MCP-Based DataOps Assistant - Complete Implementation Guide

Document Overview

This document provides end-to-end, low-level architecture and complete implementation specifications for building an MCP-based DataOps Assistant that can understand, inspect, and reason over Azure Data Factory pipelines, Key Vault secrets, infrastructure code, and operational logs[1][2].

Target Audience: AI code generation tools (Antigravity), developers implementing the system.

Implementation Scope: Complete working system with real Azure SDK integrations, not dummy/mock implementations.

---

1. System Architecture

1.1 High-Level Architecture

The system consists of four primary layers[2]:

- **Frontend Layer**: React-based chat UI for user interaction
- **Backend Layer**: FastAPI service orchestrating LLM and MCP server communication
- **MCP Server Layer**: Python service exposing tools via Model Context Protocol
- **Enterprise Systems Layer**: Azure Data Factory, Key Vault, Storage, Terraform, logs

Data Flow:

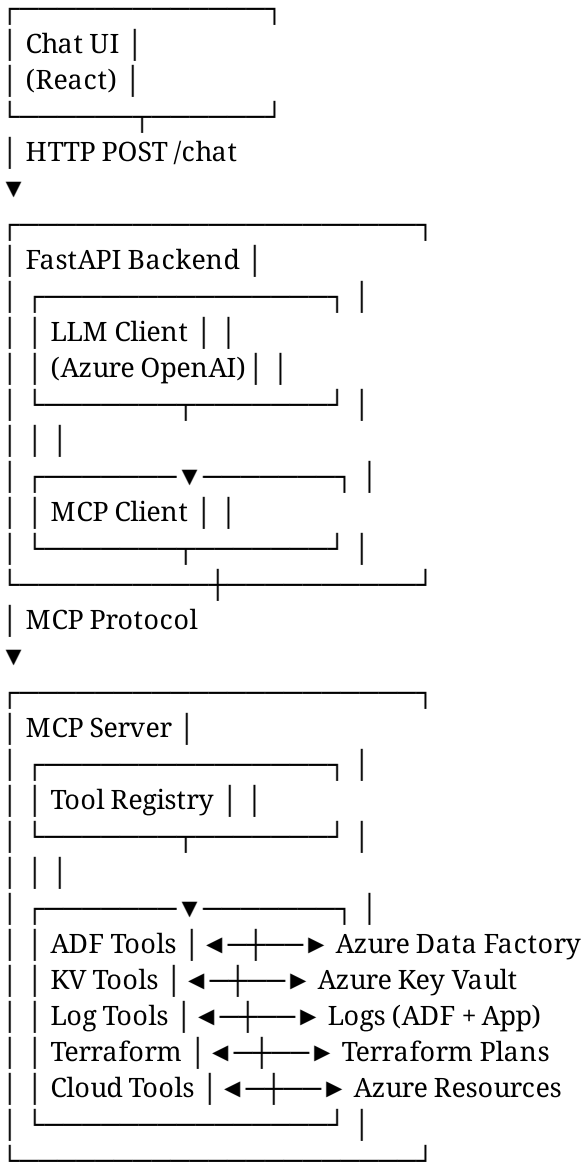
1. User sends natural language query via chat UI
2. Backend forwards to LLM with available MCP tool schemas
3. LLM decides which tools to invoke based on query context
4. MCP server executes tools against real Azure resources
5. Tool results flow back to LLM for reasoning
6. LLM generates evidence-based explanation
7. Backend returns response with tool execution traces to frontend

1.2 Technology Stack

Layer	Technology
MCP Server	Python 3.10+, MCP SDK
Backend API	FastAPI, Uvicorn
Frontend	React 18+, Vite, TailwindCSS
Azure SDKs	azure-identity, azure-mgmt-datafactory, azure-keyvault-secrets
LLM	Azure OpenAI (GPT-4)
Infrastructure	Terraform, Azure RM
Logging	Python logging, Azure Monitor

Table 1: Technology stack by layer

1.3 Component Interaction Diagram



2. Project Structure

2.1 Repository Layout

```
mcp-dataops-assistant/
├── backend/
│   ├── app/
│   │   ├── __init__.py
│   │   ├── main.py # FastAPI application entry
│   │   ├── config.py # Configuration management
│   │   ├── schemas.py # Pydantic models
│   │   ├── llm_client.py # Azure OpenAI integration
│   │   ├── mcp_client.py # MCP protocol client
│   │   └── routers/
│   │   ├── __init__.py
│   │   └── chat.py # Chat endpoint
│   ├── requirements.txt
│   └── .env.example
│
├── mcp_server/
│   ├── __init__.py
│   ├── main.py # MCP server entry
│   ├── config.py # Server configuration
│   ├── protocol.py # MCP protocol implementation
│   ├── tool_registry.py # Tool registration system
│   ├── tools/
│   │   ├── __init__.py
│   │   ├── adf_tools.py # Azure Data Factory tools
│   │   ├── keyvault_tools.py # Key Vault tools
│   │   ├── log_tools.py # Log intelligence tools
│   │   ├── terraform_tools.py # Terraform reasoning tools
│   │   └── cloud_tools.py # Cloud resource tools
│   ├── utils/
│   │   ├── __init__.py
│   │   ├── azure_client.py # Azure SDK client factory
│   │   ├── log_parser.py # Log parsing utilities
│   │   └── terraform_parser.py # Terraform plan parser
│   ├── models/
│   │   ├── __init__.py
│   │   ├── tool_schemas.py # MCP tool input/output schemas
│   │   └── azure_models.py # Azure resource models
│   ├── requirements.txt
│   └── .env.example
│
├── frontend/
│   ├── src/
│   │   ├── App.jsx
│   │   ├── main.jsx
│   │   ├── components/
│   │   │   ├── ChatWindow.jsx
```

```
│   │   │   ├── InputBar.jsx
│   │   │   ├── EnvironmentSelector.jsx
│   │   │   └── ToolTracePanel.jsx
│   │   ├── services/
│   │   │   └── api.js # Backend API client
│   │   └── styles/
│   │   └── index.css
│   ├── package.json
│   ├── vite.config.js
│   └── .env.example
│
├── infra/
│   ├── terraform/
│   │   ├── main.tf
│   │   ├── providers.tf
│   │   ├── variables.tf
│   │   └── outputs.tf
│   └── plans/ # Terraform plan files
│
├── logs/ # Application logs
├── tests/ # Unit and integration tests
├── docs/ # Additional documentation
├── docker-compose.yml
├── Dockerfile.backend
├── Dockerfile.mcp
└── README.md
```

---

3. Configuration Management

3.1 Environment Variables

All secrets and configuration are loaded from `.env` files.

Backend `.env` (backend/.env):

Azure Authentication

```
AZURE_TENANT_ID=your-tenant-id
AZURE_CLIENT_ID=your-client-id
AZURE_CLIENT_SECRET=your-client-secret
AZURE_SUBSCRIPTION_ID=your-subscription-id
```

Azure OpenAI

```
AZURE_OPENAI_ENDPOINT=https://your-openai.openai.azure.com/
AZURE_OPENAI_API_KEY=your-openai-key
AZURE_OPENAI_DEPLOYMENT_NAME=gpt-4
AZURE_OPENAI_API_VERSION=2024-02-15-preview
```

MCP Server

MCP_SERVER_URL=http://localhost:8001

Application

ENVIRONMENT=dev
LOG_LEVEL=INFO

MCP Server `.env` (`mcp_server/.env`):

Azure Authentication (same as backend)

AZURE_TENANT_ID=your-tenant-id
AZURE_CLIENT_ID=your-client-id
AZURE_CLIENT_SECRET=your-client-secret
AZURE_SUBSCRIPTION_ID=your-subscription-id

Azure Resources

AZURE_RESOURCE_GROUP=rg-mcp-demo
AZURE_DATA_FACTORY_NAME=adf-mcp-demo
AZURE_KEY_VAULT_NAME=kv-mcp-demo
AZURE_STORAGE_ACCOUNT_NAME=stmdatamcpdemo

Terraform

TERRAFORM_PLANS_DIR=../infra/plans

Application

LOG_LEVEL=INFO
LOG_FILE_PATH=../logs/mcp_server.log

Frontend `.env` (`frontend/.env`):

VITE_API_BASE_URL=http://localhost:8000

3.2 Configuration Module Implementation

File: `backend/app/config.py`

from pydantic_settings import BaseSettings
from functools import lru_cache

class Settings(BaseSettings):
# Azure Authentication
azure_tenant_id: str

azure_client_id: str
azure_client_secret: str
azure_subscription_id: str

```
    # Azure OpenAI
    azure_openai_endpoint: str
    azure_openai_api_key: str
    azure_openai_deployment_name: str
    azure_openai_api_version: str = "2024-02-15-preview"

    # MCP Server
    mcp_server_url: str = "http://localhost:8001"

    # Application
    environment: str = "dev"
    log_level: str = "INFO"

    class Config:
        env_file = ".env"
        case_sensitive = False
```

@lru_cache()
def get_settings() -> Settings:
return Settings()

File: mcp_server/config.py

from pydantic_settings import BaseSettings
from functools import lru_cache
from pathlib import Path

class MCPSettings(BaseSettings):
# Azure Authentication
azure_tenant_id: str
azure_client_id: str
azure_client_secret: str
azure_subscription_id: str

```
    # Azure Resources
    azure_resource_group: str
    azure_data_factory_name: str
    azure_key_vault_name: str
    azure_storage_account_name: str
```

```
    # Terraform
    terraform_plans_dir: Path = Path("../infra/plans")

    # Application
    log_level: str = "INFO"
    log_file_path: Path = Path("../logs/mcp_server.log")

    class Config:
        env_file = ".env"
        case_sensitive = False
```

@lru_cache()
def get_settings() -> MCPSettings:
return MCPSettings()

---

4. MCP Server Implementation

4.1 Azure Client Factory

Centralized Azure SDK client creation with credential management.

File: `mcp_server/utils/azure_client.py`

from azure.identity import ClientSecretCredential, DefaultAzureCredential
from azure.mgmt.datafactory import DataFactoryManagementClient
from azure.keyvault.secrets import SecretClient
from azure.mgmt.resource import ResourceManagementClient
from functools import lru_cache
from mcp_server.config import get_settings

class AzureClientFactory:
def **init**(self):
self.settings = get_settings()
self._credential = None

```
    @property
    def credential(self):
        """Get Azure credential (cached)"""
        if self._credential is None:
            if self.settings.azure_client_id and self.settings.azure_client_secret:
                self._credential = ClientSecretCredential(
                    tenant_id=self.settings.azure_tenant_id,
```

```
                client_id=self.settings.azure_client_id,
                client_secret=self.settings.azure_client_secret
            )
        else:
            self._credential = DefaultAzureCredential()
    return self._credential

@lru_cache()
def get_datafactory_client(self) -> DataFactoryManagementClient:
    """Get Data Factory management client"""
    return DataFactoryManagementClient(
        credential=self.credential,
        subscription_id=self.settings.azure_subscription_id
    )

@lru_cache()
def get_keyvault_client(self) -> SecretClient:
    """Get Key Vault secret client"""
    vault_url = f"https://{self.settings.azure_key_vault_name}.vault.azure.net"
    return SecretClient(
        vault_url=vault_url,
        credential=self.credential
    )

@lru_cache()
def get_resource_client(self) -> ResourceManagementClient:
    """Get Resource management client"""
    return ResourceManagementClient(
        credential=self.credential,
        subscription_id=self.settings.azure_subscription_id
    )
```

Global instance

azure_clients = AzureClientFactory()

4.2 Tool Schema Definitions

File: `mcp_server/models/tool_schemas.py`

```
from pydantic import BaseModel, Field
from typing import List, Optional, Dict, Any
from datetime import datetime
from enum import Enum

# Enums

class PipelineStatus(str, Enum):
    SUCCEEDED = "Succeeded"
    FAILED = "Failed"
    IN_PROGRESS = "InProgress"
    QUEUED = "Queued"
    CANCELLED = "Cancelled"

class LogSource(str, Enum):
    ADF = "adf"
    APP = "app"

class LogLevel(str, Enum):
    ERROR = "Error"
    WARNING = "Warning"
    INFO = "Info"

class TerraformAction(str, Enum):
    CREATE = "create"
    UPDATE = "update"
    DELETE = "delete"
    NO_OP = "no-op"

# Pipeline Tool Schemas

class GetPipelineStatusInput(BaseModel):
    pipeline_name: str = Field(..., description="Name of the ADF pipeline")
    environment: str = Field(default="dev", description="Environment (dev/prod)")

class PipelineRunInfo(BaseModel):
    run_id: str
    pipeline_name: str
    status: PipelineStatus
    start_time: datetime
    end_time: Optional[datetime]
    duration_seconds: Optional[float]
    error_message: Optional[str]

class GetPipelineStatusOutput(BaseModel):
    pipeline_name: str
    last_run_status: PipelineStatus
```

last_run_start: datetime
last_run_end: Optional[datetime]
last_success_time: Optional[datetime]
last_failure_reason: Optional[str]
recent_runs: List[PipelineRunInfo]

class GetPipelineDependenciesInput(BaseModel):
pipeline_name: str
environment: str = Field(default="dev")

class PipelineDependency(BaseModel):
pipeline_name: str
dependency_type: str # "upstream", "downstream", "dataset"
resource_name: str

class GetPipelineDependenciesOutput(BaseModel):
pipeline_name: str
upstream_pipelines: List[str]
downstream_pipelines: List[str]
datasets_consumed: List[str]
datasets_produced: List[str]
linked_services: List[str]

class GetFailedTasksSummaryInput(BaseModel):
pipeline_name: str
time_window_hours: int = Field(default=24, description="Time window in hours")

class FailedTask(BaseModel):
activity_name: str
error_code: str
error_message: str
failure_count: int
first_failure: datetime
last_failure: datetime

class GetFailedTasksSummaryOutput(BaseModel):
pipeline_name: str
time_window_hours: int
total_failures: int
failed_tasks: List[FailedTask]

Key Vault Tool Schemas

class GetKeyVaultSecretsInput(BaseModel):
prefix: Optional[str] = Field(None, description="Filter secrets by name prefix")
include_high_risk: bool = Field(default=True, description="Include high-risk secrets")

class SecretInfo(BaseModel):
name: str
enabled: bool
created_on: datetime
updated_on: datetime

tags: Dict[str, str]
risk_level: Optional[str]

class GetKeyVaultSecretsOutput(BaseModel):
secrets: List[SecretInfo]
total_count: int

class GetSecretUsageInput(BaseModel):
secret_name: str

class SecretUsage(BaseModel):
pipeline_name: str
linked_service_name: str
environment: str
is_production_critical: bool

class GetSecretUsageOutput(BaseModel):
secret_name: str
usage_count: int
usages: List[SecretUsage]

Log Tool Schemas

class FetchLogsInput(BaseModel):
source: LogSource
pipeline_name: Optional[str] = None
run_id: Optional[str] = None
time_start: Optional[datetime] = None
time_end: Optional[datetime] = None
level: Optional[LogLevel] = None

class LogEntry(BaseModel):
timestamp: datetime
level: LogLevel
source: LogSource
message: str
pipeline_name: Optional[str]
run_id: Optional[str]
activity_name: Optional[str]
error_code: Optional[str]
metadata: Dict[str, Any] = {}

class FetchLogsOutput(BaseModel):
logs: List[LogEntry]
total_count: int

class SummarizeErrorLogsInput(BaseModel):
logs: Optional[List[LogEntry]] = None
source: Optional[LogSource] = None
pipeline_name: Optional[str] = None
time_window_hours: int = 24

class ErrorCluster(BaseModel):
cluster_id: str
error_pattern: str
sample_message: str
count: int
first_occurrence: datetime
last_occurrence: datetime
affected_pipelines: List[str]

class SummarizeErrorLogsOutput(BaseModel):
total_errors: int
clusters: List[ErrorCluster]
anomalies: List[str]

Terraform Tool Schemas

class ParseTerraformPlanInput(BaseModel):
plan_path: str = Field(..., description="Path to terraform plan JSON file")

class ResourceChange(BaseModel):
resource_type: str
resource_name: str
address: str
actions: List[TerraformAction]
before: Optional[Dict[str, Any]]
after: Optional[Dict[str, Any]]
is_destructive: bool

class ParseTerraformPlanOutput(BaseModel):
plan_path: str
created_resources: List[ResourceChange]
updated_resources: List[ResourceChange]
deleted_resources: List[ResourceChange]
high_risk_changes: List[ResourceChange]

class DetectInfraDriftInput(BaseModel):
resource_group_name: str
plan_path: Optional[str] = None

class DriftItem(BaseModel):
resource_type: str
resource_name: str
drift_type: str # "extra_in_cloud", "missing_in_cloud", "configuration_drift"
details: str

class DetectInfraDriftOutput(BaseModel):
has_drift: bool
drift_items: List[DriftItem]

Cloud Resource Tool Schemas

class ListResourcesByTagInput(BaseModel):
tag_key: str
tag_value: str
resource_group: Optional[str] = None

class ResourceInfo(BaseModel):
resource_id: str
resource_name: str
resource_type: str
location: str
tags: Dict[str, str]

class ListResourcesByTagOutput(BaseModel):
resources: List[ResourceInfo]
count: int

4.3 Azure Data Factory Tools Implementation

File: mcp_server/tools/adf_tools.py

import json
from typing import List, Optional, Dict, Any
from datetime import datetime, timedelta
from azure.mgmt.datafactory.models import PipelineRun
from mcp_server.utils.azure_client import azure_clients
from mcp_server.models.tool_schemas import (
GetPipelineStatusInput, GetPipelineStatusOutput, PipelineRunInfo,
GetPipelineDependenciesInput, GetPipelineDependenciesOutput,
GetFailedTasksSummaryInput, GetFailedTasksSummaryOutput, FailedTask,
PipelineStatus
)
from mcp_server.config import get_settings
import logging

logger = logging.getLogger(**name**)
settings = get_settings()

class ADFTools:
"""Azure Data Factory tools implementation"""

```
def __init__(self):
    self.df_client = azure_clients.get_datafactory_client()
    self.resource_group = settings.azure_resource_group
    self.factory_name = settings.azure_data_factory_name

def get_pipeline_status(self, input_data: GetPipelineStatusInput) -> GetPipelineS
```

```
    """
    Get the current status and recent run history of an ADF pipeline.

    Implementation:
    1. Query pipeline runs from ADF
    2. Sort by start time descending
    3. Extract last run details
    4. Find last successful and failed runs
    5. Return structured status
    """
    try:
        logger.info(f"Fetching status for pipeline: {input_data.pipeline_name}")

        # Calculate time range (last 7 days)
        end_time = datetime.utcnow()
        start_time = end_time - timedelta(days=7)

        # Query pipeline runs
        filter_params = {
            'last_updated_after': start_time,
            'last_updated_before': end_time
        }

        runs = list(self.df_client.pipeline_runs.query_by_factory(
            resource_group_name=self.resource_group,
            factory_name=self.factory_name,
            filter_parameters=filter_params
        ).value)

        # Filter for specific pipeline
        pipeline_runs = [
            run for run in runs
            if run.pipeline_name == input_data.pipeline_name
        ]

        # Sort by start time descending
        pipeline_runs.sort(key=lambda x: x.run_start, reverse=True)
```

```
    if not pipeline_runs:
        raise ValueError(f"No runs found for pipeline: {input_data.pipeline_nar

    # Get last run
    last_run = pipeline_runs[0]

    # Find last success and failure
    last_success = next(
        (run for run in pipeline_runs if run.status == "Succeeded"),
        None
    )
    last_failure = next(
        (run for run in pipeline_runs if run.status == "Failed"),
        None
    )

    # Convert to PipelineRunInfo objects
    recent_runs = []
    for run in pipeline_runs[:10]:  # Last 10 runs
        duration = None
        if run.run_start and run.run_end:
            duration = (run.run_end - run.run_start).total_seconds()

        recent_runs.append(PipelineRunInfo(
            run_id=run.run_id,
            pipeline_name=run.pipeline_name,
            status=PipelineStatus(run.status),
            start_time=run.run_start,
            end_time=run.run_end,
            duration_seconds=duration,
            error_message=run.message if run.status == "Failed" else None
        ))

    return GetPipelineStatusOutput(
        pipeline_name=input_data.pipeline_name,
        last_run_status=PipelineStatus(last_run.status),
        last_run_start=last_run.run_start,
        last_run_end=last_run.run_end,
```

```
            last_success_time=last_success.run_start if last_success else None,
            last_failure_reason=last_failure.message if last_failure else None,
            recent_runs=recent_runs
        )

    except Exception as e:
        logger.error(f"Error fetching pipeline status: {str(e)}")
        raise

def get_pipeline_dependencies(self, input_data: GetPipelineDependenciesInput
    """
    Analyze pipeline dependencies by parsing pipeline JSON definition.

    Implementation:
    1. Fetch pipeline definition from ADF
    2. Parse activities to find Execute Pipeline activities (upstream)
    3. Parse datasets consumed and produced
    4. Extract linked services used
    5. Query other pipelines to find downstream dependencies
    """
    try:
        logger.info(f"Analyzing dependencies for pipeline: {input_data.pipeline_na

        # Get pipeline definition
        pipeline = self.df_client.pipelines.get(
            resource_group_name=self.resource_group,
            factory_name=self.factory_name,
            pipeline_name=input_data.pipeline_name
        )

        upstream_pipelines = []
        datasets_consumed = []
        datasets_produced = []
        linked_services = set()

        # Parse activities
        if hasattr(pipeline, 'activities') and pipeline.activities:
            for activity in pipeline.activities:
```

```
        # Check for Execute Pipeline activities (upstream dependencies)
        if activity.type == "ExecutePipeline":
            if hasattr(activity, 'type_properties'):
                pipeline_ref = activity.type_properties.get('pipeline', {})
                if 'referenceName' in pipeline_ref:
                    upstream_pipelines.append(pipeline_ref['referenceName'])

        # Check for Copy activities (datasets)
        elif activity.type == "Copy":
            if hasattr(activity, 'type_properties'):
                # Source dataset
                if 'source' in activity.type_properties:
                    source = activity.type_properties['source']
                    if 'dataset' in source:
                        datasets_consumed.append(source['dataset'].get('referenceNa

                # Sink dataset
                if 'sink' in activity.type_properties:
                    sink = activity.type_properties['sink']
                    if 'dataset' in sink:
                        datasets_produced.append(sink['dataset'].get('referenceName'

        # Extract linked services from activity
        if hasattr(activity, 'linked_service_name'):
            if activity.linked_service_name:
                linked_services.add(activity.linked_service_name.reference_name

# Find downstream pipelines (pipelines that execute this one)
downstream_pipelines = []
all_pipelines = list(self.df_client.pipelines.list_by_factory(
    resource_group_name=self.resource_group,
    factory_name=self.factory_name
))

for other_pipeline in all_pipelines:
    if other_pipeline.name == input_data.pipeline_name:
        continue
```

```
            if hasattr(other_pipeline, 'activities') and other_pipeline.activities:
                for activity in other_pipeline.activities:
                    if activity.type == "ExecutePipeline":
                        if hasattr(activity, 'type_properties'):
                            pipeline_ref = activity.type_properties.get('pipeline', {})
                            if pipeline_ref.get('referenceName') == input_data.pipeline_nam
                                downstream_pipelines.append(other_pipeline.name)
                                break

        return GetPipelineDependenciesOutput(
            pipeline_name=input_data.pipeline_name,
            upstream_pipelines=upstream_pipelines,
            downstream_pipelines=downstream_pipelines,
            datasets_consumed=list(set(datasets_consumed)),
            datasets_produced=list(set(datasets_produced)),
            linked_services=list(linked_services)
        )

    except Exception as e:
        logger.error(f"Error analyzing pipeline dependencies: {str(e)}")
        raise

def get_failed_tasks_summary(self, input_data: GetFailedTasksSummaryInput) -
    """
    Summarize failed activities across pipeline runs within a time window.

    Implementation:
    1. Query pipeline runs within time window
    2. For each failed run, fetch activity runs
    3. Aggregate failures by activity name and error code
    4. Count occurrences and track timestamps
    """
    try:
        logger.info(f"Analyzing failed tasks for pipeline: {input_data.pipeline_nam

        # Calculate time range
        end_time = datetime.utcnow()
        start_time = end_time - timedelta(hours=input_data.time_window_hours)
```

```
# Query pipeline runs
filter_params = {
    'last_updated_after': start_time,
    'last_updated_before': end_time
}

runs = list(self.df_client.pipeline_runs.query_by_factory(
    resource_group_name=self.resource_group,
    factory_name=self.factory_name,
    filter_parameters=filter_params
).value)

# Filter for specific pipeline and failed status
failed_runs = [
    run for run in runs
    if run.pipeline_name == input_data.pipeline_name and run.status == "Fa
]

# Aggregate failed activities
failure_map = {}  # Key: (activity_name, error_code)

for run in failed_runs:
    # Get activity runs for this pipeline run
    activity_runs = list(self.df_client.activity_runs.query_by_pipeline_run(
        resource_group_name=self.resource_group,
        factory_name=self.factory_name,
        run_id=run.run_id,
        filter_parameters={}
    ).value)

    for activity in activity_runs:
        if activity.status == "Failed":
            error_code = activity.error.get('errorCode', 'UNKNOWN') if activity.er
            error_message = activity.error.get('message', 'No error message') if ac

            key = (activity.activity_name, error_code)
```

```
            if key not in failure_map:
                failure_map[key] = {
                    'activity_name': activity.activity_name,
                    'error_code': error_code,
                    'error_message': error_message,
                    'count': 0,
                    'first_failure': activity.activity_run_end or datetime.utcnow(),
                    'last_failure': activity.activity_run_end or datetime.utcnow()
                }

            failure_map[key]['count'] += 1

            # Update timestamps
            if activity.activity_run_end:
                if activity.activity_run_end < failure_map[key]['first_failure']:
                    failure_map[key]['first_failure'] = activity.activity_run_end
                if activity.activity_run_end > failure_map[key]['last_failure']:
                    failure_map[key]['last_failure'] = activity.activity_run_end

    # Convert to FailedTask objects
    failed_tasks = [
        FailedTask(
            activity_name=data['activity_name'],
            error_code=data['error_code'],
            error_message=data['error_message'],
            failure_count=data['count'],
            first_failure=data['first_failure'],
            last_failure=data['last_failure']
        )
        for data in failure_map.values()
    ]

    # Sort by failure count descending
    failed_tasks.sort(key=lambda x: x.failure_count, reverse=True)

    return GetFailedTasksSummaryOutput(
        pipeline_name=input_data.pipeline_name,
        time_window_hours=input_data.time_window_hours,
```

```
            total_failures=len(failed_runs),
            failed_tasks=failed_tasks
        )

    except Exception as e:
        logger.error(f"Error analyzing failed tasks: {str(e)}")
        raise

def get_all_pipelines(self) -> List[Dict[str, Any]]:
    """
    List all pipelines in the Data Factory with metadata.

    Used by: getadfpipelines MCP tool
    """
    try:
        pipelines = list(self.df_client.pipelines.list_by_factory(
            resource_group_name=self.resource_group,
            factory_name=self.factory_name
        ))

        result = []
        for pipeline in pipelines:
            # Extract basic info
            pipeline_info = {
                'name': pipeline.name,
                'description': getattr(pipeline, 'description', None),
                'uses_key_vault': False,
                'linked_services': [],
                'environment': 'dev'  # Can be enhanced with tags
            }

            # Check if pipeline uses Key Vault
            if hasattr(pipeline, 'activities') and pipeline.activities:
                for activity in pipeline.activities:
                    if hasattr(activity, 'linked_service_name') and activity.linked_service
                        ls_name = activity.linked_service_name.reference_name
                        pipeline_info['linked_services'].append(ls_name)
```

```
                    # Check if linked service uses Key Vault
                    try:
                        ls = self.df_client.linked_services.get(
                            resource_group_name=self.resource_group,
                            factory_name=self.factory_name,
                            linked_service_name=ls_name
                        )
                        # Check if connection string references Key Vault
                        if hasattr(ls, 'type_properties'):
                            props_str = str(ls.type_properties)
                            if 'AzureKeyVault' in props_str or 'vault' in props_str.lower():
                                pipeline_info['uses_key_vault'] = True
                    except:
                        pass

            result.append(pipeline_info)

        return result

    except Exception as e:
        logger.error(f"Error listing pipelines: {str(e)}")
        raise
```

4.4 Key Vault Tools Implementation

File: `mcp_server/tools/keyvault_tools.py`

```
from typing import List, Dict, Any
from datetime import datetime
from mcp_server.utils.azure_client import azure_clients
from mcp_server.models.tool_schemas import (
GetKeyVaultSecretsInput, GetKeyVaultSecretsOutput, SecretInfo,
GetSecretUsageInput, GetSecretUsageOutput, SecretUsage
)
from mcp_server.config import get_settings
from mcp_server.tools.adf_tools import ADFTools
import logging

logger = logging.getLogger(__name__)
settings = get_settings()

class KeyVaultTools:
"""Azure Key Vault tools implementation"""
```

```
def __init__(self):
    self.kv_client = azure_clients.get_keyvault_client()
    self.adf_tools = ADFTools()

def get_keyvault_secrets(self, input_data: GetKeyVaultSecretsInput) -> GetKeyVa
    """
    List secrets from Key Vault with metadata.

    Implementation:
    1. List all secrets from Key Vault
    2. Filter by prefix if specified
    3. Get secret properties (enabled, dates, tags)
    4. Determine risk level from tags or naming convention
    """
    try:
        logger.info("Fetching Key Vault secrets")

        secrets_list = []

        # List all secret properties
        secret_properties = self.kv_client.list_properties_of_secrets()

        for secret_prop in secret_properties:
            # Apply prefix filter
            if input_data.prefix and not secret_prop.name.startswith(input_data.pref
                continue

            # Determine risk level
            risk_level = None
            if secret_prop.tags:
                risk_level = secret_prop.tags.get('risk', None)

            # Check naming convention for risk
            if not risk_level:
                if 'prod' in secret_prop.name.lower():
                    risk_level = 'high'
                elif 'high-risk' in secret_prop.name.lower():
```

```
                risk_level = 'high'
            else:
                risk_level = 'medium'

        # Filter by risk if needed
        if not input_data.include_high_risk and risk_level == 'high':
            continue

        secrets_list.append(SecretInfo(
            name=secret_prop.name,
            enabled=secret_prop.enabled,
            created_on=secret_prop.created_on,
            updated_on=secret_prop.updated_on,
            tags=secret_prop.tags or {},
            risk_level=risk_level
        ))

    return GetKeyVaultSecretsOutput(
        secrets=secrets_list,
        total_count=len(secrets_list)
    )

except Exception as e:
    logger.error(f"Error fetching Key Vault secrets: {str(e)}")
    raise

def get_secret_usage(self, input_data: GetSecretUsageInput) -> GetSecretUsageO
    """
    Find which pipelines and linked services use a specific secret.

    Implementation:
    1. Get all pipelines from ADF
    2. For each pipeline, get linked services
    3. For each linked service, check if it references the secret
    4. Determine if pipeline is production-critical
    """
    try:
        logger.info(f"Analyzing usage for secret: {input_data.secret_name}")
```

```
df_client = azure_clients.get_datafactory_client()
resource_group = settings.azure_resource_group
factory_name = settings.azure_data_factory_name

usages = []

# Get all linked services
linked_services = list(df_client.linked_services.list_by_factory(
    resource_group_name=resource_group,
    factory_name=factory_name
))

# Map linked services that use this secret
ls_using_secret = {}

for ls in linked_services:
    uses_secret = False

    # Check type properties for Key Vault reference
    if hasattr(ls, 'type_properties'):
        props_str = str(ls.type_properties)
        # Check if this secret name appears in properties
        if input_data.secret_name in props_str:
            uses_secret = True

    if uses_secret:
        ls_using_secret[ls.name] = ls

if not ls_using_secret:
    return GetSecretUsageOutput(
        secret_name=input_data.secret_name,
        usage_count=0,
        usages=[]
    )

# Get all pipelines
pipelines = self.adf_tools.get_all_pipelines()
```

```
        # Find pipelines using these linked services
        for pipeline in pipelines:
            for ls_name in pipeline.get('linked_services', []):
                if ls_name in ls_using_secret:
                    # Determine if production critical
                    is_prod_critical = (
                        'prod' in pipeline['name'].lower() or
                        pipeline.get('environment') == 'prod'
                    )

                    usages.append(SecretUsage(
                        pipeline_name=pipeline['name'],
                        linked_service_name=ls_name,
                        environment=pipeline.get('environment', 'dev'),
                        is_production_critical=is_prod_critical
                    ))

        return GetSecretUsageOutput(
            secret_name=input_data.secret_name,
            usage_count=len(usages),
            usages=usages
        )

    except Exception as e:
        logger.error(f"Error analyzing secret usage: {str(e)}")
        raise
```

4.5 Log Tools Implementation

File: `mcp_server/tools/log_tools.py`

```
from typing import List, Optional, Dict, Any
from datetime import datetime, timedelta
from collections import defaultdict
import re
import json
from pathlib import Path
from mcp_server.utils.azure_client import azure_clients
from mcp_server.models.tool_schemas import (
FetchLogsInput, FetchLogsOutput, LogEntry, LogSource, LogLevel,
```

SummarizeErrorLogsInput, SummarizeErrorLogsOutput, ErrorCluster
)
from mcp_server.config import get_settings
import logging

logger = logging.getLogger(**name**)
settings = get_settings()

class LogTools:
"""Log intelligence tools implementation"""

```
def __init__(self):
    self.df_client = azure_clients.get_datafactory_client()
    self.resource_group = settings.azure_resource_group
    self.factory_name = settings.azure_data_factory_name
    self.app_log_path = settings.log_file_path

def fetch_logs(self, input_data: FetchLogsInput) -> FetchLogsOutput:
    """
    Fetch logs from specified source (ADF or application logs).

    Implementation:
    - For ADF: Query pipeline runs and activity runs via Azure SDK
    - For App: Read and parse local log files
    """
    try:
        logger.info(f"Fetching logs from source: {input_data.source}")

        if input_data.source == LogSource.ADF:
            logs = self._fetch_adf_logs(input_data)
        elif input_data.source == LogSource.APP:
            logs = self._fetch_app_logs(input_data)
        else:
            raise ValueError(f"Unsupported log source: {input_data.source}")

        return FetchLogsOutput(
            logs=logs,
            total_count=len(logs)
        )
```

```
    except Exception as e:
        logger.error(f"Error fetching logs: {str(e)}")
        raise

def _fetch_adf_logs(self, input_data: FetchLogsInput) -> List[LogEntry]:
    """Fetch logs from Azure Data Factory"""
    logs = []

    # Set time range
    end_time = input_data.time_end or datetime.utcnow()
    start_time = input_data.time_start or (end_time - timedelta(hours=24))

    # Query pipeline runs
    filter_params = {
        'last_updated_after': start_time,
        'last_updated_before': end_time
    }

    runs = list(self.df_client.pipeline_runs.query_by_factory(
        resource_group_name=self.resource_group,
        factory_name=self.factory_name,
        filter_parameters=filter_params
    ).value)

    # Filter by pipeline name if specified
    if input_data.pipeline_name:
        runs = [r for r in runs if r.pipeline_name == input_data.pipeline_name]

    # Filter by run_id if specified
    if input_data.run_id:
        runs = [r for r in runs if r.run_id == input_data.run_id]

    for run in runs:
        # Add pipeline run log entry
        level = LogLevel.ERROR if run.status == "Failed" else LogLevel.INFO

        # Filter by level if specified
        if input_data.level and level != input_data.level:
```

```
            continue

        logs.append(LogEntry(
            timestamp=run.run_start,
            level=level,
            source=LogSource.ADF,
            message=f"Pipeline run {run.status}: {run.message or 'No message'}",
            pipeline_name=run.pipeline_name,
            run_id=run.run_id,
            activity_name=None,
            error_code=None,
            metadata={
                'status': run.status,
                'duration_ms': run.duration_in_ms if hasattr(run, 'duration_in_ms') el
            }
        ))

        # Get activity runs for failed pipeline runs
        if run.status == "Failed":
            try:
                activity_runs = list(self.df_client.activity_runs.query_by_pipeline_run(
                    resource_group_name=self.resource_group,
                    factory_name=self.factory_name,
                    run_id=run.run_id,
                    filter_parameters={}
                ).value)

                for activity in activity_runs:
                    if activity.status == "Failed":
                        error_code = activity.error.get('errorCode', 'UNKNOWN') if activity.
                        error_message = activity.error.get('message', 'No error message') if

                        logs.append(LogEntry(
                            timestamp=activity.activity_run_end or run.run_start,
                            level=LogLevel.ERROR,
                            source=LogSource.ADF,
                            message=f"Activity {activity.activity_name} failed: {error_messa
                            pipeline_name=run.pipeline_name,
```

```
                        run_id=run.run_id,
                        activity_name=activity.activity_name,
                        error_code=error_code,
                        metadata={
                            'activity_type': activity.activity_type,
                            'error': activity.error
                        }
                    ))
            except Exception as e:
                logger.warning(f"Could not fetch activity runs for {run.run_id}: {str(e)

    return logs

def _fetch_app_logs(self, input_data: FetchLogsInput) -> List[LogEntry]:
    """Fetch logs from application log files"""
    logs = []

    if not self.app_log_path.exists():
        logger.warning(f"App log file not found: {self.app_log_path}")
        return logs

    # Set time range
    end_time = input_data.time_end or datetime.utcnow()
    start_time = input_data.time_start or (end_time - timedelta(hours=24))

    # Read log file
    with open(self.app_log_path, 'r') as f:
        for line in f:
            try:
                # Try JSON format first
                log_data = json.loads(line.strip())

                # Parse timestamp
                timestamp = datetime.fromisoformat(log_data.get('timestamp', ''))

                # Filter by time range
                if timestamp < start_time or timestamp > end_time:
                    continue
```

```
            # Parse level
            level_str = log_data.get('level', 'INFO').upper()
            level = LogLevel[level_str] if level_str in LogLevel.__members__ else L

            # Filter by level if specified
            if input_data.level and level != input_data.level:
                continue

            # Filter by pipeline name if specified
            if input_data.pipeline_name and log_data.get('pipeline_name') != input
                continue

            logs.append(LogEntry(
                timestamp=timestamp,
                level=level,
                source=LogSource.APP,
                message=log_data.get('message', ''),
                pipeline_name=log_data.get('pipeline_name'),
                run_id=log_data.get('run_id'),
                activity_name=None,
                error_code=None,
                metadata=log_data.get('metadata', {})
            ))

        except (json.JSONDecodeError, ValueError, KeyError):
            # Fallback: parse as plain text with regex
            # Format: YYYY-MM-DD HH:MM:SS LEVEL message
            match = re.match(
                r'(\d{4}-\d{2}-\d{2}\s+\d{2}:\d{2}:\d{2})\s+(\w+)\s+(.*)',
                line.strip()
            )
            if match:
                timestamp_str, level_str, message = match.groups()
                timestamp = datetime.strptime(timestamp_str, '%Y-%m-%d %H:%M:%

                # Filter by time range
                if timestamp < start_time or timestamp > end_time:
```

```
                continue

            level = LogLevel[level_str.upper()] if level_str.upper() in LogLevel.__

            # Filter by level if specified
            if input_data.level and level != input_data.level:
                continue

            logs.append(LogEntry(
                timestamp=timestamp,
                level=level,
                source=LogSource.APP,
                message=message,
                pipeline_name=None,
                run_id=None,
                activity_name=None,
                error_code=None,
                metadata={}
            ))

    return logs

def summarize_error_logs(self, input_data: SummarizeErrorLogsInput) -> Sum
    """
    Cluster and summarize error logs to identify patterns and anomalies.

    Implementation:
    1. Fetch logs if not provided
    2. Filter for errors only
    3. Cluster similar error messages
    4. Count occurrences and track timestamps
    5. Identify anomalies (new errors, spike in frequency)
    """
    try:
        logger.info("Summarizing error logs")

        # Get logs if not provided
        if input_data.logs is None:
```

```
            fetch_input = FetchLogsInput(
                source=input_data.source or LogSource.ADF,
                pipeline_name=input_data.pipeline_name,
                time_start=datetime.utcnow() - timedelta(hours=input_data.time_wind
                time_end=datetime.utcnow(),
                level=LogLevel.ERROR
            )
            fetch_result = self.fetch_logs(fetch_input)
            logs = fetch_result.logs
        else:
            # Filter for errors
            logs = [log for log in input_data.logs if log.level == LogLevel.ERROR]

        # Cluster errors
        clusters = self._cluster_errors(logs)

        # Identify anomalies
        anomalies = self._identify_anomalies(clusters, input_data.time_window_ho

        return SummarizeErrorLogsOutput(
            total_errors=len(logs),
            clusters=clusters,
            anomalies=anomalies
        )

    except Exception as e:
        logger.error(f"Error summarizing logs: {str(e)}")
        raise

def _cluster_errors(self, logs: List[LogEntry]) -> List[ErrorCluster]:
    """Cluster similar error messages"""
    # Group by error_code first, then by normalized message
    error_groups = defaultdict(list)

    for log in logs:
        # Create clustering key
        key_parts = []
```

```
        if log.error_code:
            key_parts.append(log.error_code)

        # Normalize message (remove timestamps, numbers, IDs)
        normalized_msg = re.sub(r'\d{4}-\d{2}-\d{2}', '<date>', log.message)
        normalized_msg = re.sub(r'\d{2}:\d{2}:\d{2}', '<time>', normalized_msg)
        normalized_msg = re.sub(r'[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a
        normalized_msg = re.sub(r'\d+', '<num>', normalized_msg)

        # Extract first sentence as pattern
        pattern = normalized_msg.split('.')[0][:100]
        key_parts.append(pattern)

        key = '|'.join(key_parts)
        error_groups[key].append(log)

    # Convert to ErrorCluster objects
    clusters = []
    for cluster_id, (key, logs_in_cluster) in enumerate(error_groups.items()):
        # Get pattern and sample message
        parts = key.split('|')
        error_pattern = parts[-1]

        # Get affected pipelines
        affected_pipelines = list(set(
            log.pipeline_name for log in logs_in_cluster if log.pipeline_name
        ))

        # Get time range
        timestamps = [log.timestamp for log in logs_in_cluster]

        clusters.append(ErrorCluster(
            cluster_id=f"error_cluster_{cluster_id}",
            error_pattern=error_pattern,
            sample_message=logs_in_cluster[0].message,
            count=len(logs_in_cluster),
            first_occurrence=min(timestamps),
            last_occurrence=max(timestamps),
```

```
            affected_pipelines=affected_pipelines
        ))

    # Sort by count descending
    clusters.sort(key=lambda x: x.count, reverse=True)

    return clusters

def _identify_anomalies(self, clusters: List[ErrorCluster], time_window_hours:
    """Identify anomalous error patterns"""
    anomalies = []

    # Check for new error patterns (first seen recently)
    recent_threshold = datetime.utcnow() - timedelta(hours=time_window_hours

    for cluster in clusters:
        # New error pattern
        if cluster.first_occurrence > recent_threshold:
            anomalies.append(
                f"New error pattern detected: '{cluster.error_pattern}' "
                f"first seen at {cluster.first_occurrence.isoformat()}, "
                f"occurred {cluster.count} times"
            )

        # High frequency errors
        if cluster.count > 10:
            time_span = (cluster.last_occurrence - cluster.first_occurrence).total_seco
            if time_span > 0:
                rate = cluster.count / time_span
                if rate > 2:  # More than 2 per hour
                    anomalies.append(
                        f"High frequency error: '{cluster.error_pattern}' "
                        f"occurring at {rate:.1f} times per hour"
                    )

    return anomalies

def compare_success_vs_failure_logs(
```

```
    self,
    pipeline_name: str,
    success_run_id: str,
    failure_run_id: str
) -> Dict[str, Any]:
    """
    Compare logs between successful and failed pipeline runs.

    Implementation:
    1. Fetch logs for both runs
    2. Compare activity sequences
    3. Identify differences in execution paths
    4. Highlight errors present only in failure
    """
    try:
        logger.info(f"Comparing runs for pipeline: {pipeline_name}")

        # Fetch logs for success run
        success_logs = self.fetch_logs(FetchLogsInput(
            source=LogSource.ADF,
            pipeline_name=pipeline_name,
            run_id=success_run_id
        )).logs

        # Fetch logs for failure run
        failure_logs = self.fetch_logs(FetchLogsInput(
            source=LogSource.ADF,
            pipeline_name=pipeline_name,
            run_id=failure_run_id
        )).logs

        # Extract activity sequences
        success_activities = [
            log.activity_name for log in success_logs
            if log.activity_name
        ]
        failure_activities = [
            log.activity_name for log in failure_logs
```

```
            if log.activity_name
        ]

        # Find differences
        activities_only_in_failure = set(failure_activities) - set(success_activities)
        activities_only_in_success = set(success_activities) - set(failure_activities)

        # Get error messages from failure
        error_messages = [
            log.message for log in failure_logs
            if log.level == LogLevel.ERROR
        ]

        return {
            'pipeline_name': pipeline_name,
            'success_run_id': success_run_id,
            'failure_run_id': failure_run_id,
            'differences': {
                'activities_only_in_failure': list(activities_only_in_failure),
                'activities_only_in_success': list(activities_only_in_success),
                'error_messages': error_messages
            },
            'summary': self._generate_comparison_summary(
                success_activities,
                failure_activities,
                error_messages
            )
        }

    except Exception as e:
        logger.error(f"Error comparing logs: {str(e)}")
        raise

def _generate_comparison_summary(
    self,
    success_activities: List[str],
    failure_activities: List[str],
    error_messages: List[str]
```

```
) -> str:
    """Generate human-readable comparison summary"""
    summary_parts = []

    if not failure_activities:
        summary_parts.append("Failure run has no activity logs.")
    elif not success_activities:
        summary_parts.append("Success run has no activity logs for comparison.")
    else:
        common_count = len(set(success_activities) & set(failure_activities))
        summary_parts.append(
            f"Both runs executed {common_count} common activities."
        )

    if error_messages:
        summary_parts.append(
            f"Failure run encountered {len(error_messages)} errors. "
            f"Primary error: {error_messages[0][:200]}"
        )

    return " ".join(summary_parts)
```

4.6 Terraform Tools Implementation

File: `mcp_server/tools/terraform_tools.py`

```
import json
from typing import List, Dict, Any, Optional
from pathlib import Path
from mcp_server.models.tool_schemas import (
ParseTerraformPlanInput, ParseTerraformPlanOutput, ResourceChange,
DetectInfraDriftInput, DetectInfraDriftOutput, DriftItem,
TerraformAction
)
from mcp_server.utils.azure_client import azure_clients
from mcp_server.config import get_settings
import logging

logger = logging.getLogger(__name__)
settings = get_settings()

class TerraformTools:
"""Terraform infrastructure reasoning tools"""
```

```
def __init__(self):
    self.resource_client = azure_clients.get_resource_client()
    self.plans_dir = settings.terraform_plans_dir

def parse_terraform_plan(self, input_data: ParseTerraformPlanInput) -> ParseT
    """
    Parse Terraform plan JSON and categorize resource changes.

    Implementation:
    1. Load plan JSON file
    2. Extract resource_changes array
    3. Categorize by action type (create, update, delete)
    4. Identify high-risk changes (deletions, destructive updates)
    5. Extract before/after values for updates
    """
    try:
        logger.info(f"Parsing Terraform plan: {input_data.plan_path}")

        # Load plan file
        plan_path = Path(input_data.plan_path)
        if not plan_path.is_absolute():
            plan_path = self.plans_dir / plan_path

        if not plan_path.exists():
            raise FileNotFoundError(f"Plan file not found: {plan_path}")

        with open(plan_path, 'r') as f:
            plan_data = json.load(f)

        # Extract resource changes
        resource_changes = plan_data.get('resource_changes', [])

        created = []
        updated = []
        deleted = []
        high_risk = []
```

```
        for change in resource_changes:
            actions = change.get('change', {}).get('actions', [])
            resource_type = change.get('type', '')
            resource_name = change.get('name', '')
            address = change.get('address', '')
            before = change.get('change', {}).get('before', None)
            after = change.get('change', {}).get('after', None)

            # Determine action type
            action_list = [TerraformAction(a) for a in actions if a in TerraformAction

            is_destructive = TerraformAction.DELETE in action_list

            resource_change = ResourceChange(
                resource_type=resource_type,
                resource_name=resource_name,
                address=address,
                actions=action_list,
                before=before,
                after=after,
                is_destructive=is_destructive
            )

            # Categorize
            if TerraformAction.CREATE in action_list and TerraformAction.DELETE r
                created.append(resource_change)
            elif TerraformAction.UPDATE in action_list:
                updated.append(resource_change)
                # Check if update is destructive (e.g., changing immutable properties)
                if self._is_destructive_update(resource_type, before, after):
                    high_risk.append(resource_change)
            elif TerraformAction.DELETE in action_list:
                deleted.append(resource_change)
                high_risk.append(resource_change)

        return ParseTerraformPlanOutput(
            plan_path=str(plan_path),
            created_resources=created,
```

```
            updated_resources=updated,
            deleted_resources=deleted,
            high_risk_changes=high_risk
        )

    except Exception as e:
        logger.error(f"Error parsing Terraform plan: {str(e)}")
        raise

def _is_destructive_update(
    self,
    resource_type: str,
    before: Optional[Dict],
    after: Optional[Dict]
) -> bool:
    """Check if an update is destructive (requires replacement)"""
    if not before or not after:
        return False

    # Define properties that force replacement for common resource types
    immutable_properties = {
        'azurerm_virtual_machine': ['location', 'vm_size'],
        'azurerm_storage_account': ['location', 'account_tier'],
        'azurerm_virtual_network': ['location', 'address_space'],
        'azurerm_linux_virtual_machine': ['location', 'size'],
    }

    if resource_type in immutable_properties:
        for prop in immutable_properties[resource_type]:
            if before.get(prop) != after.get(prop):
                return True

    return False

def detect_infra_drift(self, input_data: DetectInfraDriftInput) -> DetectInfraDrif
    """
    Detect drift between Terraform plan and actual Azure resources.
```

```
    Implementation:
    1. Parse Terraform plan to get expected resources
    2. Query Azure to get actual resources in resource group
    3. Compare and identify:
       - Resources in Azure but not in plan (extra)
       - Resources in plan but not in Azure (missing)
       - Configuration drift (optional, simplified)
    """
    try:
        logger.info(f"Detecting infrastructure drift for RG: {input_data.resource_gr

        drift_items = []

        # Get expected resources from plan
        expected_resources = {}
        if input_data.plan_path:
            plan_result = self.parse_terraform_plan(
                ParseTerraformPlanInput(plan_path=input_data.plan_path)
            )

            # Build expected resource map
            for resource in (plan_result.created_resources + plan_result.updated_res
                expected_resources[resource.address] = resource

        # Get actual resources from Azure
        actual_resources = list(self.resource_client.resources.list_by_resource_grou
            resource_group_name=input_data.resource_group_name
        ))

        # Build map of actual resources
        actual_resource_map = {
            f"{r.type}/{r.name}": r
            for r in actual_resources
        }

        # Find resources in Azure but not in plan
        if expected_resources:
            for resource_key in actual_resource_map.keys():
```

```
        # Simplified matching (in production, use more robust matching)
        found_in_plan = False
        for expected_addr in expected_resources.keys():
            if resource_key in expected_addr or expected_addr in resource_key:
                found_in_plan = True
                break

        if not found_in_plan:
            drift_items.append(DriftItem(
                resource_type=actual_resource_map[resource_key].type,
                resource_name=actual_resource_map[resource_key].name,
                drift_type="extra_in_cloud",
                details=f"Resource exists in Azure but not defined in Terraform p
            ))

    # Find resources in plan but not in Azure
    if expected_resources:
        for expected_addr, expected_resource in expected_resources.items():
            # Check if exists in actual
            found_in_azure = False
            for actual_key in actual_resource_map.keys():
                if expected_resource.resource_name in actual_key:
                    found_in_azure = True
                    break

            if not found_in_azure and TerraformAction.CREATE not in expected_r
                drift_items.append(DriftItem(
                    resource_type=expected_resource.resource_type,
                    resource_name=expected_resource.resource_name,
                    drift_type="missing_in_cloud",
                    details=f"Resource defined in Terraform but not found in Azure"
                ))

    return DetectInfraDriftOutput(
        has_drift=len(drift_items) > 0,
        drift_items=drift_items
    )
```

```
    except Exception as e:
        logger.error(f"Error detecting infrastructure drift: {str(e)}")
        raise

def explain_plan_diff(
    self,
    plan_path: str,
    check_drift: bool = True
) -> Dict[str, Any]:
    """
    High-level explanation of Terraform plan with risk analysis.

    This combines parse_terraform_plan and detect_infra_drift
    to provide LLM-friendly summary.
    """
    try:
        logger.info(f"Explaining Terraform plan: {plan_path}")

        # Parse plan
        plan_result = self.parse_terraform_plan(
            ParseTerraformPlanInput(plan_path=plan_path)
        )

        # Check drift if requested
        drift_result = None
        if check_drift:
            try:
                drift_result = self.detect_infra_drift(
                    DetectInfraDriftInput(
                        resource_group_name=settings.azure_resource_group,
                        plan_path=plan_path
                    )
                )
            except Exception as e:
                logger.warning(f"Could not check drift: {str(e)}")

        # Generate summary
        summary_parts = []
```

```
    if plan_result.created_resources:
        summary_parts.append(
            f"{len(plan_result.created_resources)} resources will be created"
        )

    if plan_result.updated_resources:
        summary_parts.append(
            f"{len(plan_result.updated_resources)} resources will be updated"
        )

    if plan_result.deleted_resources:
        summary_parts.append(
            f"{len(plan_result.deleted_resources)} resources will be DELETED"
        )

    summary = ". ".join(summary_parts) + "."

    # Build risk items
    risk_items = []
    for resource in plan_result.high_risk_changes:
        risk_reason = "Resource will be deleted" if TerraformAction.DELETE in r

        risk_items.append({
            'resource': f"{resource.resource_type}.{resource.resource_name}",
            'action': ', '.join([a.value for a in resource.actions]),
            'risk_reason': risk_reason
        })

    return {
        'plan_path': plan_path,
        'summary': summary,
        'resource_counts': {
            'created': len(plan_result.created_resources),
            'updated': len(plan_result.updated_resources),
            'deleted': len(plan_result.deleted_resources)
        },
        'risk_items': risk_items,
```

```
            'has_high_risk': len(risk_items) > 0,
            'drift': drift_result.dict() if drift_result else None
        }

    except Exception as e:
        logger.error(f"Error explaining plan diff: {str(e)}")
        raise
```

4.7 Cloud Resource Tools Implementation

File: `mcp_server/tools/cloud_tools.py`

```
from typing import List, Dict, Any
from datetime import datetime, timedelta
from mcp_server.utils.azure_client import azure_clients
from mcp_server.models.tool_schemas import (
ListResourcesByTagInput, ListResourcesByTagOutput, ResourceInfo
)
from mcp_server.config import get_settings
import logging

logger = logging.getLogger(__name__)
settings = get_settings()

class CloudTools:
"""Azure cloud resource context tools"""
```

```
    def __init__(self):
        self.resource_client = azure_clients.get_resource_client()

    def list_resources_by_tag(self, input_data: ListResourcesByTagInput) -> ListReso
        """
        List Azure resources filtered by tag.

        Implementation:
        1. Query all resources (optionally filtered by RG)
        2. Filter by matching tag key and value
        3. Return resource metadata
        """
        try:
            logger.info(f"Listing resources with tag {input_data.tag_key}={input_data.ta

            # Get resources
```

```
        if input_data.resource_group:
            resources = list(self.resource_client.resources.list_by_resource_group(
                resource_group_name=input_data.resource_group
            ))
        else:
            resources = list(self.resource_client.resources.list())

        # Filter by tag
        matching_resources = []
        for resource in resources:
            if resource.tags and input_data.tag_key in resource.tags:
                if resource.tags[input_data.tag_key] == input_data.tag_value:
                    matching_resources.append(ResourceInfo(
                        resource_id=resource.id,
                        resource_name=resource.name,
                        resource_type=resource.type,
                        location=resource.location,
                        tags=resource.tags or {}
                    ))

        return ListResourcesByTagOutput(
            resources=matching_resources,
            count=len(matching_resources)
        )

    except Exception as e:
        logger.error(f"Error listing resources by tag: {str(e)}")
        raise

def get_resource_health(self, resource_id: str) -> Dict[str, Any]:
    """
    Get health status of a specific Azure resource.

    Implementation:
    1. Parse resource ID
    2. Query resource details
    3. Return health metadata
    """
```

```
    try:
        logger.info(f"Getting health for resource: {resource_id}")

        # Get resource by ID
        resource = self.resource_client.resources.get_by_id(
            resource_id=resource_id,
            api_version="2021-04-01"  # Generic API version
        )

        # Basic health info
        health_info = {
            'resource_id': resource_id,
            'resource_name': resource.name,
            'resource_type': resource.type,
            'provisioning_state': getattr(resource.properties, 'provisioningState', 'Unk
            'location': resource.location,
            'status': 'healthy' if getattr(resource.properties, 'provisioningState', '') == '
        }

        return health_info

    except Exception as e:
        logger.error(f"Error getting resource health: {str(e)}")
        raise

def get_recent_resource_changes(
    self,
    resource_id: str,
    hours: int = 24
) -> List[Dict[str, Any]]:
    """
    Get recent changes to a resource (simplified version).

    In production, this would use Azure Activity Log or Resource Graph.
    For demo, we return basic change detection.
    """
    try:
        logger.info(f"Getting recent changes for: {resource_id}")
```

```
        # Get current resource state
        resource = self.resource_client.resources.get_by_id(
            resource_id=resource_id,
            api_version="2021-04-01"
        )

        # Simplified: check last modified time if available
        changes = []

        if hasattr(resource, 'properties'):
            changes.append({
                'timestamp': datetime.utcnow().isoformat(),
                'change_type': 'state_check',
                'details': f"Current provisioning state: {getattr(resource.properties, 'pr
            })

        # Note: In production, query Azure Activity Log for actual changes:
        # - Tag updates
        # - Property changes
        # - Access policy modifications
        # This requires azure-mgmt-monitor SDK

        return changes

    except Exception as e:
        logger.error(f"Error getting resource changes: {str(e)}")
        raise
```

4.8 MCP Server Main and Tool Registry

File: `mcp_server/tool_registry.py`

```
from typing import Dict, Any, Callable
from mcp_server.models.tool_schemas import *
from mcp_server.tools.adf_tools import ADFTools
from mcp_server.tools.keyvault_tools import KeyVaultTools
from mcp_server.tools.log_tools import LogTools
from mcp_server.tools.terraform_tools import TerraformTools
from mcp_server.tools.cloud_tools import CloudTools
import logging
```

logger = logging.getLogger(**name**)

class ToolRegistry:
"""Central registry for all MCP tools"""

```
def __init__(self):
    # Initialize tool implementations
    self.adf_tools = ADFTools()
    self.kv_tools = KeyVaultTools()
    self.log_tools = LogTools()
    self.tf_tools = TerraformTools()
    self.cloud_tools = CloudTools()

    # Register all tools
    self.tools: Dict[str, Dict[str, Any]] = {}
    self._register_all_tools()

def _register_all_tools(self):
    """Register all available tools with their schemas"""

    # ADF Pipeline Tools
    self.register_tool(
        name="get_pipeline_status",
        description="Get current status and recent run history of an ADF pipeline'
        input_schema=GetPipelineStatusInput.model_json_schema(),
        output_schema=GetPipelineStatusOutput.model_json_schema(),
        handler=self.adf_tools.get_pipeline_status
    )

    self.register_tool(
        name="get_pipeline_dependencies",
        description="Analyze pipeline dependencies including upstream/downstre
        input_schema=GetPipelineDependenciesInput.model_json_schema(),
        output_schema=GetPipelineDependenciesOutput.model_json_schema(),
        handler=self.adf_tools.get_pipeline_dependencies
    )

    self.register_tool(
        name="get_failed_tasks_summary",
```

```
    description="Summarize failed activities across pipeline runs within a tim
    input_schema=GetFailedTasksSummaryInput.model_json_schema(),
    output_schema=GetFailedTasksSummaryOutput.model_json_schema(),
    handler=self.adf_tools.get_failed_tasks_summary
)

# Key Vault Tools
self.register_tool(
    name="get_keyvault_secrets",
    description="List secrets from Key Vault with metadata and risk levels",
    input_schema=GetKeyVaultSecretsInput.model_json_schema(),
    output_schema=GetKeyVaultSecretsOutput.model_json_schema(),
    handler=self.kv_tools.get_keyvault_secrets
)

self.register_tool(
    name="get_secret_usage",
    description="Find which pipelines and linked services use a specific secret
    input_schema=GetSecretUsageInput.model_json_schema(),
    output_schema=GetSecretUsageOutput.model_json_schema(),
    handler=self.kv_tools.get_secret_usage
)

# Log Tools
self.register_tool(
    name="fetch_logs",
    description="Fetch logs from ADF or application sources with filtering",
    input_schema=FetchLogsInput.model_json_schema(),
    output_schema=FetchLogsOutput.model_json_schema(),
    handler=self.log_tools.fetch_logs
)

self.register_tool(
    name="summarize_error_logs",
    description="Cluster and summarize error logs to identify patterns and an
    input_schema=SummarizeErrorLogsInput.model_json_schema(),
    output_schema=SummarizeErrorLogsOutput.model_json_schema(),
    handler=self.log_tools.summarize_error_logs
```

```
    )

    # Terraform Tools
    self.register_tool(
        name="parse_terraform_plan",
        description="Parse Terraform plan JSON and categorize resource changes v
        input_schema=ParseTerraformPlanInput.model_json_schema(),
        output_schema=ParseTerraformPlanOutput.model_json_schema(),
        handler=self.tf_tools.parse_terraform_plan
    )

    self.register_tool(
        name="detect_infra_drift",
        description="Detect drift between Terraform plan and actual Azure resour
        input_schema=DetectInfraDriftInput.model_json_schema(),
        output_schema=DetectInfraDriftOutput.model_json_schema(),
        handler=self.tf_tools.detect_infra_drift
    )

    # Cloud Resource Tools
    self.register_tool(
        name="list_resources_by_tag",
        description="List Azure resources filtered by tag key and value",
        input_schema=ListResourcesByTagInput.model_json_schema(),
        output_schema=ListResourcesByTagOutput.model_json_schema(),
        handler=self.cloud_tools.list_resources_by_tag
    )

    logger.info(f"Registered {len(self.tools)} MCP tools")

def register_tool(
    self,
    name: str,
    description: str,
    input_schema: Dict,
    output_schema: Dict,
    handler: Callable
):
```

```
    """Register a single tool"""
    self.tools[name] = {
        'name': name,
        'description': description,
        'input_schema': input_schema,
        'output_schema': output_schema,
        'handler': handler
    }

def get_tool(self, name: str) -> Dict[str, Any]:
    """Get tool definition by name"""
    if name not in self.tools:
        raise ValueError(f"Tool not found: {name}")
    return self.tools[name]

def list_tools(self) -> List[Dict[str, Any]]:
    """List all registered tools"""
    return [
        {
            'name': tool['name'],
            'description': tool['description'],
            'input_schema': tool['input_schema'],
            'output_schema': tool['output_schema']
        }
        for tool in self.tools.values()
    ]

def execute_tool(self, name: str, input_data: Dict[str, Any]) -> Any:
    """Execute a tool by name with input data"""
    tool = self.get_tool(name)
    handler = tool['handler']

    # Parse input based on tool's input schema
    # In production, validate against schema
    logger.info(f"Executing tool: {name}")

    try:
        # Call handler (handlers expect Pydantic models)
```

```
            # Convert dict to appropriate Pydantic model
            input_model_class = self._get_input_model_class(name)
            if input_model_class:
                parsed_input = input_model_class(**input_data)
                result = handler(parsed_input)
            else:
                result = handler(**input_data)

            return result

        except Exception as e:
            logger.error(f"Error executing tool {name}: {str(e)}")
            raise

    def _get_input_model_class(self, tool_name: str):
        """Map tool name to input model class"""
        mapping = {
            'get_pipeline_status': GetPipelineStatusInput,
            'get_pipeline_dependencies': GetPipelineDependenciesInput,
            'get_failed_tasks_summary': GetFailedTasksSummaryInput,
            'get_keyvault_secrets': GetKeyVaultSecretsInput,
            'get_secret_usage': GetSecretUsageInput,
            'fetch_logs': FetchLogsInput,
            'summarize_error_logs': SummarizeErrorLogsInput,
            'parse_terraform_plan': ParseTerraformPlanInput,
            'detect_infra_drift': DetectInfraDriftInput,
            'list_resources_by_tag': ListResourcesByTagInput
        }
        return mapping.get(tool_name)
```

File: `mcp_server/main.py`

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from typing import Dict, Any, List
from mcp_server.tool_registry import ToolRegistry
from mcp_server.config import get_settings
import logging
import sys
```

Configure logging

logging.basicConfig(
level=logging.INFO,
format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
handlers=[
logging.FileHandler(get_settings().log_file_path),
logging.StreamHandler(sys.stdout)
]
)

logger = logging.getLogger(**name**)

app = FastAPI(title="MCP DataOps Server", version="1.0.0")

Initialize tool registry

tool_registry = ToolRegistry()

class ToolExecutionRequest(BaseModel):
tool_name: str
input_data: Dict[str, Any]

class ToolExecutionResponse(BaseModel):
tool_name: str
success: bool
result: Any = None
error: str = None

@app.get("/")
def root():
"""Health check endpoint"""
return {"status": "healthy", "service": "MCP DataOps Server"}

@app.get("/tools")
def list_tools():
"""List all available MCP tools"""
return {
"tools": tool_registry.list_tools(),
"count": len(tool_registry.tools)
}

@app.get("/tools/{tool_name}")
def get_tool(tool_name: str):
"""Get specific tool schema"""
try:
tool = tool_registry.get_tool(tool_name)
return {
"name": tool['name'],
"description": tool['description'],
"input_schema": tool['input_schema'],

"output_schema": tool['output_schema']
}
except ValueError as e:
raise HTTPException(status_code=404, detail=str(e))

@app.post("/execute", response_model=ToolExecutionResponse)
def execute_tool(request: ToolExecutionRequest):
"""Execute an MCP tool"""
try:
logger.info(f"Executing tool: {request.tool_name}")

```
    result = tool_registry.execute_tool(
        name=request.tool_name,
        input_data=request.input_data
    )

    # Convert Pydantic model to dict if needed
    if hasattr(result, 'dict'):
        result = result.dict()

    return ToolExecutionResponse(
        tool_name=request.tool_name,
        success=True,
        result=result
    )

except Exception as e:
    logger.error(f"Tool execution failed: {str(e)}", exc_info=True)
    return ToolExecutionResponse(
        tool_name=request.tool_name,
        success=False,
        error=str(e)
    )
```

if **name** == "**main**":
import uvicorn
uvicorn.run(app, host="0.0.0.0", port=8001)

---

5. Backend Implementation (FastAPI + LLM Client)

5.1 LLM Client Implementation

File: `backend/app/llm_client.py`

```
from openai import AzureOpenAI
from typing import List, Dict, Any, Optional
from backend.app.config import get_settings
import json
import logging

logger = logging.getLogger(__name__)
settings = get_settings()

class LLMClient:
"""Azure OpenAI client for chat completions with tool calling"""
```

```
def __init__(self):
    self.client = AzureOpenAI(
        api_key=settings.azure_openai_api_key,
        api_version=settings.azure_openai_api_version,
        azure_endpoint=settings.azure_openai_endpoint
    )
    self.deployment = settings.azure_openai_deployment_name

def chat(
    self,
    messages: List[Dict[str, str]],
    tools: Optional[List[Dict[str, Any]]] = None,
    tool_choice: str = "auto"
) -> Dict[str, Any]:
    """
    Send chat completion request with optional tool calling.

    Args:
        messages: List of message dicts with 'role' and 'content'
        tools: List of available tool definitions
        tool_choice: "auto", "none", or specific tool

    Returns:
        Response dict with message and optional tool_calls
```

```
    """
    try:
        logger.info(f"Sending chat request with {len(messages)} messages")

        request_params = {
            "model": self.deployment,
            "messages": messages,
            "temperature": 0.7,
            "max_tokens": 2000
        }

        if tools:
            request_params["tools"] = tools
            request_params["tool_choice"] = tool_choice

        response = self.client.chat.completions.create(**request_params)

        message = response.choices[0].message

        result = {
            "role": message.role,
            "content": message.content,
            "tool_calls": []
        }

        # Extract tool calls if present
        if hasattr(message, 'tool_calls') and message.tool_calls:
            for tool_call in message.tool_calls:
                result["tool_calls"].append({
                    "id": tool_call.id,
                    "type": tool_call.type,
                    "function": {
                        "name": tool_call.function.name,
                        "arguments": json.loads(tool_call.function.arguments)
                    }
                })

        return result
```

```
        except Exception as e:
            logger.error(f"LLM chat error: {str(e)}")
            raise

    def build_system_prompt(self, environment: str = "dev") -> str:
        """Build system prompt for DataOps assistant"""
        return f"""You are an expert DataOps assistant with deep knowledge of Azur
```

Your role:

- Analyze pipeline failures and identify root causes
- Explain dependencies and impacts
- Correlate failures across pipelines, secrets, and infrastructure
- Provide evidence-based explanations using tool results

Current environment: {environment}

When answering:

1. Use available tools to gather facts
2. Correlate information across systems (ADF, Key Vault, logs, infrastructure)
3. Provide specific, actionable insights
4. Always cite evidence from tool results
5. Explain in clear, technical language

Available data sources:

- Azure Data Factory pipelines and runs
- Azure Key Vault secrets and usage
- Pipeline and application logs
- Terraform infrastructure plans
- Azure cloud resources

Be thorough, precise, and always ground your explanations in actual data."""

5.2 MCP Client Implementation

File: backend/app/mcp_client.py

```
import httpx
from typing import Dict, Any, List
from backend.app.config import get_settings
import logging

logger = logging.getLogger(__name__)
settings = get_settings()

class MCPClient:
    """Client for communicating with MCP server"""
```

```
def __init__(self):
    self.base_url = settings.mcp_server_url
    self.client = httpx.Client(timeout=30.0)

def list_tools(self) -> List[Dict[str, Any]]:
    """Get list of available tools from MCP server"""
    try:
        response = self.client.get(f"{self.base_url}/tools")
        response.raise_for_status()
        data = response.json()
        return data.get("tools", [])
    except Exception as e:
        logger.error(f"Error listing MCP tools: {str(e)}")
        raise

def execute_tool(self, tool_name: str, input_data: Dict[str, Any]) -> Dict[str, Any]:
    """Execute a tool on MCP server"""
    try:
        logger.info(f"Executing MCP tool: {tool_name}")

        response = self.client.post(
            f"{self.base_url}/execute",
            json={
                "tool_name": tool_name,
                "input_data": input_data
            }
        )
        response.raise_for_status()

        result = response.json()

        if not result.get("success"):
            raise Exception(f"Tool execution failed: {result.get('error')}")

        return result.get("result")

    except Exception as e:
```

```
            logger.error(f"Error executing tool {tool_name}: {str(e)}")
            raise

    def get_tools_for_llm(self) -> List[Dict[str, Any]]:
        """Convert MCP tools to OpenAI function format"""
        tools = self.list_tools()

        llm_tools = []
        for tool in tools:
            llm_tools.append({
                "type": "function",
                "function": {
                    "name": tool["name"],
                    "description": tool["description"],
                    "parameters": tool["input_schema"]
                }
            })

        return llm_tools
```

5.3 Chat Router Implementation

File: `backend/app/routers/chat.py`

```
from fastapi import APIRouter, HTTPException
from pydantic import BaseModel
from typing import List, Dict, Any, Optional
from backend.app.llm_client import LLMClient
from backend.app.mcp_client import MCPClient
import logging

logger = logging.getLogger(__name__)

router = APIRouter(prefix="/chat", tags=["chat"])

# Initialize clients
llm_client = LLMClient()
mcp_client = MCPClient()

class ChatMessage(BaseModel):
    role: str
    content: str
```

```
class ChatRequest(BaseModel):
message: str
environment: str = "dev"
history: List[ChatMessage] = []

class ToolTrace(BaseModel):
tool_name: str
input_data: Dict[str, Any]
output_data: Any

class ChatResponse(BaseModel):
message: str
tool_traces: List[ToolTrace] = []

@router.post("", response_model=ChatResponse)
async def chat(request: ChatRequest):
"""
Main chat endpoint that orchestrates LLM and MCP server.

Flow:
1. Build system prompt with context
2. Send user message to LLM with available tools
3. If LLM requests tool calls, execute via MCP server
4. Feed tool results back to LLM
5. Repeat until final answer
6. Return answer with tool execution trace
"""
try:
    logger.info(f"Chat request: {request.message[:100]}...")

    # Get available tools
    tools = mcp_client.get_tools_for_llm()

    # Build conversation messages
    messages = [
        {
            "role": "system",
            "content": llm_client.build_system_prompt(request.environment)
        }
    ]

    # Add history
    for msg in request.history:
```

```
    messages.append({
        "role": msg.role,
        "content": msg.content
    })

# Add current user message
messages.append({
    "role": "user",
    "content": request.message
})

# Track tool executions
tool_traces = []

# Conversation loop (max 10 iterations)
max_iterations = 10
iteration = 0

while iteration < max_iterations:
    iteration += 1
    logger.info(f"Conversation iteration {iteration}")

    # Get LLM response
    llm_response = llm_client.chat(
        messages=messages,
        tools=tools,
        tool_choice="auto"
    )

    # Check if LLM wants to call tools
    if llm_response.get("tool_calls"):
        logger.info(f"LLM requested {len(llm_response['tool_calls'])} tool calls")

        # Add assistant message with tool calls
        messages.append({
            "role": "assistant",
            "content": llm_response.get("content") or "",
            "tool_calls": llm_response["tool_calls"]
```

```
        })

        # Execute each tool call
        for tool_call in llm_response["tool_calls"]:
            tool_name = tool_call["function"]["name"]
            tool_input = tool_call["function"]["arguments"]

            try:
                # Execute tool via MCP
                tool_result = mcp_client.execute_tool(tool_name, tool_input)

                # Track execution
                tool_traces.append(ToolTrace(
                    tool_name=tool_name,
                    input_data=tool_input,
                    output_data=tool_result
                ))

                # Add tool result to conversation
                messages.append({
                    "role": "tool",
                    "tool_call_id": tool_call["id"],
                    "name": tool_name,
                    "content": str(tool_result)
                })

            except Exception as e:
                logger.error(f"Tool execution error: {str(e)}")
                # Add error as tool result
                messages.append({
                    "role": "tool",
                    "tool_call_id": tool_call["id"],
                    "name": tool_name,
                    "content": f"Error executing tool: {str(e)}"
                })

    else:
        # No more tool calls, we have final answer
```

```
            final_message = llm_response.get("content") or "No response generated"

            return ChatResponse(
                message=final_message,
                tool_traces=tool_traces
            )

        # Max iterations reached
        return ChatResponse(
            message="I apologize, but I couldn't complete the analysis within the allow
            tool_traces=tool_traces
        )

    except Exception as e:
        logger.error(f"Chat error: {str(e)}", exc_info=True)
        raise HTTPException(status_code=500, detail=f"Chat processing error: {str(e)}
```

5.4 FastAPI Main Application

File: backend/app/main.py

```
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from backend.app.routers import chat
from backend.app.config import get_settings
import logging

# Configure logging
logging.basicConfig(
level=logging.INFO,
format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)

logger = logging.getLogger(__name__)
settings = get_settings()

app = FastAPI(
title="MCP DataOps Backend",
version="1.0.0",
description="Backend API for MCP-based DataOps Assistant"
)
```

CORS middleware

app.add_middleware(
CORSMiddleware,
allow_origins=["*"], # *Configure appropriately for production*
allow_credentials=True,
allow_methods=[""],*
allow_headers=["*"],
)

Include routers

app.include_router(chat.router)

@app.get("/")
def root():
return {
"service": "MCP DataOps Backend",
"version": "1.0.0",
"status": "healthy",
"environment": settings.environment
}

@app.get("/health")
def health():
return {"status": "healthy"}

if **name** == "**main**":
import uvicorn
uvicorn.run(app, host="0.0.0.0", port=8000)

---

6. Frontend Implementation

6.1 API Service

File: frontend/src/services/api.js

const API_BASE_URL = import.meta.env.VITE_API_BASE_URL || 'http://localhost:8000';

export const chatAPI = {
async sendMessage(message, environment = 'dev', history = []) {
const response = await fetch(${API_BASE_URL}/chat, {
method: 'POST',
headers: {
'Content-Type': 'application/json',
},
body: JSON.stringify({
message,
environment,
history,

}),
});

```
  if (!response.ok) {
    const error = await response.json();
    throw new Error(error.detail || 'Failed to send message');
  }

  return response.json();
```

},

async checkHealth() {
const response = await fetch(${API_BASE_URL}/health);
return response.json();
},
};

6.2 Main Components

File: frontend/src/components/ChatWindow.jsx

import React from 'react';

export default function ChatWindow({ messages }) {
return (

{messages.map((msg, idx) => (
<div
key={idx}
className={flex ${msg.role === 'user' ? 'justify-end' : 'justify-start'}}
>
<div
className={max-w-3xl rounded-lg px-4 py-2 ${ msg.role === 'user' ? 'bg-blue-500 text-white' : 'bg-gray-200 text-gray-900' }}
>

{msg.role === 'user' ? 'You' : 'Assistant'}
{msg.content}

))}

);
}

File: frontend/src/components/InputBar.jsx

import React, { useState } from 'react';

```
export default function InputBar({ onSend, disabled }) {
const [input, setInput] = useState('');

const handleSubmit = (e) => {
e.preventDefault();
if (input.trim() && !disabled) {
onSend(input);
setInput('');
}
};

return (

<input
type="text"
value={input}
onChange={(e) => setInput(e.target.value)}
placeholder="Ask about pipelines, secrets, logs, or infrastructure..."
disabled={disabled}
className="flex-1 rounded-lg border border-gray-300 px-4 py-2 focus:outline-none
focus:ring-2 focus:ring-blue-500"
/>
<button
type="submit"
disabled={disabled || !input.trim()}
className="rounded-lg bg-blue-500 px-6 py-2 text-white hover:bg-blue-600 disabled:bg-gray-
300 disabled:cursor-not-allowed"
>
Send
</button>

);
}
```

File: frontend/src/components/EnvironmentSelector.jsx

```
import React from 'react';

export default function EnvironmentSelector({ environment, onChange }) {
return (
Environment:
<select
value={environment}
onChange={(e) => onChange(e.target.value)}
className="rounded border border-gray-300 px-3 py-1 text-sm focus:outline-none
focus:ring-2 focus:ring-blue-500"
>
Development Production
</select>

);
}
```

File: frontend/src/components/ToolTracePanel.jsx

```
import React from 'react';

export default function ToolTracePanel({ traces, isOpen, onToggle }) {
if (!isOpen) {
return (

Show Tool Traces ({traces.length})

);
}

return (
Tool Execution Traces

×

{traces.map((trace, idx) => (
{trace.tool_name}
Input:
{JSON.stringify(trace.input_data, null, 2)}

Output:
{JSON.stringify(trace.output_data, null, 2)}


))}

);
}
```

File: frontend/src/App.jsx

```
import React, { useState } from 'react';
import ChatWindow from './components/ChatWindow';
import InputBar from './components/InputBar';
import EnvironmentSelector from './components/EnvironmentSelector';
import ToolTracePanel from './components/ToolTracePanel';
import { chatAPI } from './services/api';

function App() {
const [messages, setMessages] = useState([]);
const [environment, setEnvironment] = useState('dev');
const [isLoading, setIsLoading] = useState(false);
const [toolTraces, setToolTraces] = useState([]);
const [showTraces, setShowTraces] = useState(false);

const handleSend = async (userMessage) => {
// Add user message
const newMessages = [
```

```
...messages,
{ role: 'user', content: userMessage },
];
setMessages(newMessages);
setIsLoading(true);

  try {
    // Send to backend
    const response = await chatAPI.sendMessage(
      userMessage,
      environment,
      messages
    );

    // Add assistant response
    setMessages([
      ...newMessages,
      { role: 'assistant', content: response.message },
    ]);

    // Update tool traces
    if (response.tool_traces) {
      setToolTraces((prev) => [...prev, ...response.tool_traces]);
    }
  } catch (error) {
    console.error('Chat error:', error);
    setMessages([
      ...newMessages,
      {
        role: 'assistant',
        content: `Error: ${error.message}`,
      },
    ]);
  } finally {
    setIsLoading(false);
  }

};
```

return (

{/* Header */}
MCP DataOps Assistant

```
    {/* Chat */}
    <ChatWindow messages={messages} />

    {/* Input */}
    <InputBar onSend={handleSend} disabled={isLoading} />

    {/* Tool Traces */}
    <ToolTracePanel
      traces={toolTraces}
      isOpen={showTraces}
      onToggle={() => setShowTraces(!showTraces)}
    />
  </div>
```

);
}

export default App;

---

7. Dependencies

7.1 Backend Requirements

File: `backend/requirements.txt`

```
fastapi0.109.0
uvicorn[standard]0.27.0
pydantic2.5.3
pydantic-settings2.1.0
openai1.10.0
httpx0.26.0
python-dotenv==1.0.0
```

7.2 MCP Server Requirements

File: `mcp_server/requirements.txt`

```
fastapi0.109.0
uvicorn[standard]0.27.0
pydantic2.5.3
pydantic-settings2.1.0
azure-identity1.15.0
azure-mgmt-datafactory4.0.0
azure-keyvault-secrets4.7.0
azure-mgmt-resource23.0.1
python-dotenv==1.0.0
```

7.3 Frontend Dependencies

File: `frontend/package.json`

```
{
"name": "mcp-dataops-frontend",
"version": "1.0.0",
"type": "module",
"scripts": {
"dev": "vite",
"build": "vite build",
"preview": "vite preview"
},
"dependencies": {
"react": "^18.2.0",
"react-dom": "^18.2.0"
},
"devDependencies": {
"@vitejs/plugin-react": "^4.2.1",
"autoprefixer": "^10.4.17",
"postcss": "^8.4.33",
"tailwindcss": "^3.4.1",
"vite": "^5.0.12"
}
}
```

---

8. Deployment Instructions

8.1 Local Development

Start MCP Server:

```
cd mcp_server
python -m venv venv
source venv/bin/activate # On Windows: venv\Scripts\activate
pip install -r requirements.txt
cp .env.example .env
```

Edit .env with your Azure credentials

python main.py

Start Backend:

cd backend
python -m venv venv
source venv/bin/activate
pip install -r requirements.txt
cp .env.example .env

Edit .env with Azure and OpenAI credentials

python -m app.main

Start Frontend:

cd frontend
npm install
cp .env.example .env

Edit .env with backend URL

npm run dev

8.2 Docker Deployment

File: `docker-compose.yml`

version: '3.8'

services:
mcp-server:
build:
context: .
dockerfile: Dockerfile.mcp
ports:
- "8001:8001"
env_file:
- mcp_server/.env
volumes:
- ./logs:/app/logs
- ./infra/plans:/app/infra/plans

backend:
build:
context: .

```
dockerfile: Dockerfile.backend
ports:
- "8000:8000"
env_file:
- backend/.env
depends_on:
- mcp-server

frontend:
build:
context: ./frontend
ports:
- "3000:80"
depends_on:
- backend
```

---

9. Usage Examples

9.1 Example Queries

Dependency Failure Analysis:
"Why did the transformpipeline fail yesterday?"

Secret Impact:
"If I rotate the db-connection-string-prod secret, what will break?"

Environment Drift:
"Compare ingestpipeline between dev and prod environments"

Terraform Analysis:
"Explain the latest Terraform plan and highlight any risky changes"

Root Cause Analysis:
"Explain yesterday's ADF incident in simple terms"

References

[1] Pranjal Tiwari tech demo explanation document (provided)
[2] Pranjal Tiwari MCP-Based Developer Assistant specification (provided)