Model–Context Protocol (MCP)

Wook-Shin Han

June 1, 2025

Agenda

1. What is MCP?
2. Three Interfaces
   - Prompts
   - Tools
   - Prompts
   - Router Prompt
3. Monolithic vs. MCP
4. Umbrella Bot Case Study
   - User Story
   - Monolithic Flow
   - MCP Flow
5. Side-by-Side Comparison
6. Best Practices

MCP in One Sentence

The **Model–Context Protocol (MCP)** is a pattern for orchestrating large-language–model applications by *separating*:

- **Prompts**: reusable user-facing templates
- **Tools**: sandboxed capabilities the *model* can invoke
- **Resources**: structured context the *application* injects

This separation improves *security*, *maintainability*, and *cost efficiency*.

Prompts — Reusable Templates

Definition

A Prompt is a named text template that the application fills with variables and hands to the model.

Typical fields:

- `name`: identifier (e.g. `summarize_doc`)
- `template`: the textual skeleton with placeholders
- optional `metadata`: temperature, max tokens, etc.

Important: prompts contain *no secrets and no API calls*. They are pure text.

Tools — Sandboxed Capabilities

Definition

A Tool is a server-side function that the *model* may call when it needs external data or side-effects.

Examples (from a typical MCP toolkit):

- `weather` — fetch current forecasts
- `python` — run isolated Python code for computation / analysis
- `image_gen` — generate or edit images
- `automations.create` — schedule reminders

The model decides when to invoke a tool; the server validates arguments and executes it.

Prompts — Reusable Templates

Definition

A Prompt is a named text skeleton with placeholders that the server fills and sends to the model.

Typical fields:

- `name`: identifier (e.g. `"doc_summarizer"`)
- `template`: body with `variables`
- Optional `metadata`: temperature, max_tokens, etc.

Key rule: Prompts never embed secrets or API calls.

Router Prompt — Traffic Director

Purpose

Decide *which* downstream service (and thus which domain prompt) should handle the user's request.

- Very small template, e.g.:

  ```
  Label the request as SUMMARISE or REWRITE.
  ```

- Returns a single class label; result is invisible to the end user.
- Lives in the same Prompt layer but serves *classification*, not content generation.

Workflow

1. Server fills `query` placeholder with raw user text.
2. Calls LLM once $\Rightarrow$ gets label.
3. Internally forwards to the matching handler (e.g. `/summarise`).

Resources — Design Goals

When choosing *what* and *how much* to inject, keep four goals in mind:

1. **Relevance**: keep it tightly scoped to the current query.
2. **Freshness**: fetch or refresh just in time if data changes quickly.
3. **Compression**: token-budget aware (summaries, top-k, delta updates).
4. **Privacy**: redact or hash any user-private fields before insertion.

Resources — Concrete Example

Suppose a shopping assistant needs stock data and user tier:

```
{
  "user_profile": {
    "user_id": 42,
    "loyalty_tier": "gold"
  },
  "inventory_rows": [
    {"sku": "NB-123", "name": "Laptop 16GB RAM", "price": 950},
    {"sku": "NB-456", "name": "Laptop 32GB RAM", "price": 1200}
  ]
}
```

Note: Email, address, and other PII are omitted; only fields required for the model's ranking logic are present.

Resources — Lifecycle

- Fetch: server queries DB / external API.
- Transform: filter, sort, summarise, anonymise.
- Inject: pass into `web.run` or SDK call as `"resources": {...}`.
- Recycle: cache computed chunks for future turns when appropriate.

Outcome: consistent, minimal, and privacypreserving context each turn.

The Monolithic Approach

- One giant prompt contains:
  API keys, SQL, shell commands, business logic, style rules. . .
- Model executes everything **inline**, e.g. hitting external APIs directly.
- Any change (schema, policy) requires editing the whole prompt.

Drawbacks: security risk, token bloat, poor reuse, fragile maintenance.

The MCP Approach

- Prompt is slim: only instructions & placeholders.
- Server injects *Resources*. Secrets stay on the server.
- Model calls *Tools* when it needs actions or fresh data.
- Each layer evolves independently.

Benefits: clear trust boundaries, cheaper context windows, fast domain extension.

User Story

"Will it rain in Seoul this weekend? Also, set a 7 AM reminder tomorrow to bring my umbrella."

Required actions:

1. Fetch weekend forecast for Seoul.
2. Schedule a reminder at 07:00 next day.

Monolithic — Prompt Snippet

```
You are a helpful bot.
1. Query https://api.openweathermap.org/...&key=abc123
2. If user wants reminder, run:
   echo "notify-send 'Bring umbrella'" | at -t {{date}}0700
...
```

Monolithic— Problems

- API key `abc123` exposed in prompt logs.
- Model could hallucinate destructive shell commands.
- Entire 200-line prompt re-sent each request.

Step 1 — Application Pre-processing

Server side (no LLM yet):

1. Call weather API (key stored in env var).
2. Trim JSON to essential fields:

```
[ {"date":"2025-06-14","pop":0.62},
  {"date":"2025-06-15","pop":0.55} ]
```

Inject as *Resources* together with prompt `umbrella_advisor`.

Step 2 — LLM Interaction

Model receives:

```
PROMPT: umbrella_advisor
RESOURCES: {"forecast_json": [...], "note":"(umbrella in hallway)"}
```

It decides to:

- Parse pop values (>0.5 rain risk).
- Call `automations.create` tool:

```
{"tool":"automations.create",
 "args":{
   "title":"Bring umbrella",
   "prompt":"Tell me to bring an umbrella.",
   "schedule":"BEGIN:VEVENT\nDTSTART:20250613T220000Z\nEND:VEVENT"}}
```

Step 3 — Response to User

After the tool succeeds, the model returns:

> *"There is about a 60% chance of rain in Seoul this weekend. I'll remind you tomorrow at 7 AM — don't forget your umbrella!"*

Secrets remain hidden, and only whitelisted actions were possible.

Monolithic vs MCP — Quick Table

Monolithic	**MCP Separated**
Secrets inside prompt	Secrets stay on server
Model executes raw commands	Model can only call allowed tools
Full prompt resend each turn	Slim prompt + resource delta
Schema change rewrite prompt	Change resource mapper only
Hard to add new domain	Plug new prompt + (maybe) tool

Best Practices

- Keep prompts short; push data to Resources.
- Treat every Tool call as an API boundary — validate arguments.
- Never expose credentials or file paths to the model.
- Summarise / chunk Resources to fit context windows.
- Version prompts separately from application code.

Key Takeaways

1. **Prompts** define *how* to speak, not *how* to execute.
2. **Tools** give controlled super-powers to the model.
3. **Resources** feed the model the right data at the right time.

Together, they deliver *secure*, *scalable*, and *maintainable* LLM applications.

Questions?

Thank you!