High-Level Design: BioSight Application

1. Introduction

BioSight is a web application designed for users to upload biological images, have them automatically classified using a machine learning model, and organize them based on the predicted class. The system also includes features for user authentication, monitoring, and an offline pipeline for detecting data drift and potentially retraining the classification model.

2. Goals & Requirements (Inferred)

- **Functional:**
  - User registration and authentication.
  - Upload single or multiple images.
  - Automatic classification of uploaded images using an ML model.
  - Organization of images into folders based on predicted class.
  - Allow users to view, download (as zip), and potentially correct image classifications.
  - Delete uploaded images.
  - Detect drift in incoming data compared to the original training data.
  - (Potentially) Retrain the ML model when significant drift is detected.
- **Non-Functional:**
  - Web-based access.
  - Reasonable prediction latency.
  - Monitoring of application performance and usage.
  - Maintainability and version control for code, data, and models.
  - Basic security for user authentication.

3. High-Level Architecture Overview

The system consists of several key components:

- **User Interface:** A browser-based frontend for user interaction.
- **Core Application:** A backend API (built with FastAPI) handling requests, business logic, ML inference, database interactions, and file management.
- **Supporting Systems:** Includes monitoring (Prometheus/Grafana) and potentially administrative interfaces.
- **Data & Model Lifecycle:** Manages the ML model, datasets, drift detection, and retraining processes using Git, DVC, and an offline pipeline (likely CI/CD).

(Refer to the HLD diagram for a visual representation)

4. Component Design & Rationale

- **4.1. User Interface (Browser-Based)**
    - **Design:** Standard HTML, CSS (`style.css`), and JavaScript served via FastAPI templates.
    - **Rationale:** Provides maximum accessibility for users across different devices without requiring native application installation. FastAPI's templating support (Jinja2) allows for straightforward integration with the backend.
- **4.2. Core Application (FastAPI Backend)**
    - **Design:** A monolithic Python backend using the FastAPI framework (`app.py`). It handles API requests, authentication, ML model interaction, database operations, and file system management.
    - **Rationale:**
        - **Python:** Excellent ecosystem for machine learning (PyTorch, scikit-learn, etc.) and web development.
        - **FastAPI:** Modern, high-performance framework leveraging asynchronous capabilities (suitable for I/O-bound tasks like file uploads and database calls), automatic data validation (Pydantic), and interactive API documentation (Swagger UI).
        - **Monolith:** Simpler to develop, deploy, and manage initially for a project of this scope compared to a microservices architecture. Trade-off is potentially tighter coupling between components.
- **4.3. ML Model Integration**
    - **Design:** The ML classification model (PyTorch/ResNet) is loaded directly within the FastAPI application process (`image_processor.py` , `model_loader.py`). Predictions are made synchronously within the `/upload` request handler.
    - **Rationale:** Simplifies deployment as the model is part of the main application container. Suitable for moderate traffic; for very high throughput or complex models, a separate ML serving microservice might be considered to decouple scaling and resource needs.
- **4.4. Database (MongoDB)**
    - **Design:** MongoDB (NoSQL document database) is used to store user credentials and metadata about uploaded images (filenames, paths, predictions, timestamps, user info, drift status, etc.) (`database.py`).
    - **Rationale:**
        - **Schema Flexibility:** Easily accommodates evolving metadata requirements without strict schema migrations typical of SQL databases. Useful for storing varied information related to images and ML processes.
        - **Ease of Use:** Python drivers (like `pymongo`) integrate well. Document structure maps naturally to Python dictionaries.
        - **Scalability:** MongoDB offers horizontal scaling capabilities if needed in the future.

- **4.5. File Storage**
    - **Design:** Uses the local file system within the application's container/server environment for storing uploaded images (`uploads/`), organized images (`organized/`), and temporary zip files. Paths are stored in the database.
    - **Rationale:** Simple to implement for a single-instance deployment. **Trade-off:** This approach does not scale horizontally easily (multiple instances wouldn't share the same local file system). For scalability, cloud-based object storage (like AWS S3, GCS) would be a better choice, requiring changes to file handling logic.
- **4.6. Monitoring (Prometheus/Grafana)**
    - **Design:** FastAPI application exposes a `/metrics` endpoint (`monitoring.py` , `app.py`) using the `prometheus-client` library. Metrics include HTTP request counts, prediction latency, upload counts, etc. An external Prometheus server scrapes this endpoint, and Grafana visualizes the data.
    - **Rationale:** Industry-standard, open-source solution for metrics collection and visualization. Provides crucial insights into application performance, resource usage, and potential bottlenecks.
- **4.7. Offline Pipeline (Drift Detection/Retraining)**
    - **Design:** A separate process, likely orchestrated via CI/CD (e.g., GitHub Actions defined in `drift_pipeline.yml`), responsible for checking data drift and potentially retraining the model. It interacts with the database, file storage (via DVC), and DVC remote storage.
    - **Rationale:** Decouples computationally intensive and potentially long-running tasks (drift analysis, retraining) from the main request-response cycle of the web application, preventing performance degradation. Allows these tasks to run on different schedules or triggers (e.g., on code push, nightly).
- **4.8. Versioning (Git/DVC)**
    - **Design:** Git is used for versioning application code, configuration files (`params.yaml`), DVC metadata files (`.dvc` , `dvc.lock`), and workflow definitions. DVC is used for versioning large data files (datasets) and potentially ML models, storing them in remote storage (e.g., Google Drive, S3).
    - **Rationale:** Git is standard for code version control. DVC complements Git by handling large files efficiently, enabling reproducibility of experiments and data pipelines without bloating the Git repository.

5. Key Design Choices & Trade-offs

- **Monolith vs. Microservices:** Chose a monolithic approach for initial simplicity. Trade-off: Less flexibility in independent scaling of components.

- **Embedded ML Model vs. Separate Service:** Embedded model simplifies deployment. Trade-off: Ties model resource needs to the web server; less scalable for heavy ML inference.
- **NoSQL (MongoDB) vs. SQL:** Chose NoSQL for schema flexibility suitable for evolving metadata. Trade-off: Less emphasis on relational integrity compared to SQL.
- **Local File Storage vs. Object Storage:** Chose local storage for simplicity. Trade-off: Significant limitation for horizontal scaling.

6. Scalability & Deployment Considerations

- **Deployment:** The application is designed to be containerized (likely using Docker), simplifying dependency management and deployment.
- **Scalability:**
  - The FastAPI backend can be scaled horizontally using multiple container instances behind a load balancer, *provided that* file storage is migrated to a shared solution (like S3/GCS) and a shared cache/session store is used if necessary.
  - MongoDB supports horizontal scaling (sharding).
  - The offline pipeline can be scaled independently based on its workload.

This HLD provides a strategic overview of the BioSight system's design, highlighting the purpose of each major component and the reasoning behind the chosen technologies and approaches.