# Distributed Imaging Services Design Document

Hamid Fathi
sahamidfathi@gmail.com

## 1 Overview

The Distributed Imaging Services project is a high-throughput, distributed software system designed to simulate the data pipeline of an autonomous underwater vehicle (AUV). The system captures raw imagery, processes it for feature extraction (SIFT) using parallel computing, and persists the data for analysis.

The solution is engineered using modern C++ (C++17), employing a microservices-oriented architecture over ZeroMQ (ZMQ). Key design goals include loose coupling, fault tolerance, and the ability to handle high-latency Computer Vision tasks without blocking the data acquisition pipeline.

## 2 System Architecture

The system utilizes a Linear Pipeline Architecture based on the Publish-Subscribe (Pub/Sub) messaging pattern. This decouples the data producers from data consumers, satisfying the critical requirement that applications must be able to start in any order and operate independently.

### 2.1 Component Topology

**App 1: Image Generator (Publisher)**

- **Role:** Simulates a hardware driver for a high-speed camera.
- **Responsibility:** Dynamic data acquisition and serialization.
- **Constraint:** Must handle dynamic changes to the data source (files added/removed) at runtime.

**App 2: Feature Extractor (Subscriber/Publisher)**

- **Role:** The computational engine.
- **Responsibility:** Deserialization, SIFT (Scale-Invariant Feature Transform) processing, and data enrichment.
- **Constraint:** CPU-bound. Requires a concurrent design to match the throughput of the Generator.

**App 3: Data Logger (Subscriber)**

- **Role:** The persistence layer.
- **Responsibility:** Transactional storage of images and metadata.
- **Constraint:** I/O-bound.

# 3 Concurrency Model (Multithreading Strategy)

The most critical performance bottleneck in the system is the Feature Extractor. While image acquisition (App 1) and logging (App 3) are fast operations, SIFT feature detection can be computationally expensive.

To prevent the processing step from blocking the entire pipeline, a Producer-Consumer Thread Pool Pattern has been implemented.

## 3.1 Thread Pool Implementation

**Main Thread (Producer).** The main thread listens to the ZMQ Subscriber socket. Its only job is to receive raw bytes and push them into a queue. This ensures the network buffer never overflows.

**Worker Threads (Consumers).** A pool of `N` threads (scaled to `std::thread::hardware_concurrency()`) continuously pull tasks from the queue.

**Isolation.** Each worker owns its own `cv::SIFT` instance and is purely CPU-bound. Processed results are pushed into a second `SafeQueue<ProcessedTask>`, which is consumed by a dedicated *sender thread*. The sender thread is the only thread that owns the ZMQ Publisher socket. This design avoids lock contention on the network layer and cleanly separates CPU-heavy feature extraction from network I/O.

## 3.2 Synchronization Primitives

To ensure thread safety, a custom `SafeQueue<T>` monitor was implemented using:

- `std::mutex`: To protect the critical section (the internal STL queue).
- `std::condition_variable`: To implement efficient waiting (blocking) when the queue is empty, preventing busy-wait CPU cycles.
- `std::lock_guard` & `std::unique_lock`: Employed for RAII-compliant locking to prevent deadlocks in case of exceptions.

# 4 Communication Layer & Protocols

## 4.1 Inter-Process Communication (IPC)

ZeroMQ (ZMQ) was selected as the transport layer.

- **Pattern:** PUB/SUB (Publish/Subscribe).

- **Rationale:** ZMQ provides message framing (preserving message boundaries) and automatic reconnection logic. Unlike TCP sockets, ZMQ handles the connection logic asynchronously, satisfying the "Any-Order Start" requirement.

### 4.2 Message Protocol

Data is encapsulated in multipart ZMQ messages. This allows separating metadata from binary payloads without parsing overhead.

**Input Stream (Generator → Extractor).**

- Frame 1: Filename (UTF-8 String)
- Frame 2: Image Blob (Compressed JPEG/PNG bytes)

**Output Stream (Extractor → Logger).**

- Frame 1: Filename (UTF-8 String)
- Frame 2: Image Blob (Pass-through from input)
- Frame 3: Keypoints Blob (Custom Binary Serialization)

## 5 Data Persistence & Serialization

### 5.1 Serialization

While libraries like Protobuf were considered, a custom binary serializer was implemented for `cv::KeyPoint` data.

- **Method:** Flat memory copy (`std::memcpy`) of the struct fields into a `std::vector<char>`.
- **Reasoning:** Minimizes dependencies and eliminates serialization overhead for this specific, high-frequency data structure.

### 5.2 Database Schema

SQLite3 is used for embedded, serverless persistence.

```
CREATE TABLE processed_images (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    filename TEXT NOT NULL,
    timestamp DATETIME DEFAULT CURRENT_TIMESTAMP,
    image_blob BLOB,    -- Raw binary image data
    keypoints_blob BLOB -- Serialized feature vectors
);
```

# 6 Design Patterns Summary

- **Observer Pattern:** The fundamental architecture (Pub/Sub), where applications observe streams of data.
- **Producer-Consumer:** Used inside the Extractor to bridge the gap between fast network I/O and slow CPU processing.
- **RAII (Resource Acquisition Is Initialization):** Strictly followed for resource management (sockets, file handles, database connections, and mutex locks).

# 7 Trade-off Analysis & Future Roadmap

## 7.1 Reliability vs. Latency

**Current:** The system favors low latency ("Best Effort"). If the Logger crashes, data in transit is dropped.

**Alternative:** To guarantee delivery, we could switch to ZMQ PUSH/PULL or a broker-based system (e.g., RabbitMQ). This was rejected to keep the system brokerless and real-time.

## 7.2 DDS vs. ZMQ

**Decision:** ZMQ was chosen for rapid development and its lightweight footprint.

**Path to Production:** For a deployed ROV system, DDS (Data Distribution Service) is superior due to its QoS (Quality of Service) policies.

## 7.3 Dynamic Configuration

**Current:** The Generator polls the filesystem to detect new files.

**Optimization:** In a production Linux environment, `inotify` (kernel-level filesystem monitoring) would be more efficient than polling, reducing idle CPU usage.

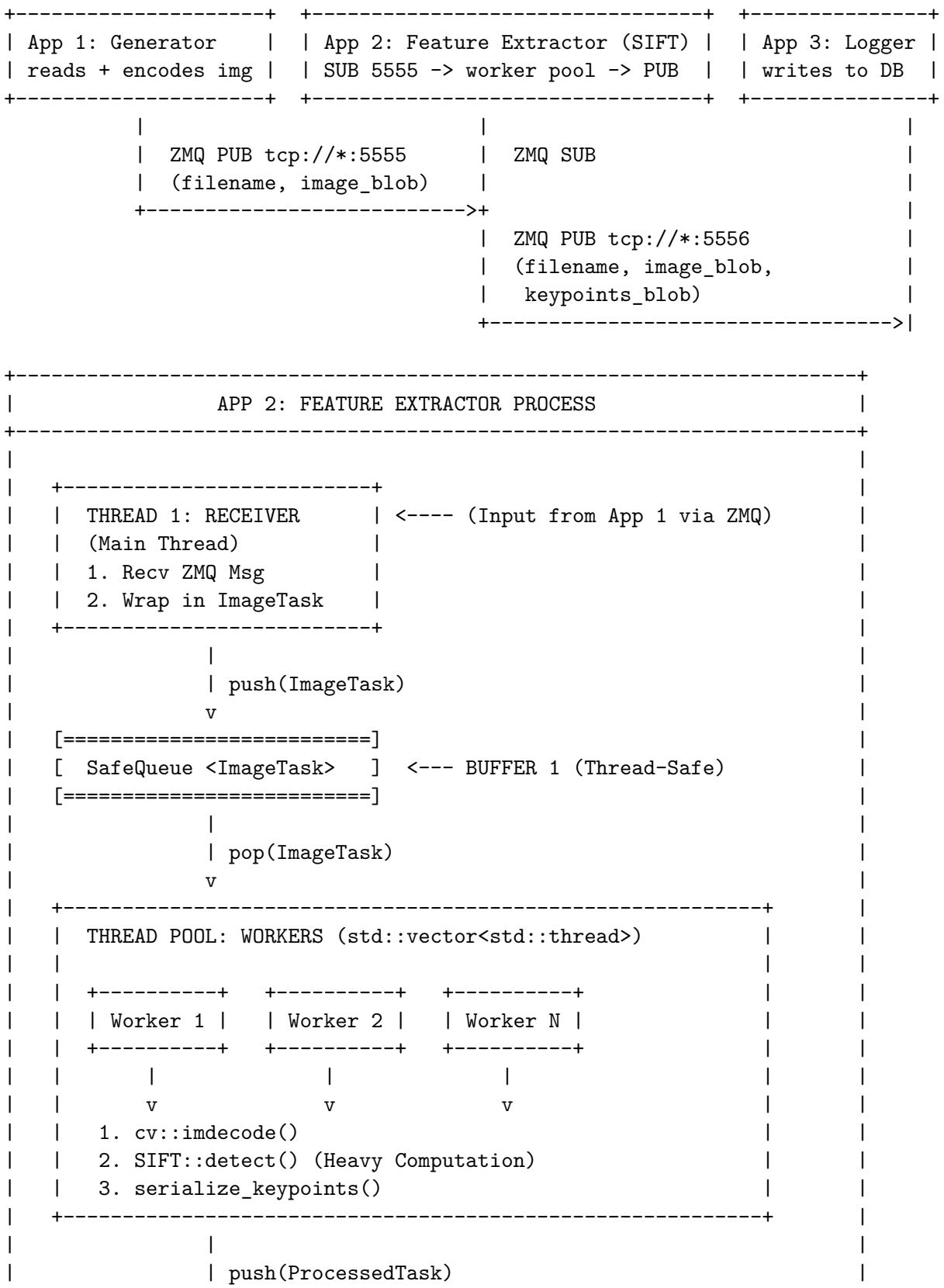
# 8 Build & Deployment

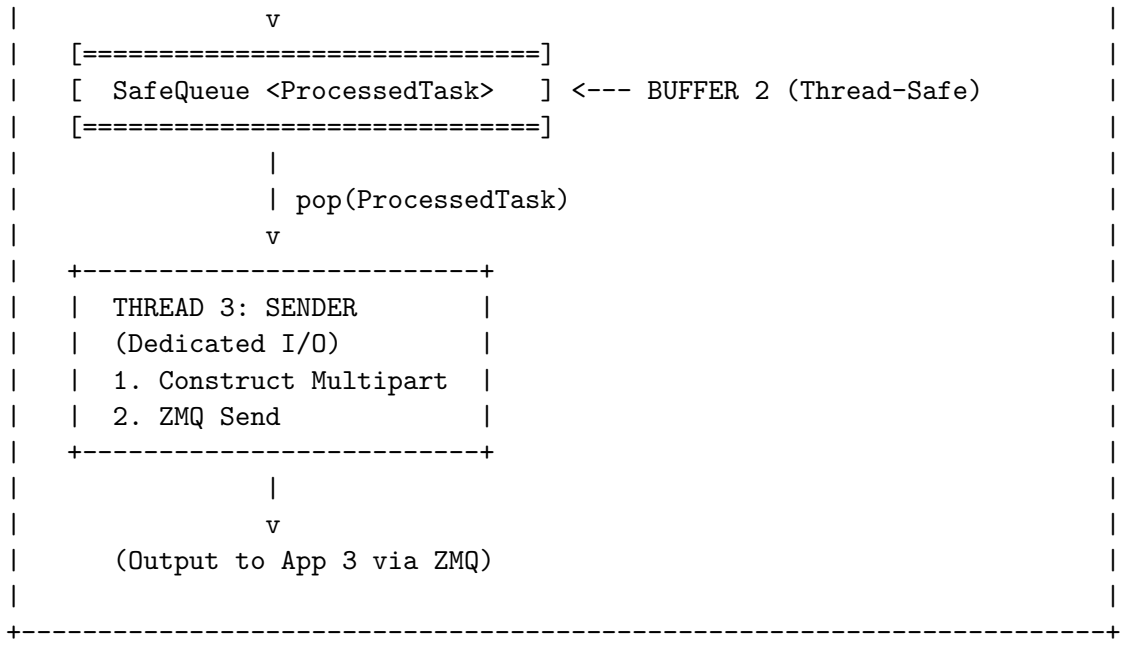
The system is built using CMake and targets Linux, making it straightforward to containerize (e.g., with Docker) for deployment.

```
# Build
mkdir build && cd build
cmake ..
make -j$(nproc)

# Run each component in a separate terminal/process
./data_logger
./feature_extractor
./image_generator ../images   # path to the image directory
```

## 9 System Overview Schematic




v
[==============================]
[ SafeQueue <ProcessedTask> ] <--- BUFFER 2 (Thread-Safe)
[==============================]
| pop(ProcessedTask)
v
THREAD 3: SENDER
(Dedicated I/O)
1. Construct Multipart
2. ZMQ Send
v
(Output to App 3 via ZMQ)