Department of Computer Engineering

CENG350 Software Engineering

Software Architecture Description (SAD) for KSO

Group 90

By

Aneliya Abdimalik, 2547651

Mukhammadrizo Maribjonov, 2742922

Author Name 3, SID

Friday 16th May, 2025

Contents

List of Figures

List of Tables

Revision History

(Clause 9.2.1)

You can use a table to show your reports' versions and their date.

To create tables, you can use https://www.latex-tables.com/; the Table 1 is generated by it.

A	B	C	D

Table 1: Table Example

1. Introduction

1.1 Purpose and objectives of KSO

The KSO is an open-source initiative focused on monitoring marine biodiversity in Sweden's first marine national park. Its main objective is to analyze large volumes of underwater video footage by integrating Citizen Science with Machine Learning techniques. Volunteers annotate video clips to identify marine life, and these annotations are then used to train AI models that automate the identification process. The system promotes open collaboration, reproducible research, and scalable biodiversity monitoring.

1.2 Scope

This SAD models the current state of the KSO system and includes suggestions for potential improvements. It presents architectural views according to the ISO/IEC/IEEE 42010 standard and the viewpoint definitions from Rozanski and Woods. The SAD describes how the system's components are organized and interact, how information is stored and processed, how the software is developed and maintained, and how it is deployed. The SAD does not detail low-level hardware, internal workings of third-party tools, or operational procedures not directly tied to the architecture.

1.3 Stakeholders and their concerns

- **Marine researchers** are interested in collecting accurate and comprehensive ecological data and identifying species efficiently.
- Citizen Science contributors care about the usability and accessibility of the annotation platform, which allows them to contribute without technical barriers.
- **Software developers** are concerned with the modularity and maintainability of the codebase to support continuous development and integration.
- **System administrators** are responsible for deployment and upkeep of the system, so their concerns include system stability, scalability, and resource requirements.
- **Data scientists** need well-structured, high-quality annotated data to train and evaluate Machine Learning models effectively.
- **Environmental agencies** rely on the KSO system to generate reliable biodiversity monitoring data that can inform policy and conservation decisions.

2. References

1. ISO/IEC/IEEE 29148:2018 – Systems and Software Engineering — Life Cycle Processes — Requirements Engineering
2. Rozanski, N., Woods, E. – Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives
3. Anton, V., Germishuys, J., Bergström, P., Lindegarth, M., Obst, M. (2021) – An Open-Source, Citizen Science and Machine Learning Approach to Analyse Subsea Movies, Biodiversity Data Journal
4. Koster Seafloor Observatory GitHub Repository – https://github.com/WildlifeAI/koster-observatory
5. CENG350 Spring 2024–2025 Project Specification Document
6. SAD Outline and Evaluation for KSO v2.0
7. SUBSIM platform – Sweden's national system for subsea image analysis and citizen science data collection

3. Glossary

Citizen Science Scientific research conducted in part by non-professional scientists, often through public volunteer contributions such as video annotation.

KSO Koster Seafloor Observatory.

Machine Learning A field of artificial intelligence where systems learn patterns from data to make predictions or decisions.

SAD System Analysis and Design.

4. Architectural Views

4.1 Functional View

Refer to *R&W Chapter 17*.

4.1.1 Stakeholders' Uses of This View

Citizen science developers use this view to understand the integration between the annotation platform and backend services. Machine learning engineers rely on it to trace the flow from aggregated annotations to model training. System integrators refer to this view to ensure internal components align with service boundaries.

4.1.2 Component Diagram

KSO Component diagram is show in 4.1. The KSO system is organized into three top-level components—**Data Management**, **Citizen Science**, and **Machine Learning & HPC**—each encapsulating a clear set of sub-parts and well-defined interface contracts:

- **Data Management**:
  - **Cold Storage** Server provides IMovieArchive for long-term movie storage.
  - **Hot Storage** Server provides IFrameCache and consumes IMovieArchive for fast access to extracted frames.
  - **SQLite Database** provides IDatabaseRepository for all CRUD operations on movies, sites, species, and annotations.

- **Citizen Sciencet**
  - **Zooniverse Interface** provides both IZooUploadService and IZooDownloadService, consuming IMovieArchive and IFrameCache to upload media and retrieve raw annotations.
  - **Clip Annotation Aggregator** provides IClipAggregationService by processing downloaded clip classifications and writing consensus labels via IDatabaseRepository.
  - **Frame Annotation Aggregator** provides IFrameAggregationService similarly for frame-level annotations.
- **Machine Learning & HPC**
  - **Model Trainer** provides IModelTrainingService, consuming aggregated labels (IClipAggregationService & IFrameAggregationService) and data storage (IDatabaseRepository) to train models.
  - **Model Evaluator** provides IModelEvaluationService, consuming trained models and database access to compute performance metrics.
  - **API Server** provides IInferenceService, consuming evaluation results and database metadata to serve real-time predictions.

Each component in the diagram is annotated with `<<provides>>` and `<<requires>>` stereotypes to represent their service contracts. For instance, the *Aggregator* component requires annotations from the *Annotation Interface* and provides consensus outputs to the *ML Trainer*, facilitating modular interaction.

4.1.3 Internal Interfaces

Internal interfaces diagram is shown in 4.2.

- Data Management Component: These interfaces manage data storage and retrieval.

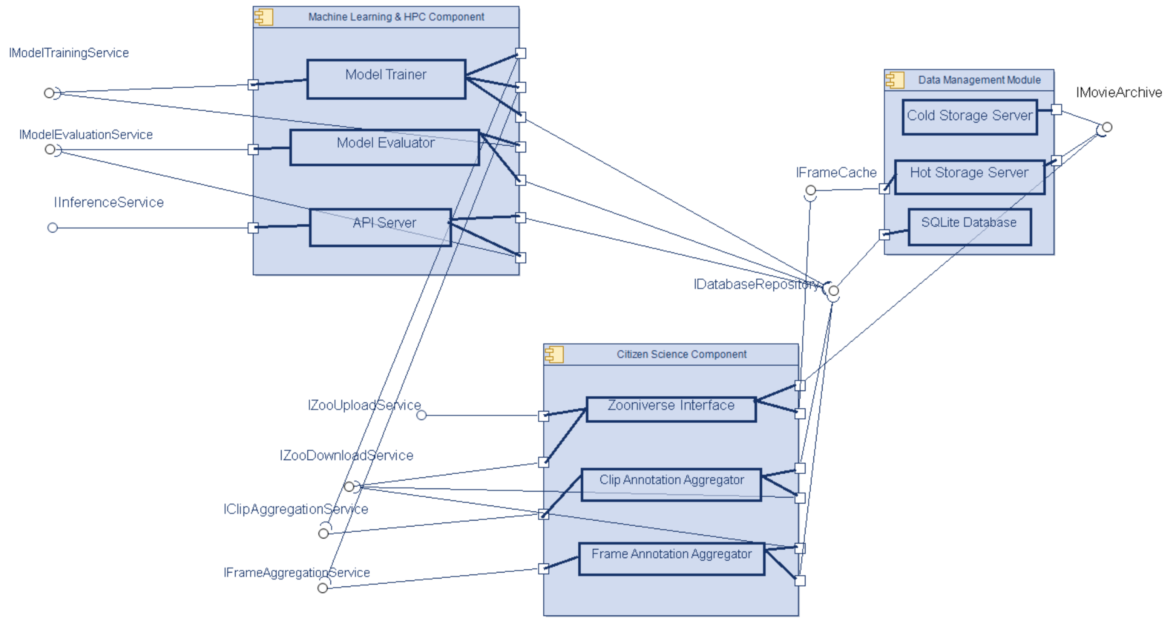

Figure 4.1: Component Diagram for KSO

- IMovieArchive: Manages storage and retrieval of movie data.
  - Fields:
    - archivePath: String – Directory path where movies are stored.
    - maxStorageSize: int – Maximum storage capacity (in MB).
  - Methods:
    - getMovie(id: String): MovieData – Retrieves a movie by ID.
    - storeMovie(data: MovieData): void – Stores a movie in the archive.
- IFrameCache: Caches frame data for quick access.
  - Fields:
    - cacheSize: int – Maximum number of frames the cache can hold.
    - cachePolicy: String – Eviction policy (e.g., "LRU" for Least Recently Used).
  - Methods:
    - getFrame(id: String): FrameData – Retrieves a frame from the cache.

- · putFrame(data: FrameData): void – Adds a frame to the cache.

- IDatabaseRepository: Handles CRUD operations for a database.
  - Fields:
    - · dbConnectionString: String – Connection string for the database.
    - · tables: List¡String¿ – List of table names managed by the repository.
  - Methods:
    - · create(table: String, data: Object): void – Creates a new record.
    - · read(table: String, id: String): Object – Reads a record by ID.
    - · update(table: String, id: String, data: Object): void – Updates a record.
    - · delete(table: String, id: String): void – Deletes a record.

- Citizen Science Component: These interfaces support interactions with citizen science platforms and data aggregation.
  - IZooUploadService: Uploads data to a platform like Zooniverse.
    - Fields:
      - · zooniverseAPIKey: String – API key for authentication.
      - · uploadEndpoint: String – URL for the upload service.
    - Methods:
      - · uploadClip(clip: ClipData): void – Uploads a video clip.
      - · uploadFrame(frame: FrameData): void – Uploads a single frame.
  - IZooDownloadService: Downloads annotations from a citizen science platform.
    - Fields:
      - · zooniverseAPIKey: String – API key for authentication.
      - · downloadEndpoint: String – URL for downloading annotations.
    - Methods:

      · getFrame(id: String): FrameData – Retrieves a frame from the cache.
      · putFrame(data: FrameData): void – Adds a frame to the cache.
      · downloadAnnotations(subjectId: String): AnnotationData – Retrieves annotations for a subject.

  – IClipAggregationService: Aggregates citizen annotations for video clips.
    * Fields:
      · aggregationThreshold: float – Minimum agreement level (e.g., 0.75 for 75%).
      · minUsers: int – Minimum number of users required for aggregation.
    * Methods:
      · aggregateClipAnnotations(annotations: List¡AnnotationData¿): ConsensusLabel – Produces a consensus label from annotations.
  – IFrameAggregationService: Aggregates citizen annotations for individual frames.
    * Fields:
      · aggregationThreshold: float – Minimum agreement level (e.g., 0.75).
      · minUsers: int – Minimum number of users required.
    * Methods:
      · aggregateFrameAnnotations(annotations: List¡AnnotationData¿): ConsensusLabel – Produces a consensus label for frames.

- Machine Learning & HPC Component: These interfaces handle model training, evaluation, and inference.
  – IModelTrainingService: Trains machine learning models.
    * Fields:
      · modelType: String – Type of model (e.g., ”YOLOv3”, ”ResNet”).

        · trainingParameters: Map¡String, Object¿ – Key-value pairs of hyperparameters (e.g., learning rate, epochs).

    ∗ Methods:

        · trainModel(data: TrainingData): Model – Trains and returns a model.

- IModelEvaluationService: Evaluates trained models.

    ∗ Fields:

        · evaluationMetrics: List¡String¿ – Metrics to compute (e.g., ”precision”, ”recall”, ”F1”).

    ∗ Methods:

        · evaluateModel(model: Model, testData: TestData): EvaluationMetrics – Returns evaluation results.

- IInferenceService: Performs predictions using trained models.

    ∗ Fields:

        · inferenceEndpoint: String – URL or address for inference service.

        · supportedModels: List¡String¿ – List of model names available for inference.

    ∗ Methods:

        · predict(input: InputData): Prediction – Generates a prediction from input data.

4.1.4 Interaction Patterns

Figure 4.3 shows the interaction between the Zooniverse Interface, Hot Storage Service, Cold Storage Service, and the Zooniverse Upload Service. The Zooniverse Interface requests the frames for a given movie segment from the Hot Storage Service. The Hot Storage Service, lacking the raw movie locally, fetches the MovieStream from the Cold Storage Service and returns it. It then extracts and returns the List¡Frame¿ to

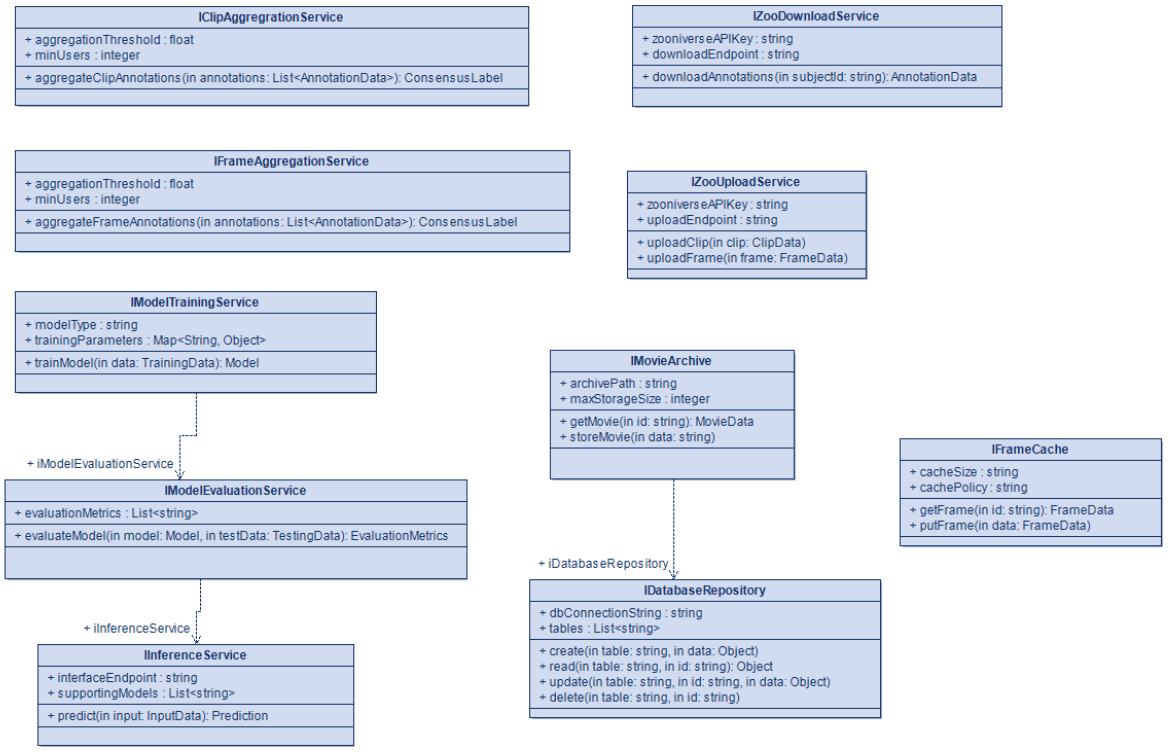
IClipAggregrationService
+ aggregationThreshold : float
+ minUsers : integer
+ aggregateClipAnnotations(in annotations: List<AnnotationData>): ConsensusLabel
IZooDownloadService
+ zooniverseAPIKey : string
+ downloadEndpoint : string
+ downloadAnnotations(in subjectId: string): AnnotationData
IFrameAggregationService
+ aggregationThreshold : float
+ minUsers : integer
+ aggregateFrameAnnotations(in annotations: List<AnnotationData>): ConsensusLabel
IZooUploadService
+ zooniverseAPIKey : string
+ uploadEndpoint : string
+ uploadClip(in clip: ClipData)
+ uploadFrame(in frame: FrameData)
IModelTrainingService
+ modelType : string
+ trainingParameters : Map<String, Object>
+ trainModel(in data: TrainingData): Model
IMovieArchive
+ archivePath : string
+ maxStorageSize : integer
+ getMovie(in id: string): MovieData
+ storeMovie(in data: string)
IFrameCache
+ cacheSize : string
+ cachePolicy : string
+ getFrame(in id: string): FrameData
+ putFrame(in data: FrameData)
+ iModelEvaluationService
IModelEvaluationService
+ evaluationMetrics : List<string>
+ evaluateModel(in model: Model, in testData: TestingData): EvaluationMetrics
+ iDatabaseRepository
IDatabaseRepository
+ dbConnectionString : string
+ tables : List<string>
+ create(in table: string, in data: Object)
+ read(in table: string, in id: string): Object
+ update(in table: string, in id: string, in data: Object)
+ delete(in table: string, in id: string)
+ iInferenceService
IInferenceService
+ interfaceEndpoint : string
+ supportingModels : List<string>
+ predict(in input: InputData): Prediction

Figure 4.2: KSO Internal Interface Class Diagram

the Zooniverse Interface, which invokes the Zooniverse Upload Service to push the clip (frames + metadata) to Zooniverse and finally receives an UploadResponse (status + uploadId).

Figure 4.4 shows the interaction between the Clip Annotation Aggregator, the Zooniverse Download Service, and the SQLite Database. The Aggregator calls fetchClassifications on the Download Service to retrieve all raw clip classifications. Upon receiving the list of Classifications, it processes them into AggregatedClipData and then persists those consensus labels via saveAggregatedClipLabels on the SQLite Database, receiving an acknowledgment once storage is complete.

Figure 4.5 shows the interaction among the Model Trainer, Clip Annotation Aggregator, Frame Annotation Aggregator, SQLite Database, and the external ML Framework. The Model Trainer first retrieves ClipDataset and FrameDataset by calling getAggregatedClipData and getAggregatedFrameData. It then fetches its TrainingConfig from the Database, passes the combined training data and config to the ML

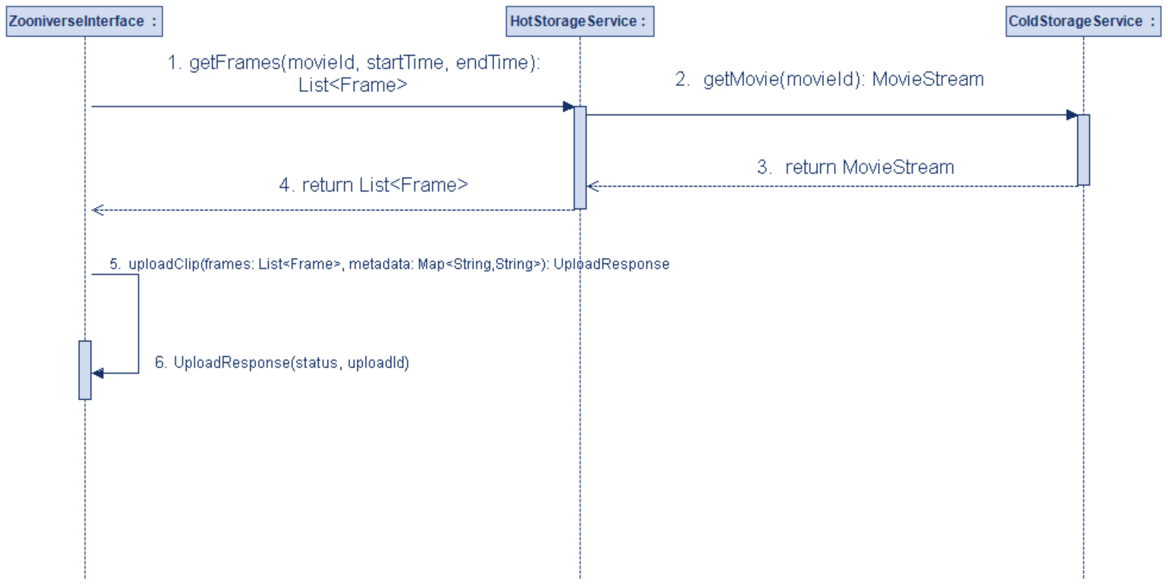
ZooniverseInterface :
HotStorageService :
ColdStorageService :
1. getFrames(movieId, startTime, endTime): List<Frame>
2. getMovie(movieId): MovieStream
3. return MovieStream
4. return List<Frame>
5. uploadClip(frames: List<Frame>, metadata: Map<String,String>): UploadResponse
6. UploadResponse(status, uploadId)

Figure 4.3: Clip Generation and Upload Sequence Diagram

Framework's trainModel operation, and upon receiving the completed ModelArtifact, it saves the resulting model metadata back to the Database and receives an acknowledgment.

4.1.5 External Interfaces

Operations Descriptions of 4.6:

- IKSOClientAPI
  - submitClipRequest(...): client apps call this to request new clip creation.
  - submitFrameRequest(...): upload a single frame with metadata.
  - getAggregationResults(...): retrieve consensus labels for a clip or frame.
  - getModelStatus(...): check training run progress or completion.
- IZooniverseREST
  - uploadSubject(...): post new media (clips/frames) to a Zooniverse workflow.
  - fetchClassifications(...): pull raw classifications for a given subject.

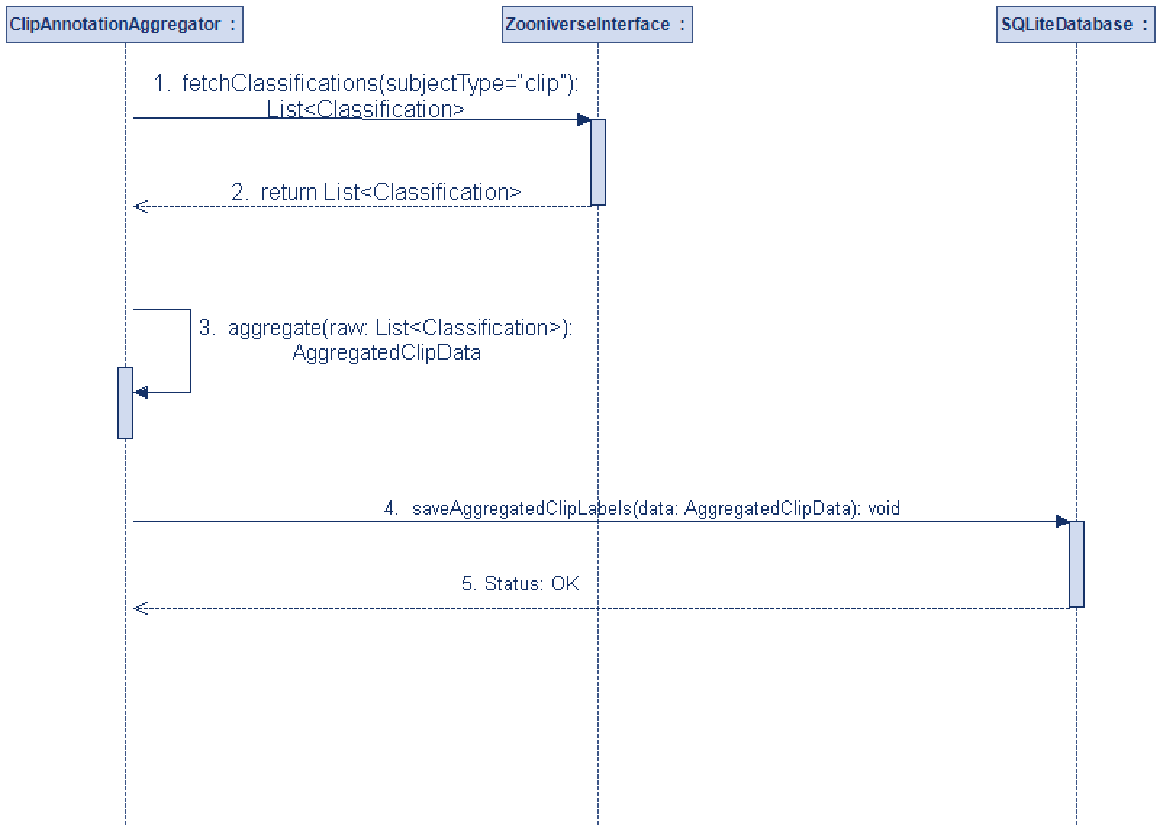
ClipAnnotationAggregator :
ZooniverseInterface :
SQLiteDatabase :
1. fetchClassifications(subjectType="clip"): List<Classification>
2. return List<Classification>
3. aggregate(raw: List<Classification>): AggregatedClipData
4. saveAggregatedClipLabels(data: AggregatedClipData): void
5. Status: OK

Figure 4.4: Clip Annotation Aggregation Sequence Diagram

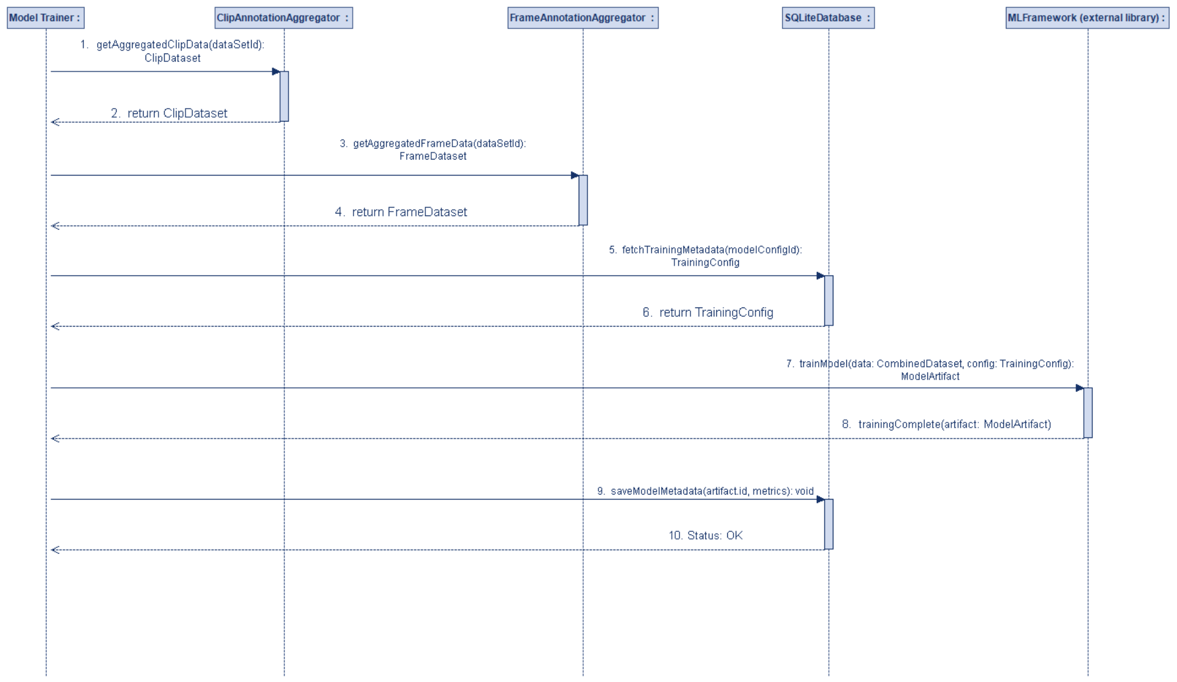
Model Trainer :
ClipAnnotationAggregator :
FrameAnnotationAggregator :
SQLiteDatabase :
MLFramework (external library) :
1. getAggregatedClipData(dataSetId): ClipDataset
2. return ClipDataset
3. getAggregatedFrameData(dataSetId): FrameDataset
4. return FrameDataset
5. fetchTrainingMetadata(modelConfigId): TrainingConfig
6. return TrainingConfig
7. trainModel(data: CombinedDataset, config: TrainingConfig): ModelArtifact
8. trainingComplete(artifact: ModelArtifact)
9. saveModelMetadata(artifact.id, metrics): void
10. Status: OK

Figure 4.5: Model Training Workflow Sequence Diagram

  - getSubjectStatus(...): query Zooniverse for upload/processing status.

- IInferenceAPI
  - predictClip(...)   predictFrame(...): obtain model predictions for media.
  - listAvailableModels(): discover which trained models are currently deployable.

4.1.6 Interaction scenarios

This scenario 4.7 begins when an external client invokes submitClipRequest on the KSO Client API with the target movie and time range. The system validates the request, extracts frames via the Hot and Cold Storage services, and uploads the clip to Zooniverse using the Zooniverse REST interface. It then enters a polling loop—repeatedly fetching raw classifications until they become available—before aggregating them into consensus labels and returning the final AggregatedClipData to the client.

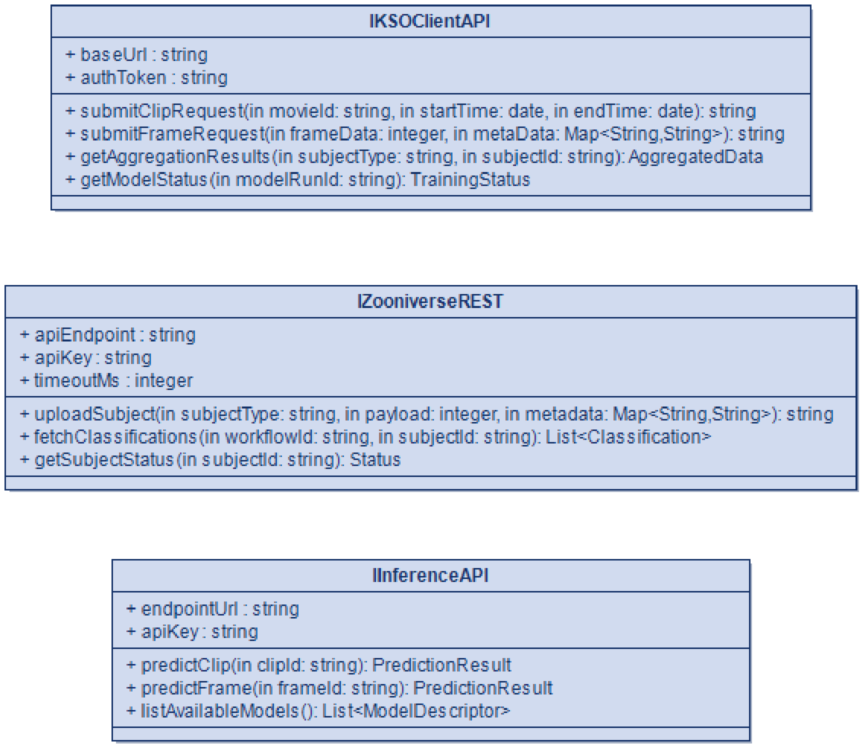
IKSOClientAPI
+ baseUrl : string
+ authToken : string
+ submitClipRequest(in movieId: string, in startTime: date, in endTime: date): string
+ submitFrameRequest(in frameData: integer, in metaData: Map<String,String>): string
+ getAggregationResults(in subjectType: string, in subjectId: string): AggregatedData
+ getModelStatus(in modelRunId: string): TrainingStatus
IZooniverseREST
+ apiEndpoint : string
+ apiKey : string
+ timeoutMs : integer
+ uploadSubject(in subjectType: string, in payload: integer, in metadata: Map<String,String>): string
+ fetchClassifications(in workflowId: string, in subjectId: string): List<Classification>
+ getSubjectStatus(in subjectId: string): Status
IInferenceAPI
+ endpointUrl : string
+ apiKey : string
+ predictClip(in clipId: string): PredictionResult
+ predictFrame(in frameId: string): PredictionResult
+ listAvailableModels(): List<ModelDescriptor>

Figure 4.6: KSO External Interface Class Diagram

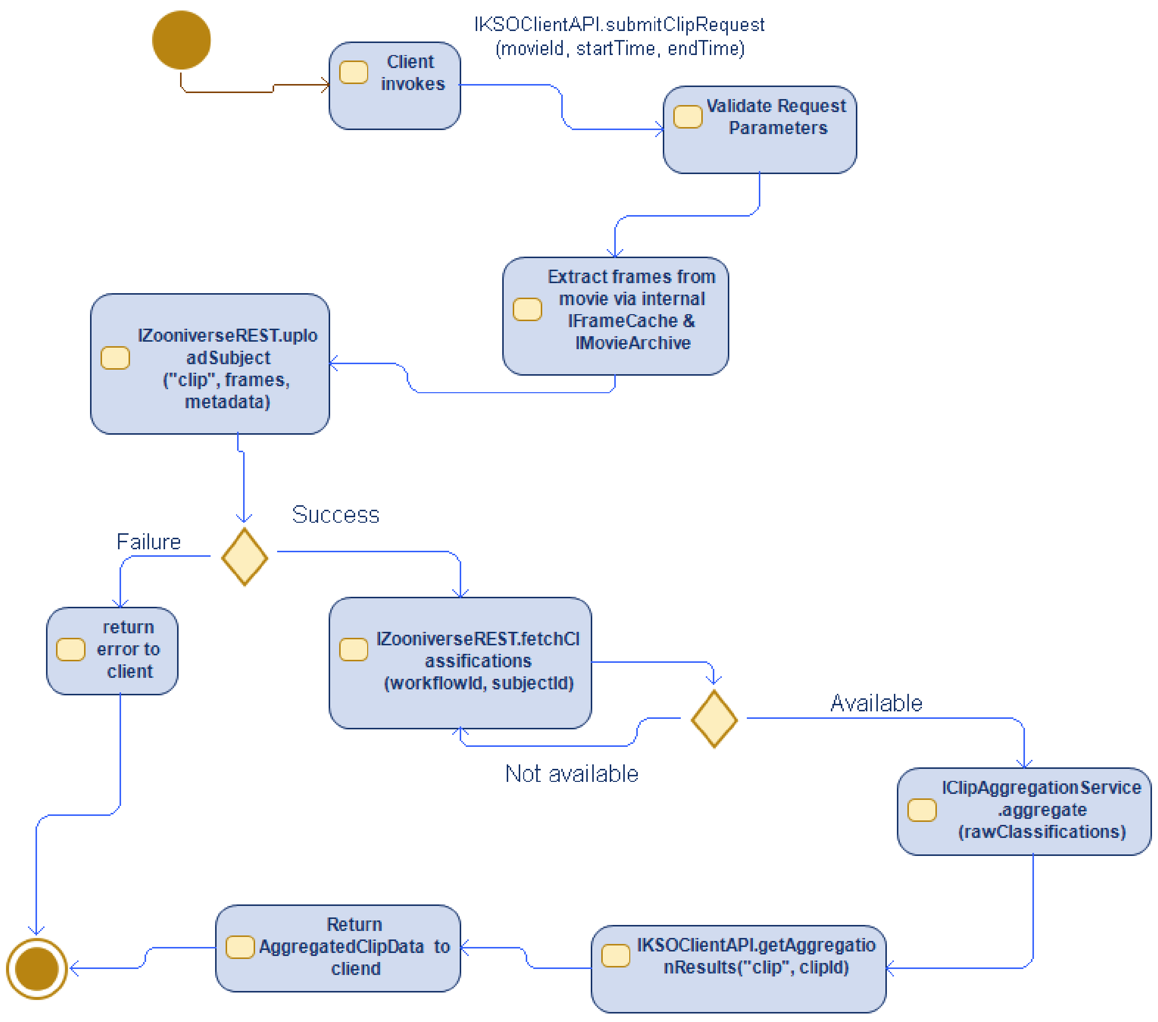
IKSOClientAPI.submitClipRequest (movieId, startTime, endTime)
Client invokes
Validate Request Parameters
Extract frames from movie via internal IFrameCache & IMovieArchive
IZooniverseREST.uploadSubject ("clip", frames, metadata)
Success
Failure
return error to client
IZooniverseREST.fetchClassifications (workflowId, subjectId)
Available
Not available
IClipAggregationService .aggregate (rawClassifications)
IKSOClientAPI.getAggregationResults("clip", clipId)
Return AggregatedClipData to cliend

Figure 4.7: Clip Submission & Aggregation Workflow

A client calls predictClip (or predictFrame) on the Inference API, triggering authentication 4.8. Upon successful validation, the API Server retrieves the chosen model's metadata from the database and invokes the inference engine with the stored model artifact. Finally, the engine returns a PredictionResult, which the API Server forwards back to the client.

This activity diagram models the scenario of user-initiated clip upload, beginning with file selection, followed by metadata generation, validation, and eventual storage. Decision nodes represent conditional logic such as format validation and user authorization.

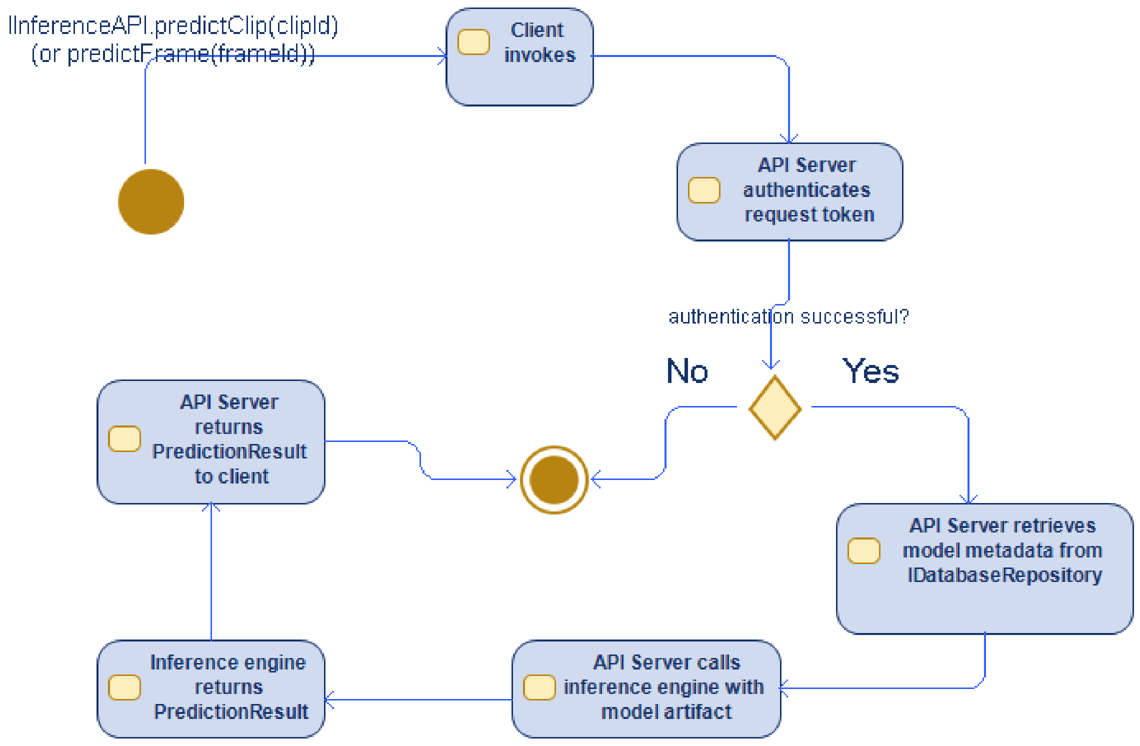
IInferenceAPI.predictClip(clipId)
(or predictFrame(frameId))
Client invokes
API Server authenticates request token
authentication successful?
No
Yes
API Server retrieves model metadata from IDatabaseRepository
API Server calls inference engine with model artifact
Inference engine returns PredictionResult
API Server returns PredictionResult to client

Figure 4.8: Real-Time Inference Workflow

4.2 Information View

Refer to *R&W Chapter 18.*

4.2.1 Stakeholders' Uses of This View

Machine learning developers use this view to access high-quality training datasets derived from annotated subsea footage. Database administrators rely on this information to maintain schema consistency and optimize query performance. Ecological researchers use the structured data to perform biodiversity analysis and generate reports.

4.2.2 Database Class Diagram

Database Class Diagram 4.9 involving the key database or main memory objects. Complete with relevant associations. Descriptions of the non-obvious names (for classes, attributes, operations) should also be given. Attributes such as `confidenceLevel` (a float between 0.0 and 1.0) and `speciesLabel` (a taxonomy-compliant string) are critical for downstream analysis and model training. Class associations between `Clip`, `Frame`, and `Annotation` reflect the flow of observational data through the system.

4.2.3 Operations on Data

Descriptions of the operations are given in the database class diagram. These operations may deal with the storage and handling of information regarding the application domain (ocean floor observation) entities.
Operations should be listed in tabular form or as bullet items.
These usually include CRUD (Create Read Update Delete) operations.

- **Clip**: `createClip()`, `fetchClipByID()`, `deleteClip()`
- **Frame**: `extractFramesFromClip()`, `getFrame()`, `updateFrame()`
- **Annotation**: `addAnnotation()`, `getUserAnnotations()`, `resolveAnnotations()`

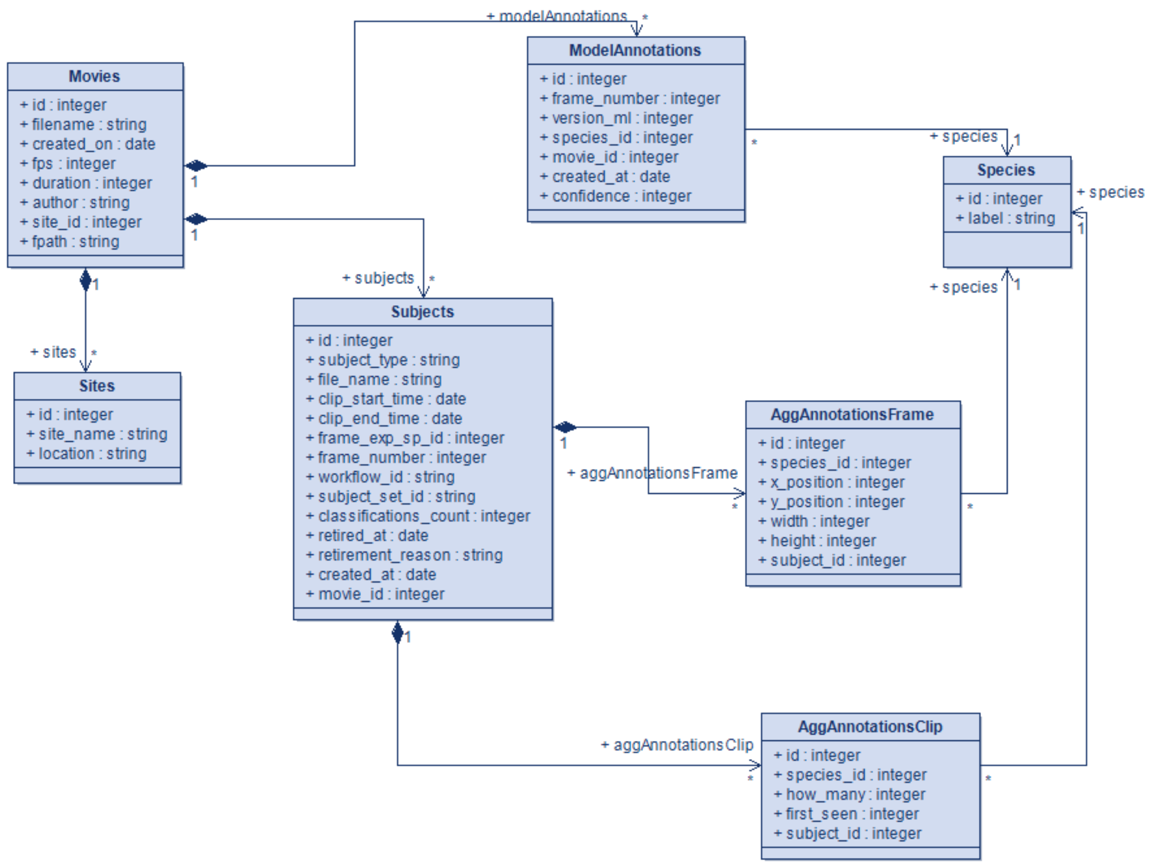

Figure 4.9: KSO Database Class Diagram

- **Species**: `addSpecies()`, `listSpecies()`

4.3 Development View

Refer to *R&W Chapter 20*.

4.3.1 Stakeholders' Uses of This View

4.3.2 Package Diagram

Package Contents & Dependencies 4.10:

- DataManagement
  - ColdStorage: long-term movie archive
  - HotStorage: frame cache for quick reads/writes
  - Database: relational store for metadata and aggregated labels
- CitizenScience
  - ZooniverseInterface: upload/download service adapters
  - ClipAggregation: consolidates clip classifications
  - FrameAggregation: consolidates frame classifications
- ML_HPC
  - ModelTraining: orchestrates dataset loading and YOLO training
  - ModelEvaluation: computes metrics on hold-out data
  - InferenceAPI: serves real-time predictions via REST

4.4 Deployment View

Refer to *R&W Chapter 21*.

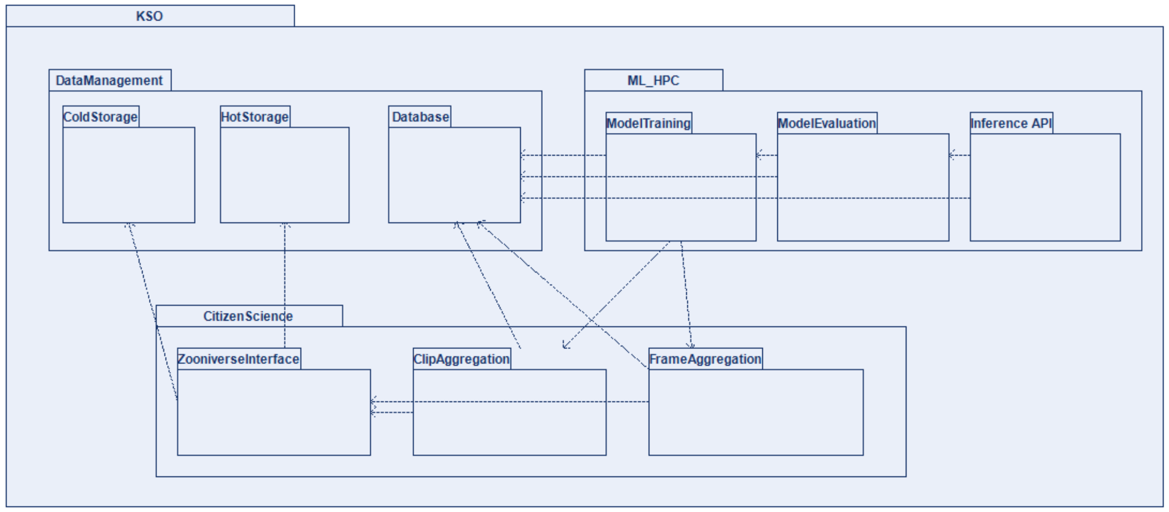

Figure 4.10: KSO Package Diagram

4.4.1 Stakeholders' Uses of This View

4.4.2 Deployment Diagram

The system is deployed using a hybrid architecture. The frontend web interface and RESTful API backend are deployed in separate containers for scalability. A GPU-enabled compute node runs the machine learning trainer. The SQLite database is attached to a persistent volume on a dedicated container. External services such as Zooniverse and SUBSIM are accessed securely through API gateways, ensuring data integrity and controlled access.

4.5 Design Rationale

Each architectural view reflects specific trade-offs made to meet project goals. In the Functional View, a modular approach was favored to simplify testing and integration. The Information View prioritizes normalized, frame-level data for fine-grained ML training. The Development View aligns packages with domain logic to improve maintainability. In the Deployment View, ML components are isolated on GPU nodes to ensure high performance without impacting core services.

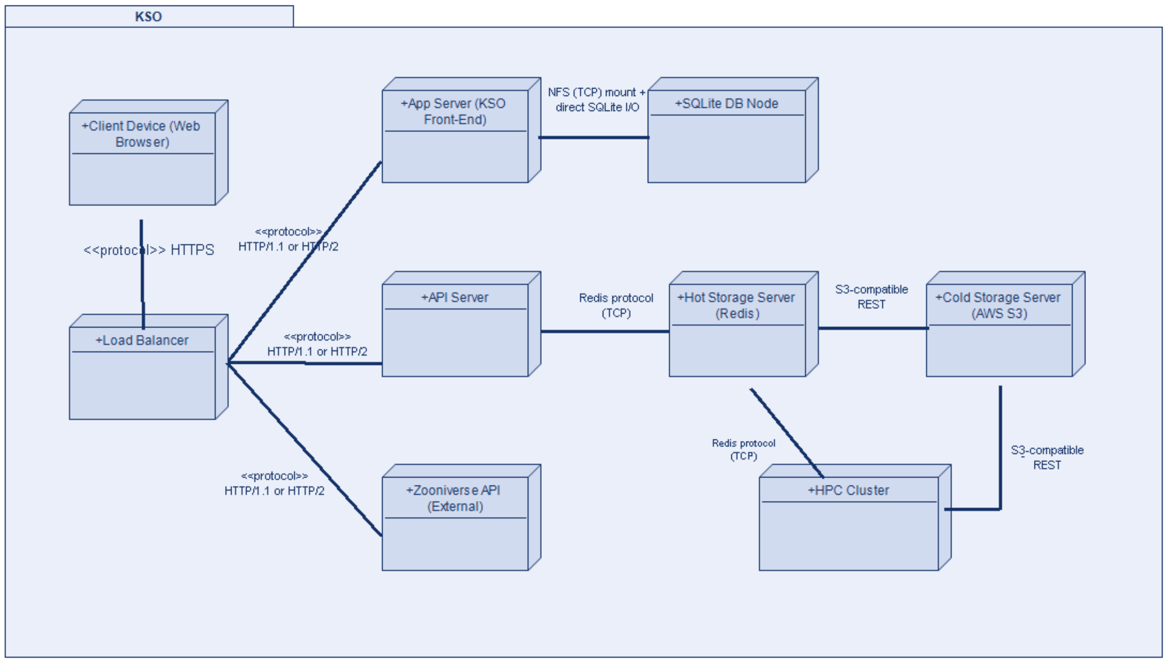
KSO
+Client Device (Web Browser)
<<protocol>> HTTPS
+Load Balancer
<<protocol>> HTTP/1.1 or HTTP/2
<<protocol>> HTTP/1.1 or HTTP/2
<<protocol>> HTTP/1.1 or HTTP/2
+App Server (KSO Front-End)
NFS (TCP) mount + direct SQLite I/O
+SQLite DB Node
+API Server
Redis protocol (TCP)
+Hot Storage Server (Redis)
S3-compatible REST
+Cold Storage Server (AWS S3)
Redis protocol (TCP)
S3-compatible REST
+Zooniverse API (External)
+HPC Cluster

Figure 4.11: KSO Deployment Diagram

5. Architectural Views for Your Suggestions to Improve the Existing System

5.1 Functional View

Refer to *R&W Chapter 17*.

5.1.1 Stakeholders' Uses of This View

This updated functional view is mainly useful for infrastructure developers, backend engineers, and system integrators. The new components like the API Gateway and Auth Service help developers manage external access and simplify how internal services interact. Using the Event Bus also helps engineers build more scalable systems, since everything doesn't have to happen at the same time. The view shows how all parts now depend on well-defined interfaces, which makes the system easier to update and maintain.

5.1.2 Component Diagram

Proposed Enhanced Component Diagram is shown in 5.1.

To improve decoupling, scalability, and security, we introduce three new "infrastructure" components—an API Gateway, an Authentication Service, and an Event

Bus—and refactor the Database into a single Database Service abstraction. The top-level remains Data Management, Citizen Science, and ML& HPC, but each subsystem now depends on the new infrastructure layer rather than talking directly to each other.

Key Improvements:

- API Gateway centralizes routing, logging, rate-limiting, and enforces use of IAuthService for every external call.
- Auth Service decouples credential management, so components need only talk the simple IAuthService contract.
- Event Bus (e.g. Kafka/RabbitMQ) enables truly asynchronous pipelines: clips → annotations → training → evaluation → inference.
- Database Service abstracts away direct DB mounts, allowing you to scale or swap the underlying engine without touching components.

This refactoring yields stronger layering (each subsystem only depends on abstract infra contracts), better resilience (Event Bus buffering), and improved security (centralized auth).

5.1.3 Internal Interfaces

In 5.2 shown complementary class-diagram sketches showing the key internal interfaces in the enhanced KSO architecture.

Description:

- IAuthService centralizes authentication and authorization: it issues and validates JWTs, and supports revocation.
- IPubSub abstracts an asynchronous event-bus (e.g. Kafka/RabbitMQ) for publishing and subscribing to domain events (clipReady, modelTrained, etc.).

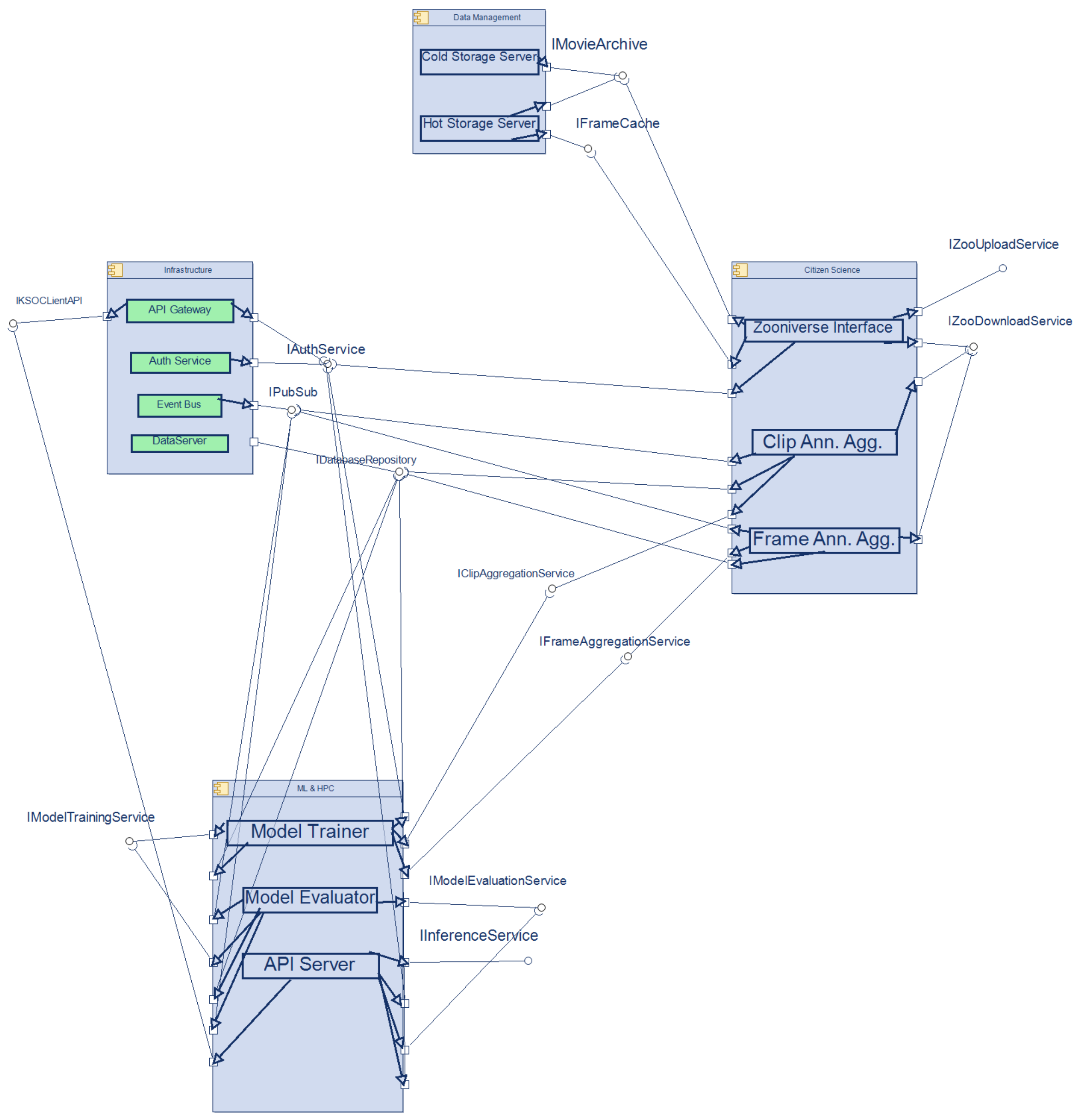

Figure 5.1: Improved KSO Component Diagram.

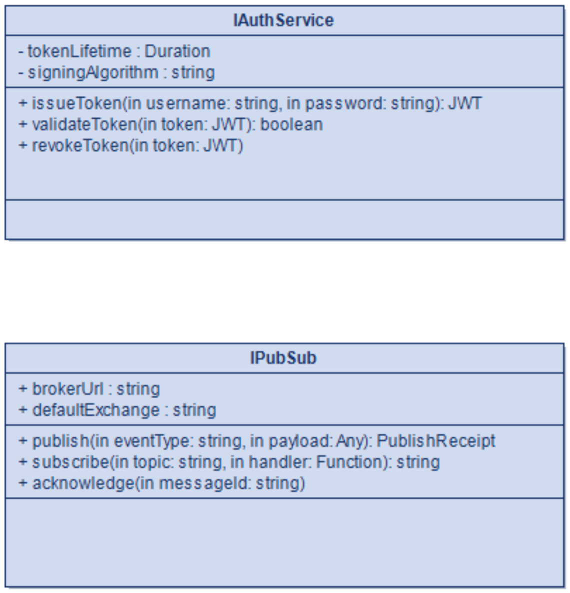
IAuthService
- tokenLifetime : Duration
- signingAlgorithm : string
+ issueToken(in username: string, in password: string): JWT
+ validateToken(in token: JWT): boolean
+ revokeToken(in token: JWT)
IPubSub
+ brokerUrl : string
+ defaultExchange : string
+ publish(in eventType: string, in payload: Any): PublishReceipt
+ subscribe(in topic: string, in handler: Function): string
+ acknowledge(in messageId: string)

Figure 5.2: Improved Internal Interface Class Diagram of KSO.

5.1.4 Interaction Patterns

This sequence 5.3 shows how, after authenticating an incoming clip-creation request, the API Gateway publishes a clipReady event. The ClipAggregationService reacts, processes classifications, persists results, then emits an annotationsReady event. Finally, the ModelTrainer picks up that event to start training—saving its run metadata to the Database Service when complete.

This section includes **1 Sequence Diagram** to show messaging sequences taking place among the system components over the **internal interfaces for your suggestions. Choose the most complex interaction to model with sequence diagram.**

5.1.5 External Interfaces

Below 5.4 are two key external-interface contracts in the enhanced KSO architecture.

- IKSOClientAPI:
  - Attributes:

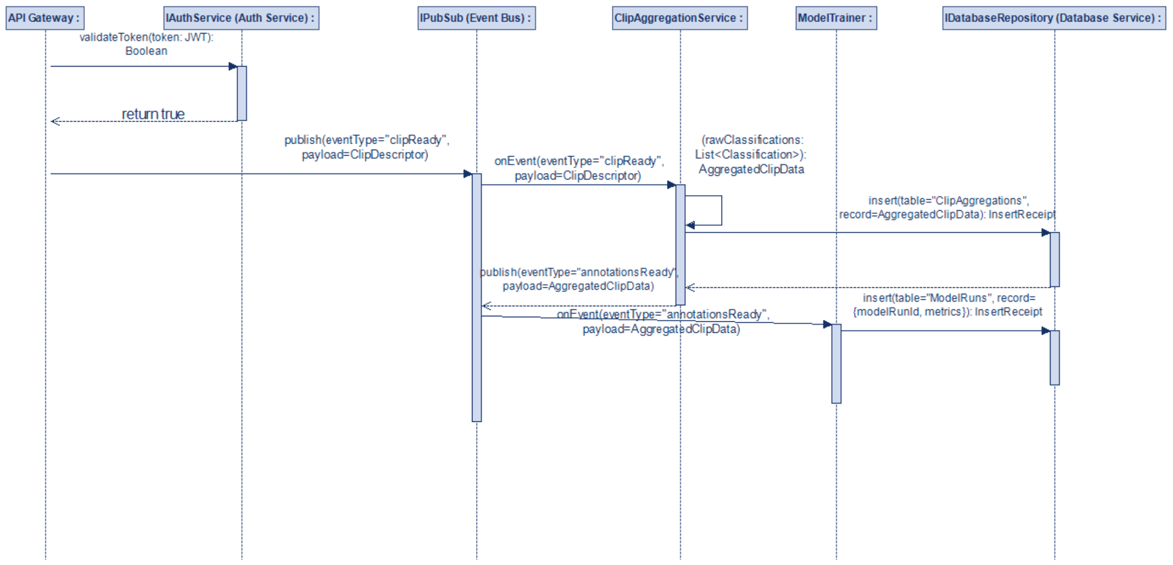
API Gateway :
IAuthService (Auth Service) :
IPubSub (Event Bus) :
ClipAggregationService :
ModelTrainer :
IDatabaseRepository (Database Service) :
validateToken(token: JWT): Boolean
return true
publish(eventType="clipReady", payload=ClipDescriptor)
onEvent(eventType="clipReady", payload=ClipDescriptor)
(rawClassifications: List<Classification>): AggregatedClipData
insert(table="ClipAggregations", record=AggregatedClipData): InsertReceipt
publish(eventType="annotationsReady", payload=AggregatedClipData)
insert(table="ModelRuns", record={modelRunId, metrics}): InsertReceipt
onEvent(eventType="annotationsReady", payload=AggregatedClipData)

Figure 5.3: Event-Driven Training Trigger Sequence Diagram

    * baseUrl: the root URI clients use for all calls.
    * apiVersion: e.g. ”v1” to support versioning.
    * timeoutMs: default request timeout.
  - Operations
    * submitClipRequest(...) and submitFrameRequest(...) initiate media processing jobs.
    * getClipAggregation(...) / getFrameAggregation(...) return consensus labels once ready.
    * getModelStatus(...) lets clients poll training progress.
    * listAvailableModels() returns descriptors of supported inference models.

- IZooniverseREST
  - Attributes:
    * endpointUrl: base URL of the Zooniverse API.
    * apiKey: credential for authenticating requests.
    * maxRetries: how many times to retry transient failures.

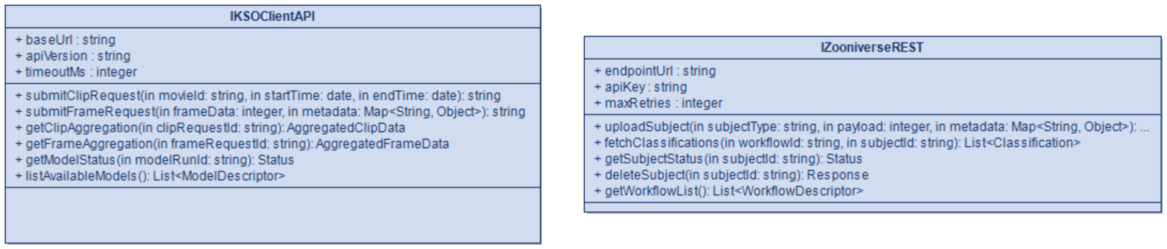
IKSOClientAPI
+ baseUrl : string
+ apiVersion : string
+ timeoutMs : integer
+ submitClipRequest(in movieId: string, in startTime: date, in endTime: date): string
+ submitFrameRequest(in frameData: integer, in metadata: Map<String, Object>): string
+ getClipAggregation(in clipRequestId: string): AggregatedClipData
+ getFrameAggregation(in frameRequestId: string): AggregatedFrameData
+ getModelStatus(in modelRunId: string): Status
+ listAvailableModels(): List<ModelDescriptor>
IZooniverseREST
+ endpointUrl : string
+ apiKey : string
+ maxRetries : integer
+ uploadSubject(in subjectType: string, in payload: integer, in metadata: Map<String, Object>): ...
+ fetchClassifications(in workflowId: string, in subjectId: string): List<Classification>
+ getSubjectStatus(in subjectId: string): Status
+ deleteSubject(in subjectId: string): Response
+ getWorkflowList(): List<WorkflowDescriptor>

Figure 5.4: External Interfaces Class Diagram for Improved System

- Operations
  - uploadSubject(...) posts new clips or frames for classification.
  - fetchClassifications(...) retrieves raw user annotations.
  - getSubjectStatus(...) polls processing state.
  - deleteSubject(...) removes obsolete subjects.
  - getWorkflowList() discovers available workflows and versions.

This section should include an **External Interfaces Class Diagram for your suggestions**. Descriptions of the operations given in the external interfaces class diagram should also be given. **You should aim for 2 external interfaces.**

5.1.6 Interaction scenarios

Below 5.5 is a proposed activity diagram outline showing the most complex end-to-end workflow over KSO's external interfaces (IKSOClientAPI and IZooniverseREST). Model it in Modelio using ActivityPartitions (swimlanes) for the Client, API Gateway, and Zooniverse API.

Structure & Notes:

Swimlanes (ActivityPartitions):

Client (consumer of IKSOClientAPI)

API Gateway (implements IKSOClientAPI, calls IAuthService, internal services, and IZooniverseREST)

Zooniverse API (external, implements IZooniverseREST)

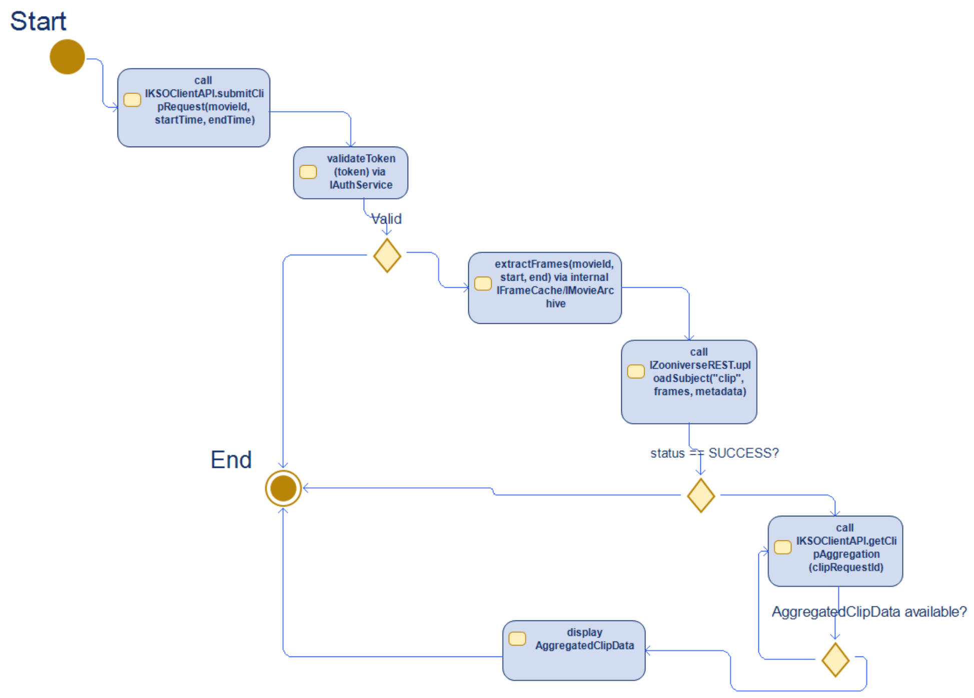
Start
call IKSOClientAPI.submitClipRequest(movieId, startTime, endTime)
validateToken (token) via IAuthService
Valid
extractFrames(movieId, start, end) via internal IFrameCache/IMovieArchive
call IZooniverseREST.uploadSubject("clip", frames, metadata)
status == SUCCESS?
End
call IKSOClientAPI.getClipAggregation (clipRequestId)
AggregatedClipData available?
display AggregatedClipData

Figure 5.5: Activity Diagram External Interaction Scenario: Clip Submission & Aggregation

Key Actions:

submitClipRequest: issues the initial job and returns a request ID.

validateToken: gates access via IAuthService.

uploadSubject: hands off frames to Zooniverse.

getClipAggregation: client polls until consensus labels are ready.

Decision Nodes:

Authentication check (valid/invalid token)

Upload success (subject created or error)

Aggregation readiness (looping until data available)

Loop:

Polling on getClipAggregation until the AggregatedClipData is non-null.

5.2 Information View

Refer to *R&W Chapter 18.*

5.2.1 Stakeholders' Uses of This View

This view helps people working with the system's data — like backend developers, DevOps teams, and database admins. The new structure helps manage events and API traffic better by introducing tables like EventLog and ClipProcessingRequest. Developers can track what happened in the system, while admins can monitor how requests move through the pipeline. This setup also makes it easier to debug when something goes wrong or when performance issues pop up.

5.2.2 Database Class Diagram

Below 5.6 is an outline of the key persistence classes in the enhanced KSO database schema.

Database Class Diagram involving the key database or main memory objects **for your suggestions.** Complete with relevant associations. Descriptions of the non-obvious names (for classes, attributes, operations) should also be given.

5.2.3 Operations on Data

Descriptions of the operations are given in the database class diagram. These operations may deal with the storage and handling of information regarding the application domain (ocean floor observation) entities.

Operations for your suggestions should be listed in tabular form or as bullet items.

These usually include CRUD (Create Read Update Delete) operations.

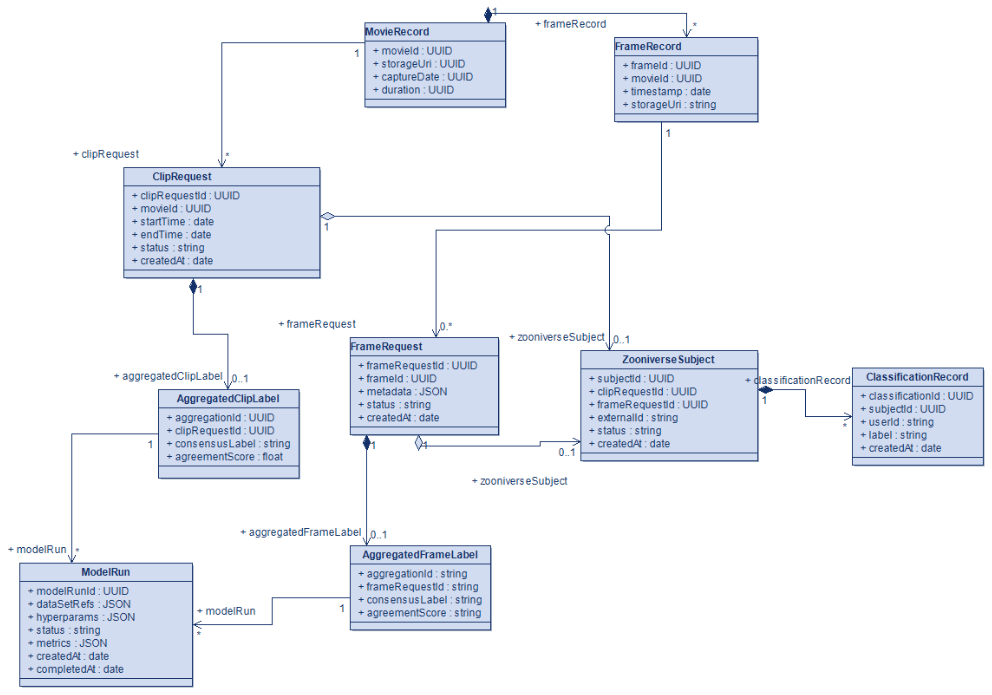

Figure 5.6: Database Class Diagram of Improved System

5.3 Development View

Refer to *R&W Chapter 20.*

5.3.1 Stakeholders' Uses of This View

This development view is helpful for developers, testers, and anyone setting up CI/CD pipelines. It shows how we split the code into separate modules like authentication, event handling, and core functionality. That makes it easier for different people or teams to work on separate parts of the system at the same time without interfering with each other. It also helps testing since you can test small packages by themselves.

5.3.2 Package Diagram

Package Contents 5.7

- **Infrastructure**

  - **API_Gateway**: external request entry-point; implements `IKSOClientAPI`.
  - **Auth_Service**: issues/validates tokens (`IAuthService`).
  - **Event_Bus**: pub/sub for domain events (`IPubSub`).
  - **Database_Service**: unified data access (`IDatabaseRepository`).

- **DataManagement**
  - **ColdStorage**: long-term movie archive (`IMovieArchive`).
  - **HotStorage**: fast frame cache (`IFrameCache`).

- **CitizenScience**
  - **Zooniverse_Interface**: pushes/pulls subjects (`IZooniverseREST`); requires Auth, Storage, and publishes `clipReady`.
  - **Clip_Aggregation**: builds clip consensus; subscribes to events, writes to DB.
  - **Frame_Aggregation**: builds frame consensus similarly.

- **ML_HPC**
  - **Model_Training**: triggers training on aggregated data; subscribes to `annotationsReady`, writes model runs.
  - **Model_Evaluation**: computes metrics on completed runs.
  - **Inference_API**: serves real-time predictions; enforces Auth and reads from DB/Evaluation.

Key Dependencies

- **API_Gateway** → **Auth_Service** (authenticate all calls)
- **Zooniverse_Interface** → **ColdStorage**, **HotStorage**, **Auth_Service**, **Event_Bus**
- **Clip_Aggregation**, **Frame_Aggregation** → **Zooniverse_Interface**, **Database_Service**, **Event_Bus**

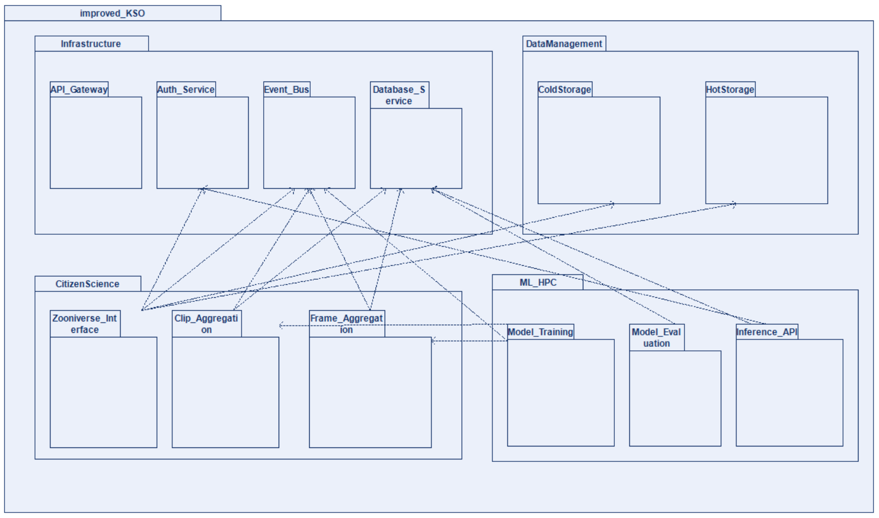

Figure 5.7: Package Diagram of Improved KSO

- **Model_Training → Clip_Aggregation**, **Frame_Aggregation**, **Database_Service**, **Event_Bus**
- **Model_Evaluation → Model_Training**, **Database_Service**
- **Inference_API → Auth_Service**, **Model_Evaluation**, **Database_Service**

This section should include a **Package Diagram** showing the higher-level software modules and their dependencies **for your suggestions**. Package decomposition hierarchy must involve 3 levels, namely, (level 0) the system (KSO), (level 1) subsystems of KSO, and (level 2) sub-subsystem levels. **Briefly explain each package's contents and show some dependencies between the packages.**

5.4 Deployment View

Refer to *R&W Chapter 21*.

5.4.1 Stakeholders' Uses of This View

This view is mostly useful for cloud admins, DevOps engineers, and people monitoring system performance. It shows where the new services like the API Gateway and Event

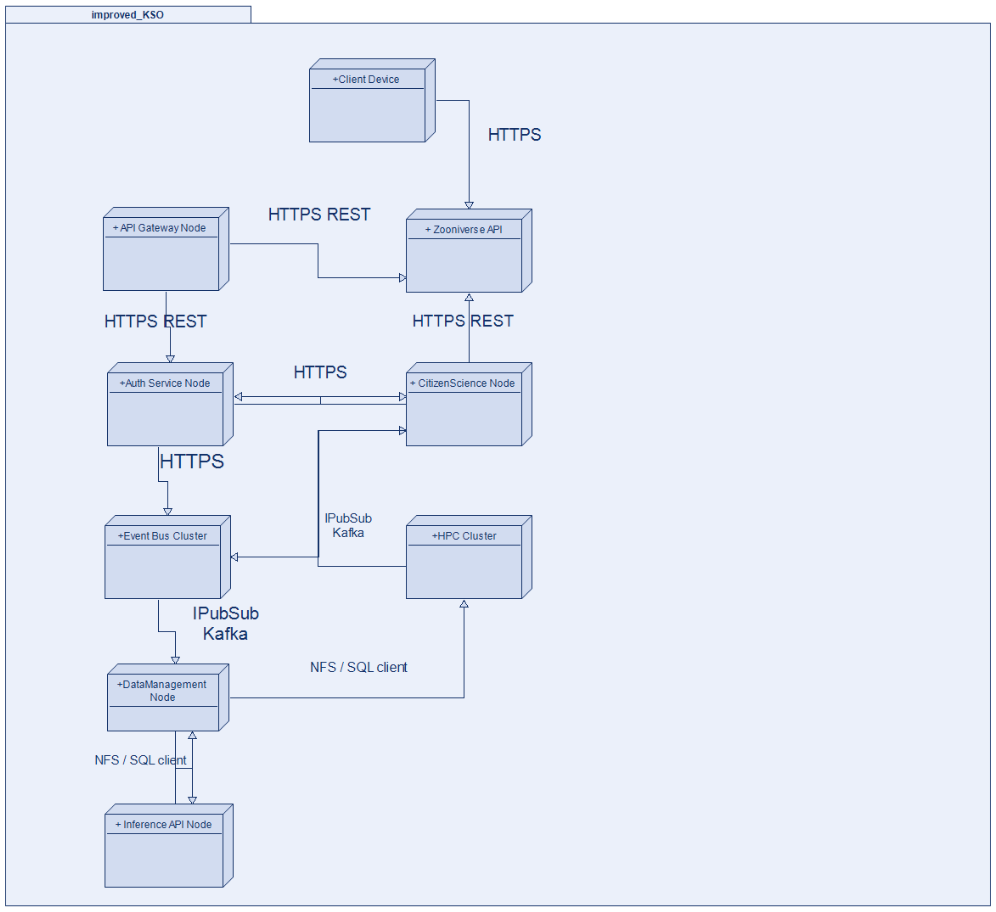

Figure 5.8: Improved KSO Deployment Diagram

Bus are deployed and how they interact. Knowing this helps allocate the right resources and scale the system if usage grows. It also helps set up alerts and logs so that problems can be caught early.

5.4.2 Deployment Diagram

Below 5.8 is a proposed deployment topology for the improved KSO system. Each Node hosts one or more Components (or Services), and the communication paths between them are annotated with protocols.

5.4.2.1 Node & Component Allocation

1. **Client Device** Runs the web UI (JavaScript) that calls `IKSOClientAPI` over HTTPS.
2. **API Gateway Node**
    - Hosts the **API Gateway Service** implementing `IKSOClientAPI`.
    - Routes and rate-limits requests, and forwards them to downstream services.
3. **Auth Service Node**
    - Runs the **Auth Service** (`IAuthService`), issuing and validating JWTs.
    - Communicates via HTTPS with the API Gateway and other service nodes.
4. **CitizenScience Node**
    - Hosts **Zooniverse Interface**, **Clip Aggregator**, and **Frame Aggregator** services.
    - Subscribes/publishes to the **Event Bus** using AMQP or Kafka.
    - Calls the Auth Service for permission checks and the Database Service for persistence.
5. **Event Bus Cluster**
    - A distributed message broker (e.g. RabbitMQ or Kafka).
    - Implements `IPubSub` for asynchronous event propagation (`clipReady`, `annotationsReady`, `modelTrained`).
6. **DataManagement Node**
    - **Cold Storage Server**: S3-compatible object store for raw movies (`IMovieArchive`).
    - **Hot Storage Server**: Redis cache for frames (`IFrameCache`).
    - **Database Service**: SQL cluster (PostgreSQL/MySQL) providing `IDatabaseRepository`.

7. **HPC Cluster**

   - Contains GPU-enabled compute nodes or Kubernetes pods for **Model Trainer** and **Model Evaluator**.
   - Communicates with the Event Bus to receive triggers and the Database Service to read/write model records.

8. **Inference API Node**

   - Hosts the **API Server** exposing `IInferenceService`.
   - Authenticates via the Auth Service, reads model metadata from the Database Service, and may fetch model artifacts from Cold Storage or a model registry.

9. **Zooniverse API (External)**

   - Third-party platform for subject upload/download, implementing `IZooniverseREST`.
   - Communicates over HTTPS with the Zooniverse Interface on the Citizen-Science node.

5.4.2.2 Communication Protocols

- **HTTPS REST** (TLS) between:
  - Client API Gateway
  - API Gateway Auth Service, CitizenScience, Inference API
  - CitizenScience Zooniverse API
- **AMQP/Kafka** between all services and the **Event Bus**
- **Redis protocol** (TCP) between services and the Hot Storage Server
- **S3-compatible REST** (HTTPS) between services and the Cold Storage Server
- **SQL over TCP/NFS** between services and the Database Service
- **Job Scheduler API** (HTTPS) between HPC compute nodes and orchestrator (optional)

5.5 Design Rationale

* **Functional View**

We introduced an **API Gateway** and **Event Bus** to decouple the core subsystems (Data Management, Citizen Science, ML & HPC). This separation via well-defined interfaces (`IKSOClientAPI`, `IPubSub`) enables asynchronous workflows, allows each subsystem to evolve independently, and improves overall system resilience by buffering spikes in load. * **Information View**

By replacing direct database access with a single **Database Service** abstraction (`IDatabaseRepository`), we hide the concrete storage technology (SQLite vs. SQL cluster). This design promotes maintainability and portability, since swapping or scaling the underlying database requires no changes to the domain-logic components. * **Development View**

The refined package decomposition enforces a strict layering: **Infrastructure** → **Data Management** → **Citizen Science** → **ML & HPC**. Dependencies flow only "downward" to abstract service contracts, enabling parallel development across teams and simplifying unit testing through easy mocking of lower-level packages. * **Deployment View**

Allocating components to specialized nodes—e.g., GPU-enabled pods in the **HPC Cluster** for training, dedicated **Inference API** servers for real-time predictions, and separate **Auth** and **API Gateway** nodes—ensures resource isolation, security zoning, and targeted autoscaling. This design maximizes performance and fault isolation in production.

State **one design rationale** specifically referring to each view you have presented for **your suggestions.**