified Connectivity:** Connect agents to BigQuery, Databases, and Maps via a single endpoint.
- **Native Security:** Integrated with IAM, VPC Service Controls, and Model Armor for threat prevention.
- **Observability:** Built-in audit logging and tracking for every agent-driven query.
- **Dialect Aware:** Handles SQL specific syntax for PostgreSQL, MySQL, and SQL Server automatically.

Documentation & Access

Visit the official portal for setup guides and API references:

`docs.cloud.google.com/mcp/overview`

Best Practices Checklist

Do's and Don'ts for Protocol Extensions

Do

- Use optional fields
- Namespace extensions
- Support graceful fallback
- Publish schemas
- Document version policies
- Test compatibility continuously

Don't

- Modify core behavior unexpectedly
- Require extensions by default
- Break old payload structures
- Ignore security validation

Industry Adoption Examples

Companies & Platforms Exploring MCP-Style Integrations

Enterprise Ecosystem Examples

- **Neo4j** — Graph-powered AI context systems
- **Google** — Workspace and contextual AI integrations
- **OpenAI** — AI tool interoperability
- **Anthropic** — MCP ecosystem support

Growing Trends

- Enterprise context orchestration
- Multi-agent interoperability
- Retrieval-Augmented Generation (RAG)
- Graph-native AI architectures
- Cross-platform AI assistants

Future Outlook

Where MCP Extensions Are Heading

Emerging Trends

- Dynamic capability negotiation
- AI-native protocol optimization
- Standard extension registries
- Cross-vendor interoperability
- Schema-driven automation

Long-Term Vision

A flexible ecosystem where innovation happens without breaking compatibility.

Key Takeaways

Summary & Best Practices

Remember

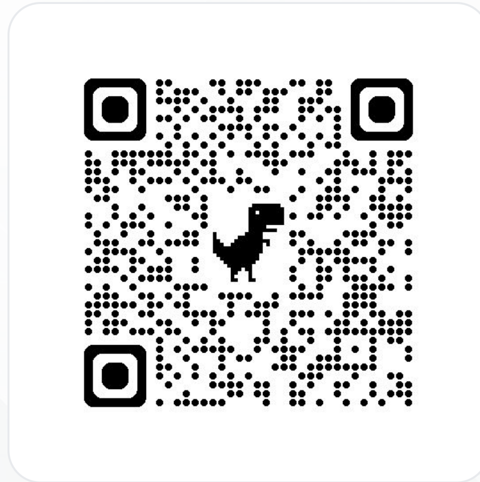
- Extensions should be optional and discoverable.
- Compatibility must be designed intentionally.
- Capability negotiation is essential.
- Versioning and fallback mechanisms reduce risk.
- Security and testing are critical.

Final Thought

A well-designed extension ecosystem accelerates innovation while preserving stability.

Thank You!

Questions & Discussion



Contact: skm.jss@gmail.com