# MCP Interview Preparation Handbook (Detailed)

Generated: 2025-09-11 07:06 UTC

# MCP & FastMCP — Interview Preparation Handbook (Detailed)

**Generated:** 2025-09-11 06:50 UTC

---

## Part 1 — Introduction

### *Why LLMs need structured protocols*

Large Language Models are exceptional at generating and synthesizing text, but they lack **built-in awareness** of enterprise systems, data formats, and safe side effects. To use LLMs productively in real applications we need a structured way to expose:
- **Contextual data** (customer profiles, logs, knowledge bases)
- **Actions** (create orders, update records, call external APIs)
- **Governance** (consistent prompts, access control, auditing)

MCP (Model Context Protocol) is a purpose-built protocol that standardizes how external systems present **resources**, **tools**, and **prompts** to LLMs. It removes ad-hoc integrations and provides discoverability, typing, and safety primitives that help LLM-driven agents behave predictably.

**Suggested Interview Answer (short):** MCP standardizes LLM access to data and actions via resources, tools, and prompts — making integrations safer and more discoverable than ad-hoc APIs.

---

## Part 2 — Core Concepts

### *2.1 Resources — Definition & Usage*

**What it is:** Resources expose read-only data to LLMs. They behave like queryable views into your systems and are intended for consumption (not modification).

**Why it matters:** LLMs produce higher-quality outputs when given accurate, structured context. Resources give LLMs access to authoritative data without granting direct DB access or unmediated API keys.

**How to hook it up (example):**

```
@mcp.resource
def get_user_profile(user_id: str) -> dict:
    # In real code, fetch from DB with proper sanitization
    return {
        "id": user_id,
        "name": "Alice Example",
        "email": "alice@example.com",
        "tier": "enterprise"
    }
```

**Gaps & Challenges:**
- *Staleness*: Resources must be kept current; consider TTLs or event-driven updates.
- *Volume*: Large datasets should be summarized or paginated to fit LLM context windows.
- *Access Control*: Not all callers should see PII — apply AuthZ filters server-side.

**Interview Angle — sample question & answer:**
Q: *How would you prevent leaking sensitive fields through resources?*
A: Enforce authorization at the resource level, redact fields by default, return scoped views based on caller roles, and log access for auditing.

---

### *2.2 Tools — Definition & Best Practices*

**What it is:** Tools are callable actions that may cause side effects (create orders, cancel subscriptions, send emails).

**Why it matters:** Tools allow LLMs not just to reason, but to act. This transforms assistants into agents that can accomplish tasks end-to-end.

**How to hook it up (example):**

```
@mcp.tool
def place_order(user_id: str, product_id: str, quantity: int) -> dict:
    order_id = create_order_in_db(user_id, product_id, quantity)
    return {"status": "ok", "order_id": order_id}
```

**Gaps & Challenges:**

- *Authorization*: Require explicit role checks for destructive tools.
- *Idempotency*: Design tools to be safe on retries (return consistent order IDs or use request tokens).
- *Long-running operations*: Offload to background jobs and return job IDs; support cancellation/status endpoints.

**Interview Angle — sample question & answer:**
Q: *What would you do if an LLM repeatedly invoked a destructive tool?*
A: Implement throttling and rate limits, require multi-step confirmations for especially destructive actions, and enforce RBAC. Use callbacks to detect abnormal patterns and temporarily disable high-risk tools.

---

## *2.3 Prompts — Templates, Versioning & Safety*

**What it is:** Prompts are reusable templates or instruction fragments that guide how LLMs should format responses or interact with users/systems.

**Why it matters:** Prompts centralize the instruction set so behavior is consistent across agents. They also reduce prompt injection risk because templates are defined by developers and versioned.

**How to hook it up (example):**

```
@mcp.prompt("summarize_policy")
def summarize_policy(policy_text: str) -> str:
    return f"Summarize the policy concisely in 3 bullets:\n\n{policy_text}"
```

**Gaps & Challenges:**

- Versioning: Changing a prompt can alter downstream behavior unpredictably — use semantic versioning and changelogs.
- Evaluation: Prompts should be A/B tested and evaluated for bias and accuracy.
- Over-restriction: Excessive templating can reduce model creativity where it's needed.

**Interview Angle — sample question & answer:**
Q: *How would you roll out a prompt change in production?*
A: Stage changes via canary prompts (subset of traffic), measure effects on correctness and hallucination rates, and keep a rollback plan and versioned archives.

---

## *2.4 Clients — The Trust Boundary*

**What it is:** MCP clients are the gateway between LLMs/agents and MCP servers. They handle authentication, authorization, request shaping, and telemetry.

**Why it matters:** Direct LLM→Server communications are unsafe — clients enforce enterprise policies, prevent abusive calls, and provide audit trails.

**How to hook it up (example):**

```
client = MCPClient("https://mcp.example.com", token="sometoken")
profile = client.resources.get_user_profile(user_id="u123")
result = client.tools.place_order(user_id="u123", product_id="P001", quantity=1)
```

**Gaps & Challenges:**

- Latency overhead from the client hop — mitigate with caching and batching.
- Credential management — rotate tokens and integrate with IAM.
- Hardening against prompt injection where LLMs attempt to influence client behavior.

**Interview Angle — sample question & answer:**
Q: *Why can’t an LLM call a server directly?*

A: Without a client, you lose the enforcement point for security, logging, rate limiting, and policy checks. The client is where governance happens.

---

### *2.5 Servers — Hosting Capabilities Safely*

**What it is:** MCP servers host and register resources, tools, and prompts, and expose them over the MCP protocol.

**Why it matters:** Servers encapsulate the system-of-record logic and enforce server-side validation, sanitization, and access policies.

**How to hook it up (example):**

```
mcp = FastMCP("OrderService")

@mcp.resource
def get_order(order_id: str) -> dict:
    return fetch_order(order_id)

@mcp.tool
def cancel_order(order_id: str) -> dict:
    return cancel_order_in_db(order_id)

mcp.run(host="0.0.0.0", port=8000)
```

**Gaps & Challenges:**

- Versioning and backward compatibility as clients depend on endpoints.
- Observability to track which LLM/agent invoked which tool.
- Ensuring statelessness for scalable horizontal deployments.

**Interview Angle — sample question & answer:**
Q: *How would you version MCP APIs?*
A: Use semantic versioning for breaking changes, provide parallel versioned endpoints, and communicate deprecation schedules. Include automated compatibility tests.

---

## Part 3 — Clients vs Servers (Deep Dive)

### *Server Responsibilities (expanded)*

Servers are responsible for:

- Implementing business logic for tools/resources
- Enforcing server-side validation and sanitization
- Returning structured errors and typed responses
- Registering prompts and maintaining versions
- Emitting telemetry and audit logs

**Implementation notes:** Keep servers stateless where possible. For long-running tasks use external queues (Celery, RQ, or cloud task services) and expose job resources for status/cancellation.

### *Client Responsibilities (expanded)*

Clients are responsible for:

- Authentication (tokens, mTLS, OAuth flows)
- Authorization checks and token scoping
- Policy enforcement (rate limiting, input sanitization)
- Observability (enriched logs, correlation IDs)
- Resilience (retries, circuit breakers, batching)

**Suggested Interview Answer:** The client exists as a governance layer — it prevents unauthorized actions, logs interactions, applies rate limits, and transforms requests to match server expectations.

---

## Part 4 — Advanced Features (Enterprise Readiness)

### *4.1 Authentication & Authorization (AuthN/AuthZ)*

**Explanation:** Use industry-grade IAM (OIDC, OAuth2, SAML) and support RBAC or ABAC depending on requirements. Tokens should be short-lived where possible; use refresh flows and centralized revocation.

**How to hook it up (code):**

```
@require_role("admin")
@mcp.tool
def delete_user(user_id: str) -> dict:
    delete_from_db(user_id)
    return {"status":"deleted"}
```

**Gaps & Challenges:**
- Mapping enterprise roles to MCP-level operations.
- Token rotation and revocation in distributed systems.
- Least-privilege enforcement for fine-grained access.

**Interview Angle:** Explain how you'd integrate MCP access with corporate SSO, and how you would implement audit trails for compliance.

---

### *4.2 Sampling & Elicitation*

**Explanation:** Sampling generates multiple candidate LLM outputs to increase confidence; elicitation uses structured prompts to collect missing information.

**How to hook it up (pattern):**
- Design prompts that request structured JSON outputs or fixed schemas.
- Use client-side logic to request N samples and aggregate or vote on outputs.

**Gaps & Challenges:**
- Cost/latency trade-offs when sampling multiple times.
- Designing prompts that reliably produce structured outputs.

**Interview Angle:** Discuss trade-offs between higher sampling for accuracy vs increased latency/cost.

---

### *4.3 Callback Handlers (Middleware)*

**Explanation:** Callbacks run on events (before_tool, after_tool, on_error) and can implement logging, validation, or dynamic policy checks.

**How to hook it up (pseudocode):**

```
def audit_callback(ctx):
    logger.info("Tool called", extra=ctx)
mcp.register_callback("before_tool", audit_callback)
```

**Gaps & Challenges:**
- Performance overhead for synchronous callbacks.
- Error handling inside callbacks — they shouldn't break tool execution unless intended.

**Interview Angle:** Describe a callback that prevents a tool from running if the request includes suspicious input.

---

### *4.4 Error Handling & Resilience*

**Explanation:** Provide structured error types, use retries/backoff for transient failures, and expose cancellation for long-running jobs.

**Patterns:** Circuit breakers, retry with jitter, fallbacks, job queues, health checks, and graceful degradation.

**Interview Angle:** Walk through how you’d design an operation that depends on a flaky downstream API.

---

# Part 5 — Agent Integration (LangChain / AutoGen)

### *5.1 Why integrate MCP with agents?*

Agents need discoverable tools and safe primitives to act. MCP provides a standardized surface so agents don't require bespoke connectors for every system.

### *5.2 How to integrate (example)*

```
# Wrap MCP tools as agent tools
def place_order_tool(product_id, qty):
    return client.call_tool("place_order", product_id=product_id, quantity=qty)
```

**Orchestration patterns:**
- *Context → Decision → Action*: fetch resource, decide, call tool.
- *Multi-step planning*: agent composes multiple tool calls into a plan.
- *Human-in-the-loop confirmations*: require explicit approval before destructive steps.

**Gaps & Challenges:**
- Agents hallucinating tool names — validate requested tool names server-side.
- Latency when chaining many calls — consider caching and batching.

**Interview Angle:** Explain how to wrap MCP tools for LangChain and enforce authentication per tool.

---

# Part 6 — Enterprise Setup & Architecture

### *6.1 Security & Data Governance*

- Encrypt data in transit and at rest.
- Use centralized secrets (Vault, KMS).
- Apply data minimization and field-level redaction for PII.
- Maintain audit logs mapped to user identities/roles.

### *6.2 Deployment & Scaling*

- Containerize MCP servers and run on Kubernetes.
- Design stateless endpoints; delegate long tasks to queues.
- Use API gateways for authentication, throttling, and routing.

### *6.3 Observability & Monitoring*

- Emit structured logs, metrics, and traces.
- Use OpenTelemetry for distributed tracing across LLM, client, and server.
- Monitor tool usage patterns for anomalies and abuse.

### *6.4 Compliance*

- Map audit logs to compliance requirements (GDPR, HIPAA, SOC2).
- Provide deletion workflows and data access request handling.
- Document data flows and retention policies.

**Interview Angle:** Be prepared to describe how you'd prove compliance (logs, retention, access controls) in an audit.

---

# Part 7 — Testing & CI/CD (Expanded)

### *7.1 Unit Testing*

- Unit test tools/resources with mocks for external systems.
- Validate structured errors and edge cases.
- Use fixtures for reproducible test data.

**Example test skeleton:**

```
def test_place_order_happy_path():
    # mock DB and payment, assert order created
    pass
```

### *7.2 Integration Testing*

- Spin up test servers (Docker Compose) and run client interactions.
- Test auth flows, error responses, and resource shaping.

### *7.3 Mocking & Simulation*

- Simulate slow or failing downstream services.
- Test cancellation and retry logic.

### *7.4 CI Pipeline*

- Steps: lint, type checks, unit tests, integration tests, security scans, build artifacts.
- Tools: GitHub Actions, GitLab CI, Jenkins.

### *7.5 CD & Safe Deployments*

- Use canaries, blue-green, and progressive rollouts.
- Automate smoke tests and rollback triggers.

**Interview Angle:** Have a clear CI/CD diagram and mention observable metrics used as promotion gates (error rate, latency, test coverage).

---

## Appendix — Suggested Answers (Short)

**Q:** What is MCP?
**A:** MCP is a protocol that exposes resources, tools, and prompts to LLMs so they can safely read context and perform actions.

**Q:** Why a client?
**A:** To enforce security, logging, rate limits, and to act as the governance layer between LLMs and servers.

**Q:** How do you secure destructive tools?
**A:** RBAC, confirmation dialogs, audit logs, idempotency, and throttling.

---

## Appendix — Where the Code Lives

The repository contains illustrative code in these folders:

- `server/` — server examples (main, auth, tools, resources)
- `client/` — simple MCP client
- `agent_integration/` — example of wrapping MCP into agent tools
- `tests/` — unit test examples
- `infra/` — Dockerfile and manifests

---

**End of Handbook**