Building Scalable MCP Servers with Domain-Driven Design

39 min read · May 24, 2025

Chris Hughes Follow

My intention is that my articles are free for everyone to read. If you don't have a medium subscription, you can access the article using this link.

. . .

AI models are becoming increasingly capable, resulting in a growing need for standardized ways to connect them with our data, tools, and systems. This has become even more pressing with the explosive growth of AI agents — autonomous systems that can understand requests, plan actions, and interact with tools to complete tasks.

In the past, integration was often ad-hoc and inconsistent; each application required custom connectors, proprietary interfaces, and duplicated effort. Just as REST APIs standardized how web services communicate, the time has come for a common language for AI models to interact with our digital ecosystem. This is where the **Model Context Protocol (MCP)** comes in — an open standard that enables LLMs to interact with external resources in a structured, maintainable way.

Whether we're building a simple weather lookup tool or a complex enterprise data integration, the architecture patterns we choose will determine how well our MCP servers scale and adapt over time. In this post, we'll explore how to apply Domain-Driven Design principles to build maintainable, scalable MCP servers.

Note on Code Examples: The code snippets in this post are simplified for illustration and learning purposes. Complete, runnable implementations with full error handling, logging, and configuration are available in the accompanying repository. Use the repository code as your starting point for real projects.

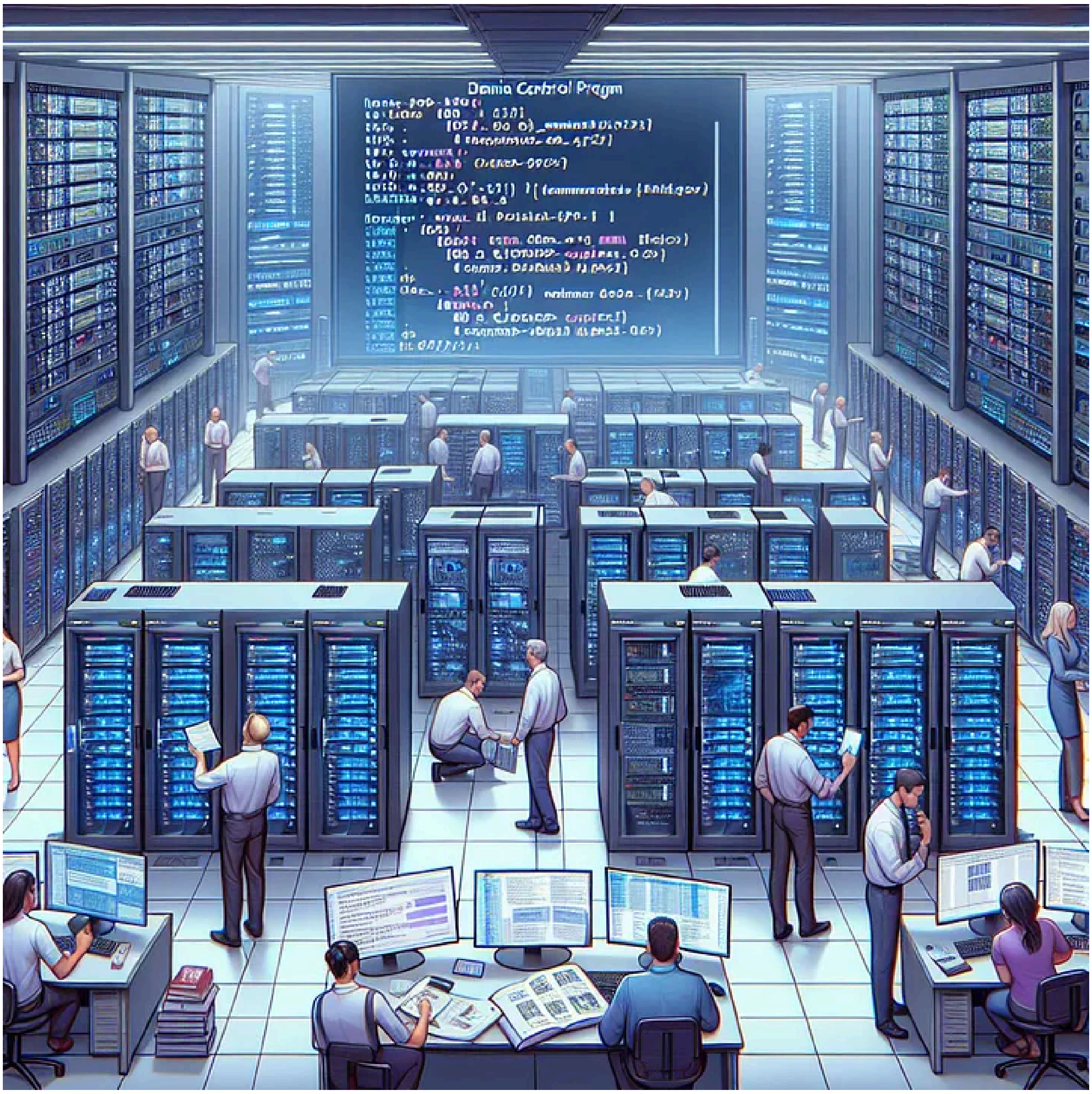

Image generated using Dalle3 with prompt "Building Scalable MCP Servers with Domain-Driven Design"

What is the Model Context Protocol?

The Model Context Protocol (MCP) is an open standard that aims to bring consistency to how applications provide context to Large Language Models (LLMs), providing LLMs with standardized access to external tools and data sources.

MCP servers expose three main types of capabilities, each serving different interaction patterns:

- **Tools** are functions that LLMs can actively call during their reasoning process. They perform computations, trigger actions, and can have side effects. For example, a `get_weather_forecast` tool might fetch real-time data from an API, or a `send_email` tool might trigger an actual email. Tools are discovered by the LLM at runtime, and the model decides when and how to invoke them based on the conversation context.
- **Resources** provide read-only access to data that can be loaded into the LLM's context. Unlike tools, resources are primarily informational and don't perform actions. Examples could include `file://documents/report.pdf` or `database://customers/recent_orders`. Resources can be static (fixed URIs) or dynamic (parameterized templates like `user://{user_id}/profile`). They're typically loaded by client applications rather than being called directly by the model.
- **Prompts** are reusable templates that help users initiate specific types of conversations or workflows. They can accept parameters and return structured prompt content. For instance, a `code_review_prompt` might take a programming language and code snippet as parameters, then return a formatted prompt that guides the LLM to perform a thorough code review.

When we build an MCP server, we're creating a bridge between an LLM and our specific data or functionality. For example, an LLM can use these tools to check the weather, search company documents, or analyse data — all without having these capabilities built directly into the model during training; all the model needs to know is how to use tools and resources. The beauty of MCP is its standardization — once we build a tool following the protocol, it can be used by any MCP-compatible client.

Building Our First MCP Server

Let's start with a practical example to understand MCP in action. We'll build a simple weather MCP server inspired by the official MCP quickstart guide, then gradually improve it using enterprise-grade architectural patterns.

Here's a basic weather server that exposes two tools for getting weather information:

```python
# /src/servers/simple_weather/weather.py

import logging
from typing import Any

import httpx
from mcp.server.fastmcp import FastMCP

logging.basicConfig(level=logging.DEBUG)
# Initialize FastMCP server
mcp = FastMCP("weather")

# Constants
NWS_API_BASE = "https://api.weather.gov"
USER_AGENT = "weather-app/1.0"


async def make_nws_request(url: str) -> dict[str, Any] | None:
    """Make a request to the NWS API with proper error handling."""
    headers = {"User-Agent": USER_AGENT, "Accept": "application/geo+json"}
    async with httpx.AsyncClient() as client:
        try:
            response = await client.get(url, headers=headers, timeout=30.0)
            response.raise_for_status()
            return response.json()
        except Exception:
            return None


def format_alert(feature: dict) -> str:
    """Format an alert feature into a readable string."""
    props = feature["properties"]
    return f"""
Event: {props.get("event", "Unknown")}
Area: {props.get("areaDesc", "Unknown")}
Severity: {props.get("severity", "Unknown")}
Description: {props.get("description", "No description available")}
Instructions: {props.get("instruction", "No specific instructions provided")}
"""


@mcp.tool()
async def get_alerts(state: str) -> str:
    """Get weather alerts for a US state.

    Args:
        state: Two-letter US state code (e.g. CA, NY)
    """
    url = f"{NWS_API_BASE}/alerts/active/area/{state}"
    data = await make_nws_request(url)

    if not data or "features" not in data:
        return "Unable to fetch alerts or no alerts found."

    if not data["features"]:
        return "No active alerts for this state."

    alerts = [format_alert(feature) for feature in data["features"]]
    return "\n---\n".join(alerts)


@mcp.tool()
async def get_forecast(latitude: float, longitude: float) -> str:
    """Get weather forecast for a location.

    Args:
        latitude: Latitude of the location
        longitude: Longitude of the location
    """
    # First get the forecast grid endpoint
    points_url = f"{NWS_API_BASE}/points/{latitude},{longitude}"
    points_data = await make_nws_request(points_url)

    if not points_data:
        return "Unable to fetch forecast data for this location."

    # Get the forecast URL from the points response
    forecast_url = points_data["properties"]["forecast"]
    forecast_data = await make_nws_request(forecast_url)

    if not forecast_data:
        return "Unable to fetch detailed forecast."

    # Format the periods into a readable forecast
    periods = forecast_data["properties"]["periods"]
    forecasts = []
    for period in periods[:5]:  # Only show next 5 periods
        forecast = f"""
{period["name"]}:
Temperature: {period["temperature"]}°{period["temperatureUnit"]}
Wind: {period["windSpeed"]} {period["windDirection"]}
Forecast: {period["detailedForecast"]}
"""
        forecasts.append(forecast)

    return "\n---\n".join(forecasts)


if __name__ == "__main__":
    print("starting weather server")
    # Initialize and run the server
    mcp.run(transport="stdio")
```

This implementation works and provides useful weather functionality to LLMs. You can run it and connect through MCP clients to get real weather data. But as we examine the code more closely, several architectural issues become apparent:

Mixed Responsibilities: In the `get_alerts` function, business logic is directly coupled with MCP infrastructure. In just 15 lines, it's handling:

- URL construction for external APIs
- HTTP communication and error handling
- JSON parsing and data extraction
- Business logic (what constitutes an alert)
- Response formatting for LLM consumption
- MCP protocol concerns (the `@mcp.tool()` decorator)

Testing Challenges: How would you test the alert formatting logic? Currently, you'd need to either:

- Make real HTTP calls to the NWS API (slow, unreliable, requires internet)
- Mock the entire `httpx` library (complex setup, brittle tests)
- Test everything together as an integration test (hard to isolate failures)

No Domain Model: We're working directly with raw dictionaries from the API response. When you see `props.get("event", "Unknown")`, what business rule does that represent? What happens if the API changes its response format?

Inflexible Infrastructure: The HTTP client is hard-coded into our business logic. What if we need to add:

- Authentication headers for a different weather service?
- Retry logic for rate limiting?
- Caching to avoid redundant API calls?
- Different timeout settings for different endpoints?

Limited Reusability: This code is tightly coupled to MCP. If you wanted to expose the same weather functionality through a REST API, CLI tool, or GraphQL endpoint, you'd need to duplicate all the business logic.

Additionally, we're losing the semantic richness of our domain. When the code processes this API response:

```
{
  "properties": {
    "event": "Heat Advisory",
    "severity": "Moderate",
    "areaDesc": "Harris County",
    "description": "Heat index values up to 108 degrees..."
  }
}
```

We immediately flatten it into a string for display. We lose the ability to:

- Reason about alert severity levels
- Group alerts by geographic area
- Store historical alert data for analysis
- Apply business rules based on alert types

These aren't just theoretical concerns — they become real limitations as requirements evolve. What happens when someone asks for "severe weather alerts only" or "alerts from the past week"?

When Does This Matter?

For simple scripts, proof-of-concepts, or single-purpose tools, the straightforward approach shown above may be the best solution in terms of simplicity, ease of understanding and maintenance. If you're building a quick weather lookup for personal use or a prototype to demonstrate an idea, the additional complexity of Domain-Driven Design probably isn't justified.

However, DDD's benefits emerge when building systems that need to handle:

- **Changing requirements:** "Now we need to support multiple weather providers"

- **Complex business rules**: “Only show severe alerts during business hours”
- **Integration across interfaces**: The same logic needs to work in REST APIs, CLIs, and MCP servers
- **Long-term maintenance**: The system will be modified and extended over months or years
- **Multiple team members**: Other developers need to understand and modify the code

The patterns we’ll explore next become valuable when your system needs to evolve and adapt over time.

Domain-Driven Design: A Foundation for LLM Interactions

Now that we’ve seen the limitations of the simple approach, let’s explore how Domain-Driven Design can address these problems.

In the past, I’ve applied Domain-Driven Design principles to web servers with great success, and when I first encountered the Model Context Protocol, I immediately saw how these same patterns could apply. DDD isn’t just another architectural pattern — it’s a way of thinking about software that puts our business domain at the centre. Instead of organizing code by technical concerns (controllers, models, views), DDD organizes it by business capabilities (weather forecasting, order processing, user management).

This domain-centric approach is even more critical when working with LLMs than with traditional software. Since LLMs operate primarily through understanding context and semantics, having clearly defined domain concepts directly improves their ability to select and use the right tools appropriately. When our MCP servers are built around well-modelled domains with a consistent ubiquitous language, we provide LLMs with the precise context they need to interpret requests correctly and choose the appropriate tools, resources, or prompts.

Understanding the Weather Domain

Before writing any code, DDD encourages us to understand the domain we’re working in. In weather forecasting, we have concepts like:

- **Forecasts**: Complete weather predictions for specific locations
- **Periods**: Time segments within a forecast (like “Tonight” or “Tuesday”)

- **Alerts**: Official warnings about dangerous weather conditions that were active at a specific time and location

These aren't just data structures — they're meaningful concepts in our domain. When we model our software using this domain language, several things happen:

LLMs Understand Better: When an LLM sees a function called `get_severe_alerts()`, it immediately understands this will return serious weather threats. A function called `execute_query_for_weather_data_endpoint_with_severity_filter()` requires parsing technical jargon to understand the business purpose.

Code Becomes Self-Documenting: When you see `alert.severity == "Extreme"` in code, you immediately know this is handling the most serious weather threats. Raw dictionary access like `data["properties"]["severity"] == "Extreme"` buries this business meaning in implementation details.

Requirements Are Clearer: Stakeholders can say "We need to track severe alerts over time" instead of "We need to store API responses with severity properties in a database table."

The concept of a **ubiquitous language** — a common vocabulary shared between domain experts and developers — is central to DDD. In an MCP context, this language becomes the bridge between human intent, LLM understanding, and system capabilities.

DDD Building Blocks for Our Weather Domain

Let's recap some DDD concepts, and apply these to our weather domain:

Value Objects represent domain concepts without identity. Our `WeatherPeriod` (temperature, wind, forecast text) and `Coordinate` (latitude, longitude) are perfect examples. Two weather periods with the same data are effectively identical.

Entities have unique identity that persists over time. While our simple example doesn't need entities, a more complex system might have `WeatherStation` entities with unique identifiers and changing conditions.

Domain Services encapsulate business logic that doesn't naturally belong in entities or value objects. A `WeatherService` may be defined to contain the business rules for retrieving and formatting weather data - this maps perfectly to MCP tools.

Repositories provide collection-like interfaces for accessing domain objects. A `WeatherAlertRepository` could be used find alerts by region and time range without exposing whether they're stored in files, databases, or memory.

Application Services orchestrate domain objects to fulfill specific use cases while staying independent of infrastructure concerns. We could use a `WeatherMCPService` to coordinate domain services and repositories while handling MCP protocol requirements.

The Root Problem: Infrastructure Leaking Into Domain Logic

Looking back at our simple weather server, the core issue is that infrastructure concerns have leaked into business logic. The `get_alerts` function doesn't just handle the business concept of "getting weather alerts" - it's also responsible for HTTP communication, JSON parsing, error handling, and response formatting.

This mixing of concerns makes the code:

- **Hard to test**: You can't verify alert logic without HTTP calls
- **Difficult to change**: Switching APIs requires changing business logic
- **Impossible to reuse**: The weather logic only works within MCP
- **Hard to understand**: Business rules are buried in infrastructure code

DDD gives us patterns to separate these concerns cleanly, making our code more maintainable, testable, and adaptable to changing requirements.

The Refactored Domain-Driven Implementation

Let's refactor our weather server using DDD principles. We'll build it layer by layer, showing how each piece contributes to a more maintainable architecture.

Domain Models: Speaking the Weather Language

First, we create domain models that represent weather concepts using language that domain experts would recognize:

```
# /src/servers/weather_service/domain/models.py

@dataclass
class WeatherPeriod:
    """Represents a single period in a weather forecast.
    A weather period typically corresponds to a 12-hour timeframe
    (e.g., "Tuesday Night", "Wednesday") and contains the core
    meteorological data for that period.

    Attributes:
        name: Human-readable period name (e.g., "Tonight", "Tuesday")
        temperature: Temperature value in the specified unit
        temperature_unit: Temperature unit ("F" for Fahrenheit, "C" for Celsius
        wind_speed: Wind speed description (e.g., "5 to 10 mph")
        wind_direction: Wind direction abbreviation (e.g., "NW", "SSE")
        detailed_forecast: Complete narrative forecast for this period
    """

    name: str
    temperature: float
    temperature_unit: str
    wind_speed: str
    wind_direction: str
    detailed_forecast: str

    def to_display_string(self) -> str:
        """Format for human-readable display"""
        return f"{self.name}: {self.temperature}°{self.temperature_unit} - {sel


@dataclass
class Forecast:
    """Weather forecast aggregate containing multiple time periods.

    Represents a complete weather forecast for a location, typically
    covering the next 5-7 periods (2-3 days). Follows DDD aggregate
    pattern where Forecast is the aggregate root.

    Attributes:
        periods: List of forecast periods, ordered chronologically
        error: Error message if forecast retrieval failed, None if successful
        latitude: Location latitude in decimal degrees
        longitude: Location longitude in decimal degrees
        retrieved_at: UTC timestamp when forecast was fetched
    """

    periods: List[WeatherPeriod]
    error: str
    latitude: float
    longitude: float
    retrieved_at: datetime.datetime = None

    def to_display_string(self) -> str:
        """Format for human-readable display"""
        if not self.periods:
            return f"Error: {self.error or 'No forecast data available'}"

        return (
            f"Forecast retrieved at: {self.retrieved_at.strftime('%Y-%m-%d %H:%
            + f"\nLocation: {self.latitude:.4f}, {self.longitude:.4f}"
            + "\n".join(period.to_display_string() for period in self.periods)
        )


@dataclass
class WeatherAlert:
    """Individual weather alert issued by meteorological authorities.

    Represents a single weather warning, watch, or advisory (e.g., tornado
    warning, flood watch, heat advisory). Alerts are value objects in DDD
    terms - they're identified by their content rather than a unique ID.

    Attributes:
        event: Alert type/category (e.g., "Tornado Warning", "Heat Advisory")
        area: Geographic description of affected area (e.g., "Harris County, TX
        severity: Alert severity level ("Minor", "Moderate", "Severe", "Extreme
        description: Full alert description with details and impacts
        instructions: Safety instructions for the public (None if not provided)
    """

    event: str
    area: str
    severity: str
    description: str
    instructions: Optional[str] = None

    def to_display_string(self) -> str:
        """Format alert for display"""
        result = f"Event: {self.event}\nArea: {self.area}\nSeverity: {self.seve
        result += f"\nDescription: {self.description}"

        if self.instructions:
            result += f"\nInstructions: {self.instructions}"

        return result
```

Notice how these models use the vocabulary of the weather domain. Terms like “forecast”, “period”, “alert”, and “severity” are immediately meaningful to both weather experts and LLMs. Our **ubiquitous language** creates shared understanding across all parts of our system.

The models also include display formatting methods. This follows DDD principles by keeping presentation logic close to the domain objects that know how to represent themselves.

Domain Services: Encapsulating Weather Intelligence

Next, let's create a domain service that encapsulates our weather-related business logic; we can define an abstract interface that expresses what our domain can do, independent of how it's implemented:

```python
# /src/servers/weather_service/domain/service/interfaces.py

class WeatherService(ABC):
    """Abstract interface for weather data retrieval services.

    Defines the domain contract for weather operations. Implementations
    should handle external API communication while maintaining domain
    model consistency.

    This follows the DDD domain service pattern for operations that
    don't naturally belong to a single entity or value object.
    """

    @abstractmethod
    async def get_forecast(self, latitude: float, longitude: float) -> Forecast
        """Retrieve weather forecast for geographic coordinates.

        Args:
            latitude: Latitude in decimal degrees, range [-90, 90]
            longitude: Longitude in decimal degrees, range [-180, 180]

        Returns:
            Forecast aggregate with periods and metadata
        """
        pass

    @abstractmethod
    async def get_alerts(self, state: str) -> List[WeatherAlert]:
        """Retrieve active weather alerts for a US state.

        Args:
            state: Two-letter US state code (e.g., "CA", "TX", "FL")
                Must be normalized to uppercase by implementations
                Invalid state codes should return empty list, not error

        Returns:
            List of WeatherAlert domain objects representing all currently
            active alerts for the specified state. Empty list indicates no
            active alerts (normal condition, not an error).
        """
        pass
```

Abstract Interfaces: Domain Focus and LLM-Friendly Code Generation

This abstract interface serves multiple purposes beyond traditional software engineering benefits. By keeping the interface focused purely on domain concepts — forecasts, alerts, coordinates, states — we create a clear contract that isolates business logic from technical implementation details.

This domain-focused design has an unexpected modern benefit: it significantly improves LLM-assisted code generation. When an LLM sees a well-defined interface like `WeatherService`, it understands exactly what needs to be implemented:

```
# Example LLM prompt: "Implement a mock WeatherService for testing"
class MockWeatherService(WeatherService):
    """The LLM knows exactly what methods to implement and their signatures"""
    async def get_forecast(self, latitude: float, longitude: float) -> Forecast
        # LLM generates appropriate mock data with correct types
        return Forecast(
            periods=[WeatherPeriod(name="Test", temperature=72.0, ...)],
            error=None,
            latitude=latitude,
            longitude=longitude,
            retrieved_at=datetime.now()
        )
```

The constrained interface guides the LLM to generate implementations that follow our domain model, use the correct types, and maintain business logic separation. This makes AI-assisted development much more reliable than working with loosely-defined functions or mixed-responsibility classes.

This interface is crucial because it expresses our domain capabilities without binding us to any specific implementation. We can create an NWS implementation, a mock implementation for testing, or even a composite implementation that combines multiple weather services.

The concrete implementation focuses solely on translating between the external API and our domain models:

```python
# /src/servers/weather_service/domain/service/services.py

import datetime
from typing import Any, Dict, List

from servers.weather_service.domain.models import Forecast, WeatherAlert, Weath
from servers.weather_service.domain.service.interfaces import WeatherService
from servers.weather_service.infrastructure.adaptors import make_request


class NWSWeatherService(WeatherService):
    """National Weather Service implementation of WeatherService.

    Integrates with the NWS API (weather.gov) which provides free
    weather data for US locations. Uses the two-step process:
    1. GET /points/{lat},{lon} to get forecast endpoint
    2. GET forecast endpoint to retrieve actual forecast data

    Args:
        make_http_request: HTTP client function for dependency injection
    """

    def __init__(self, make_http_request=make_request):
        self.get = make_http_request
        self.base_url = "https://api.weather.gov"
        self.headers = {
            "User-Agent": "weather-app/1.0",
            "Accept": "application/geo+json",
        }

    async def get_forecast(self, latitude: float, longitude: float) -> Forecast
        """Get weather forecast for a location.

        Args:
            latitude: Latitude of the location
            longitude: Longitude of the location
        """
        # First get the forecast grid endpoint
        points_url = f"{self.base_url}/points/{latitude},{longitude}"
        request_time = datetime.datetime.now()
        points_data = await self.get(points_url, headers=self.headers)

        if not points_data:
            return Forecast(
                periods=[],
                error="Unable to fetch forecast data for this location.",
                retrieved_at=request_time,
                latitude=latitude,
                longitude=longitude,
            )

        # Get the forecast URL from the points response
        forecast_url = points_data["properties"]["forecast"]
        forecast_data = await self.get(forecast_url, headers=self.headers)

        if not forecast_data:
            return Forecast(
                periods=[],
                error="Unable to fetch detailed forecast.",
                retrieved_at=request_time,
                latitude=latitude,
                longitude=longitude,
            )

        # Convert API data to Domain objects
        periods = []
        for period in forecast_data["properties"]["periods"][:5]:
            periods.append(
                WeatherPeriod(
                    name=period["name"],
                    temperature=period["temperature"],
                    temperature_unit=period["temperatureUnit"],
                    wind_speed=period["windSpeed"],
                    wind_direction=period["windDirection"],
                    detailed_forecast=period["detailedForecast"],
                )
            )

        return Forecast(
            periods=periods,
            error=None,
            retrieved_at=request_time,
            latitude=latitude,
            longitude=longitude,
        )

    async def get_alerts(self, state: str) -> List[WeatherAlert]:
        """Get weather alerts for a US state.

        Args:
            state: Two-letter US state code (e.g. CA, NY)
        """
        data = await self.get(
            f"{self.base_url}/alerts/active/area/{state}", headers=self.headers
        )

        if not data or "features" not in data:
            return []  # "Unable to fetch alerts or no alerts found."

        if not data["features"]:
            return []  # "No active alerts for this state."

        alerts = []
        for feature in data["features"]:
            props = feature["properties"]
            alerts.append(
                WeatherAlert(
                    event=props.get("event", "Unknown"),
                    area=props.get("areaDesc", "Unknown"),
                    severity=props.get("severity", "Unknown"),
                    description=props.get("description", "No description availa
                    instructions=props.get("instruction"),
                )
            )

        return alerts
```

Infrastructure Separation: Keeping External Concerns at the Boundary

Notice how we've isolated HTTP communication in the infrastructure layer by injecting the `make_http_request` function:

```
# infrastructure/adaptors.py
async def make_request(url: str, headers: Dict[str, str] = None) -> Dict[str, A
    """HTTP client adapter for external API calls."""
    async with httpx.AsyncClient() as client:
        try:
            response = await client.get(url, headers=headers, timeout=30.0)
            response.raise_for_status()
            return response.json()
        except Exception:
            return None  # Simplified error handling
```

This separation is a key DDD principle — infrastructure concerns like HTTP clients, database connections, or file I/O should be isolated from domain logic. This provides several benefits:

Testing Flexibility: We can inject a stub HTTP client for unit tests:

```
# In tests - using a STUB to return hardcoded responses
async def http_stub(url, headers=None):
    if "points" in url:
        return {"properties": {"forecast": "https://api.weather.gov/forecast/..
    elif "forecast" in url:
        return {"properties": {"periods": [test_period_data]}}
    # ... other test cases
```

```
weather_service = NWSWeatherService(http_stub)
forecast = await weather_service.get_forecast(37.7749, -122.4194)
# Test domain logic without network calls
```

Configuration Management: We can inject different HTTP clients with different settings:

```
# For production - with retries and auth
production_client = make_authenticated_request_with_retries

# For development - with debug logging
debug_client = make_request_with_logging

weather_service = NWSWeatherService(production_client)
```

Repository Layer: Abstracting Data Persistence

The National Weather Service API provides current conditions but no historical data. To enable trend analysis and provide context for LLM interactions, we need to persist weather information over time. In DDD, repositories provide collection-like interfaces for accessing domain aggregates while abstracting away storage concerns.

Let's start by defining what our weather domain needs from data persistence:

```
#/src/servers/weather_service/domain/repository/interfaces.py

class WeatherForecastRepository(ABC):
    """Repository for weather forecast aggregate persistence.

    Stores and retrieves complete Forecast aggregates by geographic location
    and time. Supports location-based queries and time-windowed filtering
    for both current and historical forecast analysis.
    """

    @abstractmethod
    async def get_forecasts(
        self,
        latitude: float,
        longitude: float,
        time_window: Union[int, timedelta] = 3,
        time_unit: str = "hours",
        limit: Optional[int] = None,
    ) -> List[Forecast]:
        """Find forecasts for a location within a time window.

        Args:
            latitude: Geographic latitude in decimal degrees
            longitude: Geographic longitude in decimal degrees
            time_window: How far back to search (int uses time_unit)
            time_unit: "hours" or "days" when time_window is int
            limit: Maximum forecasts to return

        Returns:
            List of complete Forecast aggregates, newest-first
        """
        pass

    @abstractmethod
    async def save_forecast(
        self, latitude: float, longitude: float, forecast: Forecast
    ) -> None:
        """Save a complete forecast aggregate.

        Args:
            latitude: Geographic latitude
            longitude: Geographic longitude
            forecast: Complete Forecast with all periods and metadata
        """
        pass


class WeatherAlertRepository(ABC):
    """Repository for weather alert snapshot aggregate persistence.

    Manages AlertSnapshot aggregates that capture the complete set of alerts
    active for a state at specific points in time. Enables historical alert
    pattern analysis and trend monitoring.
    """

    @abstractmethod
    async def get_alerts(
        self,
        state: str,
        time_window: Union[int, timedelta] = 24,
        time_unit: str = "hours",
        limit: Optional[int] = None,
    ) -> List[AlertSnapshot]:
        """Find alert sets for a state within a time window
        Args:
            state: Two-letter US state code
            time_window: Number of time units to look back
            time_unit: Either "hours" or "days"
            limit: Maximum number of alert sets to return

        Returns:
            List of alert sets matching criteria, sorted newest first
        """
        pass

    @abstractmethod
    async def save_alerts(
        self, state: str, alerts: List[WeatherAlert]
    ) -> AlertSnapshot:
        """Save alerts as a new alert set

        Args:
            state: Two-letter US state code
            alerts: List of alerts to save (empty list = no alerts)

        Returns:
            The created AlertSnapshot
        """
        pass
```

These interfaces express our domain's data needs without committing to any storage technology. We can query forecasts by location and time, alerts by state and time, and persist both types of weather data.

Building Reusable Infrastructure

Rather than implementing each repository from scratch, we can identify common patterns and create reusable infrastructure. Our weather data has a common characteristic: everything is timestamped and organized by geographic keys (locations for forecasts, states for alerts). This leads us to a two-layer approach:

Generic Infrastructure (reusable across any domain):

```
# /src/servers/weather_service/infrastructure/repositories.py

class TimestampedCollectionRepository(JsonFileRepository, Generic[T, K]):
    """Generic repository for timestamped domain objects with TTL support.

    Implements a generic pattern for storing collections of timestamped
    objects with automatic expiration and
    size limits. Uses the Repository pattern from DDD to abstract
    persistence concerns.

    Type Parameters:
        T: Domain object type (must have timestamp attribute)
        K: Key type for grouping objects (str for location keys)

    Args:
        file_path: JSON file path for persistence
        max_items_per_key: Maximum objects to retain per key
    """

    def __init__(self, file_path: str, max_items_per_key: int = 20):
        """Initialize repository"""
        self.collections: Dict[K, Deque[T]] = {}
        self.max_items_per_key = max_items_per_key
        super().__init__(file_path)

    async def find_items(
        self,
        key: K,
        time_window: Union[int, timedelta] = 24,
        time_unit: str = "hours",
        limit: Optional[int] = None,
        timestamp_getter: Callable[[T], datetime] = lambda x: x.retrieved_at,
    ) -> List[T]:
        """Find items within a time window, newest first.

        Args:
            key: Grouping key (e.g., location identifier)
            time_window: How far back to look (int or timedelta)
            time_unit: Units for time_window if int ("hours" or "days")
            limit: Maximum items to return (None for no limit)
            timestamp_getter: Function to extract timestamp from items

        Returns:
            List of items matching criteria, ordered newest to oldest
        """
        if key not in self.collections:
            return []

        # Calculate cutoff time
        cutoff_time = self._calculate_cutoff_time(time_window, time_unit)

        # Filter items by time
        valid_items = [
            item
            for item in self.collections[key]
            if timestamp_getter(item) >= cutoff_time
        ]

        # Apply limit if specified
```

```python
        if limit is not None and limit > 0:
            valid_items = valid_items[:limit]

        return valid_items

    async def save_item(self, key: K, item: T) -> T:
        """Save an item under the given key"""
        # Initialize deque if this is the first item for this key
        if key not in self.collections:
            self.collections[key] = deque(maxlen=self.max_items_per_key)

        # Add new item at the beginning (newest first)
        self.collections[key].appendleft(item)

        # Save to file
        self._save_to_file(self._serialize_data())

        return item

    async def get_most_recent_item(self, key: K) -> Optional[T]:
        """Get the most recent item for a key"""
        if key not in self.collections or not self.collections[key]:
            return None

        return self.collections[key][0]  # First item is most recent

    def _calculate_cutoff_time(
        self, time_window: Union[int, timedelta], time_unit: str
    ) -> datetime:
        """Calculate cutoff time based on window and unit"""
        if isinstance(time_window, timedelta):
            return datetime.now() - time_window

        if time_unit == "hours":
            return datetime.now() - timedelta(hours=time_window)
        elif time_unit == "days":
            return datetime.now() - timedelta(days=time_window)
        else:
            raise ValueError(
                f"Invalid time_unit: {time_unit}, must be 'hours' or 'days'"
            )
```

Domain-Specific Implementation (weather business logic):

```python
#/src/servers/weather_service/domain/repository/repositories.py

class JsonFileWeatherAlertRepository(
    TimestampedCollectionRepository[AlertSnapshot, str], WeatherAlertRepository
):
    """Repository for weather alerts using JSON file storage"""

    def __init__(
        self, file_path: str = "weather_alerts.json", max_sets_per_state: int =
    ):
        """Initialize repository"""
        super().__init__(file_path, max_sets_per_state)

    async def get_alerts(
        self,
        state: str,
        time_window: Union[int, timedelta] = 24,
        time_unit: str = "hours",
        limit: Optional[int] = None,
    ) -> List[AlertSnapshot]:
        """Find alert sets for a state within a time window"""
        key = self._make_state_key(state)
        return await self.find_items(key, time_window, time_unit, limit)

    async def save_alerts(
        self, state: str, alerts: List[WeatherAlert]
    ) -> AlertSnapshot:
        """Save alerts as a new alert set"""
        # Create new alert set
        alert_set = AlertSnapshot(
            alerts=alerts, retrieved_at=datetime.now(), state=state.upper()
        )

        key = self._make_state_key(state)
        return await self.save_item(key, alert_set)

    def _make_state_key(self, state: str) -> str:
        """Create a unique key for state"""
        return f"state:{state.upper()}"

    def _serialize_item(self, alert_set: AlertSnapshot) -> Dict[str, Any]:
        """Convert an AlertSnapshot to a serializable dictionary"""
        return {
            "state": alert_set.state,
            "retrieved_at": alert_set.retrieved_at.isoformat(),
            "alerts": [self._serialize_alert(alert) for alert in alert_set.aler
        }

    def _deserialize_item(self, data: Dict[str, Any]) -> AlertSnapshot:
        """Create an AlertSnapshot from a dictionary"""
        return AlertSnapshot(
            state=data["state"],
            retrieved_at=datetime.fromisoformat(data["retrieved_at"]),
            alerts=[
                self._deserialize_alert(alert_data) for alert_data in data["ale
            ],
        )

    def _deserialize_key(self, key_str: str) -> str:
        """Keys are already strings"""
        return key_str

    def _serialize_alert(self, alert: WeatherAlert) -> dict:
        """Convert a WeatherAlert to a serializable dictionary"""
        return {
            "event": alert.event,
            "area": alert.area,
            "severity": alert.severity,
            "description": alert.description,
            "instructions": alert.instructions,
        }

    def _deserialize_alert(self, data: dict) -> WeatherAlert:
        """Create a WeatherAlert from a dictionary"""
        return WeatherAlert(
            event=data["event"],
            area=data["area"],
            severity=data["severity"],
            description=data["description"],
            instructions=data.get("instructions"),
        )
```

n our implementation, domain repository implementations live in the domain layer (`domain/repository/repositories.py`) rather than the infrastructure layer. This follows the principle that these implementations contain weather-specific business logic—how to construct location keys, what time windows make sense for weather data, how to serialize weather concepts.

The infrastructure layer (`infrastructure/repositories.py`) contains the generic, reusable patterns that any domain could use. This separation ensures our domain repositories are focused on weather concepts while leveraging proven technical patterns.

Why This Layered Approach Works

This separation provides several key benefits:

Domain Focus: The weather repositories contain only weather-specific logic — how to construct location keys from coordinates, how to serialize weather data, what time windows make sense for forecasts vs. alerts.

Reusability: The `TimestampedCollectionRepository` can be used by any domain that needs time-based data storage - user activity logs, financial transactions, system events, etc.

Testing Clarity: We can test the generic infrastructure separately from domain-specific behavior, making failures easier to diagnose and fix.

Technology Independence: If we wanted to switch from JSON files to Redis, we'd only need to change the infrastructure layer. The domain-specific logic about forecasts and alerts remains unchanged.

Application Service: Orchestrating the Domain

Now that we have domain services for weather operations and repositories for data persistence, we need something to coordinate these components and expose them through the MCP protocol. This is where the Application Service pattern from DDD becomes essential.

The application service sits between the domain layer and the external world (in this case, MCP clients). It has a specific responsibility: **orchestrate domain objects to fulfill complete use cases** while handling protocol-specific concerns.

Our `WeatherMCPService` serves as the translator between MCP protocol requirements and our weather domain capabilities:

```
#/src/servers/weather_service/application/mcp_server.py

class WeatherMCPService(MCPApplicationService):
    """MCP Application Service for weather domain operations.

    Exposes weather domain capabilities through the Model Context Protocol.
    Orchestrates WeatherService and Repository operations while maintaining
    separation from MCP protocol details.

    Available Tools:
        - get_forecast: Real-time weather forecast retrieval
        - get_alerts: Current weather alerts for US states

    Available Resources:
        - historical://alerts/{state}: Historical alert data

    Available Prompts:
        - weather_analysis_prompt: Structured weather analysis template
    """
    def __init__(
        self,
        mcp: FastMCP,
        weather_service: WeatherService,
        weather_forecast_repository: WeatherForecastRepository,
        weather_alert_repository: WeatherAlertRepository,
    ):
        self.weather_service = weather_service
        self.weather_forecast_repository = weather_forecast_repository
        self.weather_alert_repository = weather_alert_repository
        super().__init__(mcp)

    @property
    def tools(self) -> List[Tuple[str, str, Callable]]:
        """Register tools for the MCP server"""
        return [
            (
                "get_forecast",
                "Get weather forecast for a location",
                self.get_forecast,
            ),
            (
                "get_alerts",
                "Get weather alerts for a US state",
                self.get_alerts,
            ),
        ]

    @property
    def resources(self) -> List[Tuple[str, str, str, Callable]]:
        """Register resources for the MCP server"""
        return [
            (
                "historical://alerts/{state}",
                "Get historical weather alerts for a US state",
                "Get historical weather alerts for a US state. State should be
                self.get_historical_alerts,
            ),
        ]

    @property
    def prompts(self) -> List[Tuple[str, str, Callable]]:
        """Register prompts for the MCP server"""
        return [
            (
                "weather_analysis",
                "Analyze weather conditions for a location",
                self.weather_analysis_prompt,
            ),
        ]

    def weather_analysis_prompt(self, location: str) -> str:
        """
        A prompt template for analyzing weather conditions for a location.

        Args:
            location: The location to analyze weather for
        """
        return f"""
        You are a weather analysis expert providing detailed insights about wea

        Analyze the current and forecasted weather conditions for {location}.
        Include information about:
        - Current temperatures and conditions
        - Expected changes over the next few days
        - Any notable weather patterns or anomalies
        - Practical advice based on the conditions

        Present your analysis in a clear, structured format that's easy to unde
        """

    async def get_forecast(self, latitude: float, longitude: float) -> str:
        """MCP tool: Get weather forecast for coordinates.

        Retrieves current forecast from weather service and persists
        to repository for historical tracking. Coordinates are validated
        and forecast is formatted for LLM consumption.

        Args:
            latitude: Decimal degrees latitude [-90, 90]
            longitude: Decimal degrees longitude [-180, 180]

        Returns:
            Human-readable forecast string with periods and metadata

        Note:
            This is an MCP tool function - return values should be
            formatted for LLM understanding rather than programmatic use.
        """
        forecast = await self.weather_service.get_forecast(latitude, longitude)
        await self.weather_forecast_repository.save_forecast(
            latitude, longitude, forecast
        )
        return forecast.to_display_string()

    async def get_alerts(self, state: str) -> str:
        """Retrieve active weather alerts for a US state.

        Fetches current weather warnings, watches, and advisories from the
        National Weather Service for the specified state. Returns all active
        alerts regardless of severity level.

        Args:
            state: Two-letter US state code (e.g., "CA", "TX", "FL")
                   Case-insensitive, automatically normalized to uppercase
        Returns:
            List of active WeatherAlert objects, empty list if no alerts

        """
        alerts = await self.weather_service.get_alerts(state)

        if not alerts:
            return f"No active alerts for {state}"

        await self.weather_alert_repository.save_alerts(state, alerts)

        return "\n---\n".join(alert.to_display_string() for alert in alerts)

    async def get_historical_alerts(self, state: str) -> str:
        """MCP resource handler for historical weather alerts.

        Provides access to recently cached weather alerts through the MCP
        resource system. This allows LLMs to access historical alert data
        for analysis and comparison.

        URI Template: historical://alerts/{state}

        Args:
            state: Two-letter US state code (e.g., "CA", "TX")

        Returns:
            Formatted historical alerts or "no data" message

        Note:
            This is a resource handler, not a tool. Resources provide
            read-only access to data, while tools perform actions.
        """
        alerts = await self.weather_alert_repository.get_alerts(
            state=state,
        )

        if not alerts:
            return f"No historical alerts found for {state}"

        return "\n\n===\n\n".join(alert_set.to_display_string() for alert_set i
```

Notice how this application service:

- **Orchestrates** domain services and repositories without containing business logic
- **Translates** between domain models and MCP protocol requirements

- **Coordinates** multiple operations (fetch data, persist it, format response)
- **Exposes both tools and resources** through MCP

Additionally, our the application service takes its dependencies through constructor injection. This allows us to compose different implementations — a production service with real APIs and databases, or a test service with mocks and stubs.

Once again, we have abstracted much of the technical complexity to `MCPApplicationService` our infrastructure layer, so the implementation of our application service is focused on our domain.

```python
#/src/servers/weather_service/infrastructure/application.py

class MCPApplicationService(ABC):
    """Base class for MCP Application Services following DDD patterns.

    Application Services orchestrate domain objects to fulfill use cases
    while remaining independent of infrastructure concerns. This class
    provides the MCP-specific infrastructure setup.

    In DDD terms:
    - Tools map to domain service operations
    - Resources provide read-only access to aggregates
    - Prompts offer templated workflows

    Args:
        mcp: FastMCP server instance for protocol handling
    """

    def __init__(self, mcp: FastMCP):
        self.mcp = mcp

        self._register_tools()
        self._register_resources()
        self._register_prompts()

    @property
    @abstractmethod
    def tools(self) -> List[Tuple[str, str, Callable]]:
        pass

    @property
    @abstractmethod
    def resources(self) -> List[Tuple[str, str, str, Callable]]:
        pass
    @property
    @abstractmethod
    def prompts(self) -> List[Tuple[str, str, Callable]]:
        pass

    def _register_tools(self):
        """Register MCP tools"""
        for name, description, tool in self.tools:
            self.mcp.tool(name=name, description=description)(tool)

    def _register_resources(self):
        """Register MCP resources"""
        for uri, name, description, resource in self.resources:
            self.mcp.resource(uri, name=name, description=description)(resource

    def _register_prompts(self):
        """Register MCP prompts"""
        for name, description, prompt in self.prompts:
            self.mcp.prompt(name=name, description=description)(prompt)

    def run(self, transport: str = "stdio"):
        """Run the MCP server"""
        self.mcp.run(transport=transport)
```

Understanding Our MCP Server's Capabilities

Our domain-driven weather server demonstrates all three MCP capability types:

Tools (Active Operations):

- `get_forecast(latitude, longitude)`: Fetches real-time weather data and persists it for historical tracking
- `get_alerts(state)`: Retrieves current weather alerts and saves them as snapshots

Resources (Contextual Data):

- `historical://alerts/{state}`: Provides read-only access to previously cached alert data
- Resource templates like this enable LLMs to construct parameterized requests (`historical://alerts/CA`)

Prompts (Workflow Templates):

- `weather_analysis_prompt(location)`: Structures comprehensive weather analysis requests
- Prompts guide LLMs toward specific reasoning patterns and output formats

This combination allows for sophisticated interactions: an LLM can use tools for current data, resources for historical context, and prompts for structured analysis — all coordinated through the client.

Building MCP Clients

While our domain-driven server provides clean, well-structured capabilities, the client is equally important — it determines how LLMs discover, select, and coordinate these capabilities to fulfill user requests. The client acts as the intelligent orchestrator, bridging human intent with server capabilities.

In the MCP ecosystem, clients act as the bridge between users and MCP servers. While servers expose capabilities (tools, resources, prompts), clients orchestrate how these capabilities are used:

- **Discovery**: Clients connect to servers and discover available tools, resources, and prompts
- **Selection**: Clients decide which capabilities to use based on user queries
- **Coordination**: Clients manage the conversation flow, calling tools and loading resources
- **Context Management**: Clients control what information gets provided to the LLM

This separation allows the same MCP server to work with different client strategies — from simple tool calling to sophisticated workflow orchestration.

The Model Context Protocol gives us flexibility in client design. We can build simple clients that just call tools, or sophisticated clients that intelligently manage resources and apply workflow templates. Let's explore three different approaches, each suited to different interaction patterns.

Let's explore three different client strategies that showcase MCP's flexibility:

Simple Chat Client: Basic Tool Coordination

This client demonstrates the fundamental MCP interaction pattern: the LLM discovers available tools, decides when to use them based on user queries, and incorporates their results into the conversation flow.

```python
# /src/client/simple_client.py
class SimpleChatMCPClient:
    """Basic MCP client with tool calling and conversation management.

    Provides core MCP functionality:
    - Connects to MCP servers via stdio
    - Discovers and calls tools
    - Manages conversation state with LLM providers
    - Handles interactive chat sessions

    Args:
        llm_provider: LLM service provider (Azure OpenAI, Anthropic, etc.)
    """

    def __init__(self, llm_provider: LLMProvider):
        # Initialize session and client objects
        self.session: Optional[ClientSession] = None
        self.exit_stack = AsyncExitStack()
        self.provider = llm_provider
        self.available_resources = []
        self.resource_templates = []
        self.available_prompts = []

    async def connect_to_server(self, server_script_path: str):
        """Connect to an MCP server"""
        is_python = server_script_path.endswith(".py")
        is_js = server_script_path.endswith(".js")
        if not (is_python or is_js):
            raise ValueError("Server script must be a .py or .js file")

        command = "python" if is_python else "node"
        server_params = StdioServerParameters(
            command=command, args=[server_script_path], env=None
        )

        stdio_transport = await self.exit_stack.enter_async_context(
            stdio_client(server_params)
        )
        self.stdio, self.write = stdio_transport
        self.session = await self.exit_stack.enter_async_context(
            ClientSession(self.stdio, self.write)
        )

        await self.session.initialize()
        await self.provider.initialize()
        await self.discover_tools()
        await self._discover_resources()
        await self._discover_prompts()

    async def discover_tools(self):
        response = await self.session.list_tools()
        self._mcp_server_tools = response.tools
        self._formatted_tools = await self.provider.format_tools_for_model(
            self._mcp_server_tools
        )
        logger.info(
            f"Connected to server with tools: {[tool.name for tool in self._mcp
        )

    async def _discover_resources(self):
        """Discover available resources and resource templates from MCP server.

        MCP servers can expose two types of resources:
        1. Concrete resources: Fixed URIs with static content
        2. Resource templates: URI patterns with parameters (e.g., "user://{id}

        This method queries both endpoints and stores the results for later
        resource selection and loading.

        Populates:
            self.available_resources: List of concrete resource descriptors
            self.resource_templates: List of parameterized resource templates
        """
        if not self.session:
            logger.error("Cannot discover resources: No active session")
            return

        try:
            # First, get concrete resources
            logger.info("Discovering concrete resources...")
            resources_response = await self.session.list_resources()

            # Then, get resource templates
            logger.info("Discovering resource templates...")
            templates_response = await self.session.list_resource_templates()

            # Store both concrete resources and resource templates
            self.available_resources = []
            self.resource_templates = []

            # Process concrete resources
            if hasattr(resources_response, "resources"):
                for resource in resources_response.resources:
                    if hasattr(resource, "uri"):
                        self.available_resources.append(
                            {
                                "uri": resource.uri,
                                "name": resource.name,
                            "description": getattr(resource, "description",
                            "mimeType": getattr(resource, "mimeType", "text
                        }
                    )

            # Process resource templates
            if hasattr(templates_response, "resourceTemplates"):
                for template in templates_response.resourceTemplates:
                    if hasattr(template, "uriTemplate"):
                        self.resource_templates.append(
                            {
                                "uriTemplate": template.uriTemplate,
                                "name": template.name,
                                "description": getattr(template, "description",
                                "mimeType": getattr(template, "mimeType", "text
                            }
                        )

            # Log discovered resources
            logger.info(
                f"Discovered {len(self.available_resources)} concrete resources
            )
            logger.info(f"Discovered {len(self.resource_templates)} resource te

            for template in self.resource_templates:
                logger.info(f"Template: {template['uriTemplate']}")

        except Exception as e:
            logger.error(f"Error discovering resources: {str(e)}")

    async def _discover_prompts(self):
        """Discover prompt templates available from the MCP server.

        Prompts in MCP are user-controlled templates that provide standardized
        ways to initiate specific types of conversations. They can accept
        parameters and return structured prompt content.

        Populates:
            self.available_prompts: List of prompt descriptors with metadata
        """
        if not self.session:
            return

        try:
            prompts_response = await self.session.list_prompts()
            self.available_prompts = []

            if hasattr(prompts_response, "prompts"):
                for prompt in prompts_response.prompts:
                    self.available_prompts.append(
                        {
                            "name": prompt.name,
                            "description": getattr(prompt, "description", ""),
                        "arguments": getattr(prompt, "arguments", []),
                    }
                )

            logger.info(f"Discovered {len(self.available_prompts)} prompts")
        except Exception as e:
            logger.error(f"Error discovering prompts: {str(e)}")

    async def handle_user_query(self, query: str) -> str:
        """Process a query using the LLM provider and available tools"""

        # Add the user's query
        await self.provider.add_user_message(query)

        response_parts = []

        while True:
            # Get next response from provider (text and any tool calls)
            text, tool_calls = await self.provider.get_model_response(
                self._formatted_tools
            )

            if text:
                response_parts.append(text)

            # If no tool calls, we're done
            if not tool_calls:
                break

            # Process each tool call
            for tool_call in tool_calls:
                tool_name = tool_call["name"]
                tool_args = tool_call["arguments"]
                tool_call_id = tool_call.get("id")

                logger.info(f"[Calling tool {tool_name} with args {tool_args}]"

                # Execute tool call via MCP
                result = await self.session.call_tool(tool_name, tool_args)

                # Add tool result to conversation
                await self.provider.process_tool_result(
                    tool_name,
                    result,
                    tool_call_id,
                )

        return "\n".join([text for text in response_parts if text])

    async def start_interactive_session(self):
        """Run an interactive chat loop"""
        print("\nMCP Client Started!")
        print("Type your queries or 'quit' to exit.")
        while True:
            try:
                query = input("\nQuery: ").strip()

                if query.lower() == "quit":
                    await self.cleanup()
                    break

                response = await self.handle_user_query(query)
                print("\n" + response)

            except Exception as e:
                print(f"\nError: {str(e)}")

    async def cleanup(self):
        """Clean up resources"""
        await self.exit_stack.aclose()
```

Now, let’s use our client to interact with our server.

```
python src/chat.py src/run_weather_service.py azure simple

MCP Client Started!
Type your queries or 'quit' to exit.

Query: What is the forecast for Austin Texas?
INFO:httpx:HTTP Request: POST https://dev-ch-uk.openai.azure.com/openai/deploym
INFO:client.simple_client:[Calling tool get_forecast with args {'latitude': 30.
INFO:mcp.server.lowlevel.server:Processing request of type CallToolRequest
INFO:httpx:HTTP Request: GET https://api.weather.gov/points/30.2672,-97.7431 "H
INFO:httpx:HTTP Request: GET https://api.weather.gov/gridpoints/EWX/156,91/fore
INFO:httpx:HTTP Request: POST https://dev-ch-uk.openai.azure.com/openai/deploym

Here is the forecast for Austin, Texas:

- Overnight: Mostly cloudy, low around 76°F. South wind around 5 mph.
- Saturday: Mostly sunny, high near 99°F. Heat index up to 105°F. South wind 5-
- Saturday Night: Mostly cloudy, low around 76°F. Heat index up to 102°F. South
- Sunday: Partly sunny, high near 98°F. Heat index up to 105°F. South wind 5-10
- Sunday Night: Slight chance of showers and thunderstorms after 7pm. Mostly cl

Stay cool and hydrated—it's going to be hot!

Query: Are there any alerts in Texas at the moment?
INFO:httpx:HTTP Request: POST https://dev-ch-uk.openai.azure.com/openai/deploym
INFO:client.simple_client:[Calling tool get_alerts with args {'state': 'TX'}]
```

```
INFO:mcp.server.lowlevel.server:Processing request of type CallToolRequest
INFO:httpx:HTTP Request: GET https://api.weather.gov/alerts/active/area/TX "HTT
INFO:httpx:HTTP Request: POST https://dev-ch-uk.openai.azure.com/openai/deploym

Yes, there are several weather alerts in Texas at the moment:

1. Heat Advisory:
   - Areas: Jim Wells, Inland Kleberg, Inland Nueces, Inland San Patricio Count
   - Heat index values up to 111°F expected from 1 PM to 7 PM CDT today.
   - Impacts: Hot temperatures and high humidity may cause heat illnesses.
   - Advice: Drink plenty of fluids, stay in air conditioning, avoid the sun, a

2. Flood Warning (Neches River near Diboll):
   - Areas: Angelina, Houston, Polk, Trinity, Tyler Counties
   - Minor flooding is occurring and forecast to continue until Sunday evening.
   - Impacts: Flooded boat ramps and trails; minor lowland flooding.
   - Advice: Do not drive through flooded areas. Turn around, don't drown.

3. Flood Warning (Sabine River near Deweyville):
   - Areas: Newton, Orange Counties (TX) and Beauregard, Calcasieu (LA)
   - Minor flooding is occurring and forecast to continue.
   - Impacts: Minor lowland flooding at 24.0 feet river stage.
   - Advice: Stay updated and avoid flooded areas.

If you need more details for a specific area, let me know!
```

That looks like it worked well; successfully using our tools where appropriate!

LLM Resource Selector: AI-Driven Context Management

Now, let's extend our client so that it can access the resource that we defined on our server; using an LLM to select which resources to load.

```python
# /src/client/resource_selector_client.py

class LLMResourceSelector(SimpleChatMCPClient):
    """MCP client that uses LLM intelligence to select relevant resources.

    Extends basic client with smart resource selection. Before processing
    queries, asks the LLM to analyze available resources and load only
    those relevant to the user's request.

    This reduces token usage and improves response quality by providing
    focused context rather than all available resources.
    """

    def __init__(self, llm_provider: LLMProvider):
        super().__init__(llm_provider)
        self.resource_templates = []
        self.available_resources = []

    async def handle_user_query(self, query: str) -> str:
        """
        Process a query using LLM-driven resource selection
        """
        # First, have the LLM analyze the query to determine which resources to
        resource_uris = await self._select_relevant_resources(query)

        if resource_uris:
            logger.info(f"Identified resources as relevant: {resource_uris}")
            # Load the selected resources
            resource_content = await self._load_resources(resource_uris)

            # Enhance the query with the selected resources
            enhanced_query = (
                f"I have the following information that may be relevant to the
                f"{resource_content}\n\n"
                f"Using this information where relevant, please answer: {query}
                "You do not have to use a resource if you don't think it will h
            )

            # Use the enhanced query
            return await super().handle_user_query(enhanced_query)
        else:
            # No relevant resources identified, use the original query
            return await super().handle_user_query(query)

    async def _select_relevant_resources(self, query: str) -> list:
        """
        Have the LLM decide which resources would be relevant for the given que
        Returns a list of resource URIs to load.
        """
        # Create a summary of available resources for the LLM
        resource_descriptions = self._format_resource_descriptions()

        # Create a prompt for resource selection
        resource_selection_prompt = (
            f"I need to determine which resources would help answer this query:
            f"Available resources:\n{resource_descriptions}\n\n"
            f"For resource templates, I need to determine appropriate parameter
            f"Analyze the query and list only the URIs of resources that would
            f"with any parameter substitutions for templates. Format as a JSON
        )

        # Ask the LLM to select resources
        await self.provider.add_user_message(resource_selection_prompt)
        response_text, _ = await self.provider.get_model_response([])

        # Extract the resource URIs from the response
        return self._extract_resource_uris(response_text)
```

For a weather query about California patterns, it might automatically load `historical://alerts/CA` to provide historical context alongside current weather tools, let's try it out.

```
python src/chat.py src/run_weather_service.py azure resource_selector

INFO:client.simple_client:Discovered 1 resource templates
INFO:client.simple_client:Template: historical://alerts/{state}

MCP Client Started!
Type your queries or 'quit' to exit.

Query: Are there any active weather alerts in California?
INFO:httpx:HTTP Request: POST https://dev-ch-uk.openai.azure.com/openai/deploym
INFO:httpx:HTTP Request: POST https://dev-ch-uk.openai.azure.com/openai/deploym
INFO:client.simple_client:[Calling tool get_alerts with args {'state': 'CA'}]
INFO:mcp.server.lowlevel.server:Processing request of type CallToolRequest
INFO:httpx:HTTP Request: GET https://api.weather.gov/alerts/active/area/CA "HTT
INFO:httpx:HTTP Request: POST https://dev-ch-uk.openai.azure.com/openai/deploym

Yes, there are several active weather alerts in California:

- Wind Advisories for San Gorgonio Pass near Banning and Santa Barbara County S
- Beach Hazards Statements for Ventura County, Malibu Coast, Los Angeles County

If you need details about a specific area or alert, let me know!

Query: Do you have access to historical weather alerts in the same area?
INFO:httpx:HTTP Request: POST https://dev-ch-uk.openai.azure.com/openai/deploym
INFO:client.resource_selector_client:Identified resources as relevant: ['histor
INFO:mcp.server.lowlevel.server:Processing request of type ReadResourceRequest
INFO:httpx:HTTP Request: POST https://dev-ch-uk.openai.azure.com/openai/deploym

Yes, I have access to historical weather alerts for California. For example, I

Query: Can you give me an example of a historical weather alert in California,
INFO:httpx:HTTP Request: POST https://dev-ch-uk.openai.azure.com/openai/deploym
INFO:client.resource_selector_client:Identified resources as relevant: ['histor
INFO:mcp.server.lowlevel.server:Processing request of type ReadResourceRequest
INFO:httpx:HTTP Request: POST https://dev-ch-uk.openai.azure.com/openai/deploym

Yes, here is an example of a historical weather alert in California, including
```

```
Event: Wind Advisory
Area: San Gorgonio Pass Near Banning
Severity: Moderate
Description: West winds 25 to 35 mph with gusts up to 55 mph expected.
When: Until 3 AM PDT Sunday (active as of May 24, 2025)
Impacts: Gusty winds will blow around unsecured objects. Reduced visibility in
Instructions: Winds this strong can make driving difficult, especially for high

This alert was active around May 24, 2025.
```

Important Note: In MCP, it's the **client's responsibility** to provide resources to the LLM context. Here we've chosen to use an LLM to analyze queries and select appropriate resources, but this is just one approach. You could equally implement resource selection using:

- **Regex patterns**: Match query keywords to resource categories
- **Business rules**: "Always load historical data for trend analysis queries"
- **Configuration-driven selection**: User-defined mappings between query types and resources
- **Hybrid approaches**: Combine multiple selection strategies

The key insight is that MCP provides the resource infrastructure, but the selection strategy is entirely up to your client implementation.

Prompt-Aware Client: Template-Driven Workflows

This client demonstrates MCP's prompt system, automatically recognizing when users want to invoke structured workflows and applying the appropriate templates to guide the LLM's response.

```
class PromptAwareMCPClient(SimpleChatMCPClient):
    """Extends SimpleChatMCPClient with prompt discovery and usage"""

    async def start_interactive_session(self):
        """Show available prompts at start of interaction"""
        print("\nMCP Client Started!")

        # Show available prompts
        if self.available_prompts:
            print("\nAvailable MCP Prompts:")
            for i, prompt in enumerate(self.available_prompts, 1):
                args = ", ".join([f"{arg.name}" for arg in prompt.get("argument
                print(f"{i}. {prompt['name']} - {prompt['description']}")
            print(
                "\nYou can use these prompts in your queries by calling them li
            )

        # Use the parent class's interactive session logic
        await super().start_interactive_session()

    async def handle_user_query(self, query: str) -> str:
        """
        Process a query using LLM-driven resource selection
        """
        # First, have the LLM analyze the query to determine which resources to
        prompt_info = await self._select_relevant_prompt(query)

        if prompt_info:
            logger.info(f"Identified prompt call as: {prompt_info}")
            # Load the selected resources

            enhanced_query = await self.use_prompt(prompt_info)

            # Use the enhanced query
            return await super().handle_user_query(enhanced_query)
        else:
            # No relevant resources identified, use the original query
            return await super().handle_user_query(query)

    async def _select_relevant_prompt(self, query: str) -> list:
        """
        Have the LLM decide which resources would be relevant for the given que
        Returns a list of resource URIs to load.
        """

        # Create a prompt for prompt selection
        prompt_selection_prompt = (
            f"I need to determine whether the user is trying to use an availabl
            f"Their query is: '{query}'\n\n"
            f"The available prompts you have are:\n{self.available_prompts}\n\n
            f"Only use a prompt if the user explicitly asked for it "
            f"If they are trying to use a prompt, please return the name of the
            f'that should be used as Json with format {{"prompt_name": ..., "ar
            f"If they are not trying to use a prompt, please return an empty Js
        )

        # Ask the LLM to select resources
        await self.provider.add_user_message(prompt_selection_prompt)
        response_text, _ = await self.provider.get_model_response([])

        # Extract the prompt name and arguments from the response
        try:
            response_json = json.loads(response_text)
```

```
                if isinstance(response_json, dict):
                    if not response_json:
                        return []
                    prompt_name = response_json.get("prompt_name")
                    arguments = response_json.get("arguments", {})
                    if prompt_name and isinstance(arguments, dict):
                        return [prompt_name, arguments]
            except json.JSONDecodeError:
                logger.debug(f"Failed to parse JSON response: {response_text}")
                return []

    async def use_prompt(self, prompt_info) -> str:
        """Use an MCP prompt with given arguments"""
        try:
            prompt_name, arguments = prompt_info
            prompt_result = await self.session.get_prompt(prompt_name, argument

            return prompt_result.messages[0].content.text
        except Exception as e:
            return f"Error using prompt '{prompt_info}': {str(e)}"
```

Flexible Prompt Selection: As with resource selection, we've chosen to use an LLM to detect when users want to invoke prompt templates, but this is just one approach. Alternative strategies include:

- **Keyword matching**: Detect phrases like "analyze", "report on", or "summarize"
- **Command prefixes**: Users type a specified format, e.g., `/weather-analysis Los Angeles` to explicitly invoke prompts
- **Menu-driven selection**: Present available prompts and let users choose
- **Context-aware detection**: Analyze conversation history to suggest relevant prompts

Let's try it out!

```
python src/chat.py src/run_weather_service.py azure prompt_aware
INFO:mcp.server.lowlevel.server:Processing request of type ListPromptsRequest
INFO:client.simple_client:Discovered 1 prompts

MCP Client Started!

Available MCP Prompts:
```

```
1. weather_analysis - Analyze weather conditions for a location
You can use these prompts in your queries by calling them like Python functions

MCP Client Started!
Type your queries or 'quit' to exit.

Query: weather_analysis(Austin, Texas)
INFO:httpx:HTTP Request: POST https://dev-ch-uk.openai.azure.com/openai/deploym
INFO:client.prompt_aware_client:Identified prompt call as: ['weather_analysis',
INFO:mcp.server.lowlevel.server:Processing request of type GetPromptRequest
INFO:httpx:HTTP Request: POST https://dev-ch-uk.openai.azure.com/openai/deploym
INFO:client.simple_client:[Calling tool get_forecast with args {'latitude': 30.
INFO:mcp.server.lowlevel.server:Processing request of type CallToolRequest
INFO:httpx:HTTP Request: GET https://api.weather.gov/points/30.2672,-97.7431 "H
INFO:httpx:HTTP Request: GET https://api.weather.gov/gridpoints/EWX/156,91/fore
INFO:client.simple_client:[Calling tool get_alerts with args {'state': 'TX'}]
INFO:mcp.server.lowlevel.server:Processing request of type CallToolRequest
INFO:httpx:HTTP Request: GET https://api.weather.gov/alerts/active/area/TX "HTT
INFO:httpx:HTTP Request: POST https://dev-ch-uk.openai.azure.com/openai/deploym

Weather Analysis for Austin, Texas
1. Current Temperatures and Conditions
- Overnight: Mostly cloudy, low around 76°F. Light south wind (~5 mph).
- Saturday: Mostly sunny, high near 99°F. Heat index values as high as 105°F. S
- Saturday Night: Mostly cloudy, low around 76°F. Heat index up to 102°F. South

2. Expected Changes Over the Next Few Days
- Sunday: Partly sunny, high near 98°F. Heat index values as high as 105°F. Sou
- Sunday Night: Mostly cloudy, low around 77°F. Slight chance (20%) of showers

3. Notable Weather Patterns or Anomalies
- Persistent high temperatures near 100°F with heat index values exceeding 100°
- Winds will be moderate, with occasional gusts up to 25 mph.
- A slight chance of showers and thunderstorms late Sunday night, but otherwise
- No active weather alerts specifically for Austin, but a Heat Advisory is in e

4. Practical Advice Based on Conditions
- Heat Safety: The combination of high temperatures and humidity can lead to he
- Monitor Vulnerable Groups: Check on elderly neighbors, children, and pets, as
- Prepare for Gusty Winds: Secure loose outdoor items.
- Stay Informed: While no severe weather is forecast for Austin, keep an eye on
- If you must be outdoors, wear light clothing, use sunscreen, and take frequen

Summary
Austin is experiencing a period of intense heat with little relief at night and
```

Here, we can see that our model used both available tools to provide a comprehensive weather report!

Benefits Realized Through Domain-Driven Design

The beauty of this architecture is that each component has a single, clear responsibility:

- **Domain models** represent weather concepts using business language
- **Domain services** encapsulate weather-specific business rules
- **Repositories** manage data persistence with time-based queries
- **Application services** orchestrate domain operations for MCP
- **Infrastructure** handles external concerns like HTTP and file I/O
- **Clients** coordinate between user intent and server capabilities

This refactored architecture delivers significant advantages across multiple dimensions:

Testability at Every Layer

Each layer can be tested independently with appropriate test doubles:

```
# Test domain logic without external dependencies using a STUB
async def test_weather_service_handles_api_errors():
    # Stub that returns hardcoded responses to simulate API failure
    async def http_stub(url, headers=None):
        return None  # Hardcoded failure response

    weather_service = NWSWeatherService(http_stub)
    forecast = await weather_service.get_forecast(37.7749, -122.4194)

    # Verify business logic handles errors correctly
    assert forecast.error is not None
    assert len(forecast.periods) == 0
    assert forecast.latitude == 37.7749

# Test application layer with FAKE domain services
async def test_mcp_service_saves_forecasts():
    # Fake provides working implementation with simplified behavior
    class FakeWeatherService(WeatherService):
        async def get_forecast(self, lat: float, lon: float) -> Forecast:
            return Forecast(
                periods=[WeatherPeriod(name="Test", temperature=72.0, ...)],
                error=None, latitude=lat, longitude=lon, retrieved_at=datetime.
            )

    fake_weather_service = FakeWeatherService()
```

```
        fake_repository = FakeWeatherRepository()

        mcp_service = WeatherMCPService(
            mcp, fake_weather_service, fake_repository, fake_alert_repo
        )

        # Test MCP orchestration without external dependencies
        result = await mcp_service.get_forecast(37.7749, -122.4194)

        assert "San Francisco" in result
        assert fake_repository.save_forecast_called
```

Infrastructure Flexibility

Need to switch from JSON file storage to Redis? Just implement the repository interface:

```
class RedisWeatherForecastRepository(WeatherForecastRepository):
    """Redis-backed implementation of weather forecast repository."""

    def __init__(self, redis_client):
        self.redis = redis_client

    async def get_forecasts(self, latitude: float, longitude: float, **kwargs):
        # Same interface, different storage mechanism
        key = f"forecasts:{latitude}:{longitude}"
        data = await self.redis.get(key)
        return self._deserialize_forecasts(data) if data else []

    async def save_forecast(self, latitude: float, longitude: float, forecast:
        key = f"forecasts:{latitude}:{longitude}"
        serialized = self._serialize_forecast(forecast)
        await self.redis.lpush(key, serialized)
        await self.redis.expire(key, 86400)  # 24 hour TTL
```

The domain and application layers remain completely unchanged — only the infrastructure implementation changes.

Enhanced LLM Understanding Through Domain Language

Well-modeled domains with consistent terminology directly improve how LLMs reason about capabilities. Compare these two resource descriptions:

```
# Poor domain modeling
"endpoint://data/fetch?type=weather&format=json&location={coords}&time=historic

# Good domain modeling with ubiquitous language
"historical://alerts/{state} - Get historical weather alerts for a US state"
```

The second immediately conveys to an LLM that:

- This provides **historical** (not current) data
- It's about weather **alerts** (not forecasts)
- It's organized by US **states** (not coordinates)
- Parameter should be a state code (not lat/lon)

This semantic clarity helps LLMs choose appropriate resources and construct valid requests.

Clear Architectural Boundaries

The layered architecture provides clear separation of concerns that align with DDD principles:

Domain Layer (`domain/`):

- Contains weather-specific repository implementations that use infrastructure
- `WeatherService` , `Forecast` , `WeatherAlert` domain models and services
- Weather-specific business logic and data access patterns
- Can import from infrastructure but stays focused on weather concepts

Application Layer (`application/`):

- Contains `WeatherMCPService`
- Orchestrates domain objects to fulfill use cases
- Handles MCP protocol translation

- Coordinates between domain services and repositories

Infrastructure Layer (`infrastructure/`):

- Contains generic, reusable technical components
- HTTP clients, file I/O, serialization patterns
- `TimestampedCollectionRepository`, `JsonFileRepository` base classes
- Can be used by any domain, contains no weather-specific logic

This separation ensures changes in one layer don't cascade throughout the system, making the codebase easier to understand, modify, and maintain.

Conclusion: Building AI Systems That Scale

Domain-Driven Design and the Model Context Protocol form a powerful partnership for building maintainable AI integrations. By organizing our code around business domains rather than technical concerns, we create systems that are not only easier to understand and modify, but also provide LLMs with the clear, semantically rich interfaces they need to operate effectively.

The patterns demonstrated here — ubiquitous language, bounded contexts, domain services, repositories, and application services — scale from simple weather lookups to complex enterprise integrations. As the AI ecosystem continues evolving, having these architectural foundations ensures your MCP servers can adapt and grow with changing requirements.

The Key Insight: Domain Language as the Bridge

The most important insight from this exercise is that DDD's emphasis on domain modelling aligns perfectly with what makes LLMs effective: consistent terminology, well-defined capabilities, and semantic clarity. When we build systems that speak the language of the business domain, we create interfaces that both humans and AI can understand and use effectively.

Consider the evolution from our original implementation:

```
# Before: Technical jargon, mixed concerns
@mcp.tool()
async def get_alerts(state: str) -> str:
    url = f"{NWS_API_BASE}/alerts/active/area/{state}"
    data = await make_nws_request(url)
    # ... URL construction, HTTP calls, JSON parsing all mixed together
```

To our domain-driven approach:

```
# After: Clear domain language, separated concerns
async def get_alerts(self, state: str) -> str:
    """Get weather alerts for a US state."""
    alerts = await self.weather_service.get_alerts(state)
    await self.weather_alert_repository.save_alerts(state, alerts)
    return "\n---\n".join(alert.to_display_string() for alert in alerts)
```

The second version immediately communicates its intent: "Get weather alerts, save them for historical analysis, and format them for presentation." The domain language makes both the purpose and the process clear.

Looking Forward: The Future of AI Integration

As AI systems become more sophisticated, the importance of well-designed integration points will only grow. The ad-hoc, tightly-coupled approaches that might work for proof-of-concepts will become maintenance nightmares in production systems.

The architectural patterns we've explored here — clear domain boundaries, dependency injection, repository abstractions, and protocol separation — provide a foundation that can evolve with changing requirements. Whether you're integrating with new AI models, switching data sources, or scaling to handle enterprise workloads, these patterns ensure your systems remain flexible and maintainable.

Whether we are building your first MCP server or refactoring existing AI integrations, remember that the time invested in proper domain modeling and clean architecture pays dividends in maintainability, testability, and the overall quality of AI interactions with your systems. The future of AI integration lies not in ad-hoc connections, but in thoughtfully designed systems that bridge the gap between human intent and machine capability through shared domain understanding.

Mcp Server Mcp Client Domain Driven Design Llm Agentic Ai

Follow

Written by Chris Hughes

1.1K followers · 0 following

Principal Machine Learning Engineer/Scientist Manager at Microsoft. All opinions are my own.

Responses (2)

Bgerby

What are your thoughts?

Karunsharma
Jul 16

surely going to try and implement this with local ollama models

1 Reply

Benjamin Keen
Jun 11

Really nice article (as usual) Chris; just finished reading - I might try and implement this myself for another domain.

1 Reply

More from Chris Hughes

Chris Hughes

A Brief Overview of Cross Entropy Loss

A refresher on a commonly used Loss Function

Sep 25, 2024 83 3

In TDS Archive by Chris Hughes

Demystifying PyTorch's WeightedRandomSampler by example

A straightforward approach to dealing with imbalanced datasets

Aug 30, 2022 338 6

Chris Hughes

Understanding PPO: A Game-Changer in AI Decision-Making Explained for RL Newcomers

From Theory to Implementation: A Comprehensive Guide to Reinforcement Learning's Game-Changing Algorithm

Sep 10, 2024 332 5

Chris Hughes

Rethinking Object Detection as Language Modelling: Lessons from Reimplementing Pix2Seq

What I learned building Google's radical approach to object detection from scratch—and why it changed how I think about computer vision

Jul 2 223 4

See all from Chris Hughes

Recommended from Medium

Saima Khan

From Monolith to Modular (Part 2): How I Integrated MCP Using FastMCP and LangGraph

Plug-and-Play AI with FastMCP and LangGraph

Jul 11 5

Sanath Shetty

Exposing MCP Servers as APIs: Building Bridges Between AI Models and Applications

In the rapidly evolving landscape of AI integration, developers face a common challenge: how to efficiently connect AI models with…

May 19 2

Rishav Paul

Integrating Model Context Protocol (MCP) Servers in your backend code with Open AI’s Agent SDK

AI agents are popping up everywhere, and they often need to use outside tools or data to do their jobs. An MCP Server (Model Context…

Apr 20

Anil Jain | AI / ML Architect | Data Architect

Agentic MCP and A2A Architecture: A Comprehensive Guide

1. Introduction

Apr 14 27 3

Edwin Lisowski

MCP Explained: The New Standard Connecting AI to Everything

How Model Context Protocol is making AI agents actually do things

Apr 15 1.6K 30

Sanjeeb Panda

Model Context Protocol (MCP) in AI Agent Development :Text To Sql

Introduction

Apr 17 39 2

See more recommendations