MCP-Auto-ML Technical Report

1. Prerequisites

1.1 System Requirements

Component	Requirement	Purpose
Python	3.10+ with uv package manager	runtime environment
Database	MongoDB 6.0+	Database
Cloud Storage	AWS S3 bucket	Model storage
Credentials	Kaggle API key	Dataset downloads
MCP Client	Claude/Anthropic or compatible	Protocol interaction
Networking	Ports 8000 (MCP) & 27017 (MongoDB)	Service communication

The following are system requirements. These need to be filled out in-order to have access to all of the MPC tools.

1.2 Initial Setup

```
# Install core dependencies using uv
uv pip install "fastapi>=0.110" "pandas>=2.2" "scikit-learn>=1.4"
"boto3>=1.34" "pymongo>=4.6" "kaggle>=1.6" "joblib>=1.3"

# Configure environment variables
echo "AWS_ACCESS_KEY_ID=your_key" >> .env
echo "S3_BUCKET=your-bucket-name" >> .env
```

The uv package manager must be used to start the project.

2. Tech Stack Architecture

Layer	Components	Protocol Integration
Data Ingestion	Kaggle API, pandas CSV parsing	MCP tools: download_kaggle_dataset
Data Processing	pandas, scikit-learn preprocessing	MCP tools: clean_dataset, transform
ML Modeling	scikit-learn, GridSearchCV	MCP tools: train_model, hyperparameter
Cloud Integration	boto3 (AWS S3), pymongo	MCP tools: save_model_to_s3
Visualization	matplotlib, seaborn	MCP tools: visualize_data_distribution
API Server	FastAPI, JSON-RPC 2.0	MCP protocol implementation

3. Core Tool Implementation

3.1 Data Ingestion Tools

download_kaggle_dataset()

```
@mcp.tool(description="Download Kaggle dataset")
async def download_kaggle_dataset(name: str, kaggle_url: str) -> str:
    # Regex extraction of dataset ID
    match = re.search(r"kaggle\.com/datasets/([^/]+/[^/?#]+)", kaggle_url)

    # Kaggle API authentication
    api = KaggleApi()
    api.authenticate()

    # Temporary directory for download
    with tempfile.TemporaryDirectory() as tmp_dir:
        api.dataset_download_files(match.group(1), path=tmp_dir, unzip=True)
        csv_file = next(f for f in os.listdir(tmp_dir) if f.endswith(".csv"))

        # Load into pandas and cache
        df = pd.read_csv(os.path.join(tmp_dir, csv_file))
        dataset_cache[name] = df
```

Parameters:

- `name`: Dataset identifier for caching
- `kaggle_url`: URL pattern: `https://www.kaggle.com/datasets/<user>/<dataset>`

The following tool is used to submit Kaggle dataset links. The tool, takes in the link then downloads the dataset for local use. It uses the Kaggle api to be able to access the datasets. The Kaggle api requires a Kaggle.json configuration file to be in the system. Once configured, it will be able to download any Kaggle dataset.

3.2 Data Processing Tools

clean_dataset()

```
@mcp.tool(description="Data cleaning pipeline")
async def clean_dataset(name: str, encode_categoricals: bool = True) -> str:
    df = dataset_cache[name]

    # Missing value handling
    for col in df.columns:
        if df[col].dtype in ['float64', 'int64']:
            df[col].fillna(df[col].mean(), inplace=True)
        elif df[col].dtype == 'object':
            df[col].fillna(df[col].mode()[^0], inplace=True)

    # Deduplication
```

```
    df = df.drop_duplicates()

    # Categorical encoding
    if encode_categoricals:
        df = pd.get_dummies(df, drop_first=True)

    dataset_cache[name] = df
```

Data Flow:

```
Raw Data → Missing Value Imputation → Deduplication → One-Hot Encoding → Clean Data
```

The following tool reads the dataset and cleans the data. It gets updates missing values, where if is an object, it uses the mode and if it uses numbers, then the mean. It also removed duplicates as well as encode the categorical columns to numbers. The tool finalizes the dataset to be ready for visualization and training.

3.3 Model Training Tools

train_model()

```
@mcp.tool(description="Model training endpoint")
async def train_model(name: str, target_column: str,
                     model_type: str = "classification",
                     model_name: Optional[str] = None) -> str:

    X = df.drop(columns=[target_column])
    y = df[target_column]
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

    model = self._get_model(model_type, model_name)
    model.fit(X_train, y_train)

    # Metrics calculation
    if model_type == "classification":
        acc = accuracy_score(y_test, model.predict(X_test))
        return f"Accuracy: {acc:.4f}"
    else:
        mse = mean_squared_error(y_test, model.predict(X_test))
        return f"MSE: {mse:.4f}"
```

Supported Models:

```
{
    "classification": {
        "logistic_regression": LogisticRegression(max_iter=1000),
        "random_forest": RandomForestClassifier(n_estimators=100),
```

```
            "svm": SVC(kernel='linear')
        },
        "regression": {
            "linear_regression": LinearRegression(),
            "random_forest": RandomForestRegressor(n_estimators=100)
        }
    }
```

The following tool is the heart of the program. It takes in the name of the dataset, the target variable, the type of model, and the specific model. For classification, i t is able to train logistic regression, random forest, and svm. For regression, it trains linear regression or random forest regressor. After training the model it provides an overall accuracy that was achieved. MSE for regression and accuracy for classification.

3.4 Cloud Integration Tools

save_model_to_s3()

```
@mcp.tool(description="Model persistence to AWS")
async def save_model_to_s3(name: str) -> str:
    model = model_cache[name]
    local_file = f"{name}_model.pkl"

    # Joblib serialization
    joblib.dump(model, local_file)

    # Boto3 S3 upload
    s3_client.upload_file(local_file, S3_BUCKET, local_file)

    # Cleanup
    os.remove(local_file)
```

AWS IAM Requirements:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["s3:PutObject"],
        "Resource": "arn:aws:s3:::your-bucket/*"
    }]
}
```

This tool saves the model to AWS S3 bucket. Any successfully trained model is stored to the user's cloud bucket incase they wish to use it or improve it in the future. It requires the boto3 library, and the system needs to be set up with the aws credentials.

4. Protocol Implementation Details

4.1 MCP Server Configuration

```
class FastMCP:
    def __init__(self, name: str):
        self.app = FastAPI()
        self.tools = []

        @self.app.post("/tools/execute")
        async def execute_tool(request: Dict[str, Any]):
            tool = next(t for t in self.tools if t['name'] == request['tool'])
            return await tool['function'](**request['parameters'])
```

This is what sets up the MCP Server. The code starts up the MCP server, which is ready to be given to a LLM. The MCP server will act as a context book or a tool box for the LLM to use when dealing with machine learning tasks.

4.2 JSON-RPC Communication

```
// Client Request
{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "tools/execute",
    "params": {
        "tool": "train_model",
        "parameters": {
            "name": "heart_disease",
            "target_column": "cp",
            "model_type": "classification"
        }
    }
}

// Server Response
{
    "jsonrpc": "2.0",
    "id": 1,
    "result": "Classification model trained. Accuracy: 0.9457"
}
```

This is how the client (Claude) sends requests to the MCP tool. The server processes the request, then sends a message back based on which tool was used. In the example, the machine learning tool was used, and it returns back the result of the model that was trained.

5. Optimization

5.1 Caching

```python
dataset_cache: Dict[str, pd.DataFrame] = {}
model_cache: Dict[str, Any] = {}

def _get_cached_item(cache: Dict, name: str) -> Any:
    if name not in cache:
        raise ValueError(f"Item '{name}' not in cache")
    return cache[name]
```

The following helps increase the speed of the responses, and prevents continues calls to Kaggle. Caching uses O(1) look up complexity and helps save memory for the calls to the dataset. It also prevents duplication of the dataset for occurring during the process.

5.2 Parallel Processing

```python
# GridSearchCV configuration for hyperparameter tuning
GridSearchCV(
    estimator=model,
    param_grid=param_grid,
    cv=5,
    n_jobs=-1,  # Utilize all CPU cores
    verbose=2
)
```

The following uses parallel processing to help search through possible parameters of the model, and look for the best one available. n_jobs = -1, uses multiple CPU cores to help receive a faster response for the best hyper-parameter.

6. Validation Metrics

Stage	Metric	Heart Disease Dataset Result
Data Cleaning	Final Shape	919 rows × 23 columns
Model Training	Accuracy (Logistic Reg)	94.57%
Hyperparameter Tuning	Best Parameters	{'C': 1, 'penalty': 'l2'}
Cloud Persistence	S3 Object Size	1.7 KB (serialized model)

The following the validation metrics for each of the tools. Clean tool will output the new shape of the data frame. Model training will output Accuracy. Hyperparameter tuning, will out the best parameters in the model. Cloud Persistence will output the size and where the model was stored.

Comparison: MCP-Auto-ML vs. Paid LLMs

MCP-Auto-ML extends what paid LLMs can do by providing real, automated execution of machine learning workflows and direct control over storage and deployment along with dataset. Paid LLMs, by themselves, are limited to generating code or text-they cannot execute and keep a consistent workflow. LLM's can't execute code or deploy applications. MCP is able to provide LLM's the ability to do, while maintaining security.

Key Differences

Feature/Capability	MCP-Auto-ML	Paid LLMs (GPT-4, Gemini, etc.)
Code Execution	Directly runs Python code, automates tasks	Cannot execute code, only generate text/code
Cloud Deployment	Saves models to AWS S3, data to MongoDB	Cannot deploy or persist models/data
Workflow Automation	Orchestrates multi-step ML pipelines	Can only describe steps or generate scripts
Tool/API Integration	Calls real APIs (Kaggle, AWS, MongoDB, etc.)	No native tool or API integration
Live Data Access	Downloads and processes external datasets	Cannot fetch or process external data
Result Persistence	Stores models and datasets for future use	No ability to persist or retrieve results
Token/Context Efficiency	Feeds only relevant data to LLM/agent	Limited by context window and token limits
Open Protocol/Extensibility	Open, can be extended with new tools easily	Closed, proprietary APIs
Security/User Control	User controls data flow and execution	Data leaves user control, less transparent

7. Conclusion

The MCP-Auto-ML system implements a complete ML pipeline through 10 specialized tools following the Model Context Protocol standard. It is able to give extensions to LLM's that they orginally couldn't do. MCP can give access to LLM's to be able to use external tools to complete the tasks given. Future enhancements could integrate AWS MCP Servers for improved cloud resource management and implement the new releases that is released by MCP. MCP is relatively new, so there is more enhacements to come in the future.