Monash University Information Technology
FIT3162 - Computer Science Project 2

Test Report

Stream Processing Architecture using Apache Kafka for Analytics and Visualisation On A Dashboard

Khai Fung, Lim
29297311
klim0022@student.monash.edu

Yi Ping, Ho
29352258
yhoo0007@student.monash.edu

Chiu Gin, Chong
28842022
ccho0024@student.monash.edu

Fernando Ng
28737377
fern0002@student.monash.edu

Semester 1, 2020

Team 1C
Supervisor: Dr. Vishnu Monn Baskaran
WORD COUNT: 4183

Contents

1 Introduction

Testing the software is an imperative part of a software development project as it is necessary in ensuring that code errors and bugs are minimized and the deliverables of the project behaves as intended. In this test report, we will be documenting the test plan carried out by the team for our Active Video Surveillance project in different stages accordingly.

The main parts of our software that were tested are the front-end module which is our dashboard and the back-end module of our software that includes the Kafka cluster, MongoDB database and Hadoop Distributed File System (HDFS). Our tests incorporate both manual testing to check all essential features of the software and automated testing to determine if the code performs as expected. Since our software code is written in the Java language, the automated tests are executed using the JUnit framework from Java SE Development Kit. These tests are mainly used for functional testing of our code where the functions declared in our code are tested to guarantee their performance and correct output. For all of our testing, we are constantly clearing the MongoDB database, HDFS database, Kafka and Zookeeper logs to ensure a new testing environment for each test.

User Story ID	**User Acceptance Criteria ID**	**Test Case ID**
1	1.1	1
1	1.2	21-25
2	2.1	21-25
2	2.2	3
3	3.1	2
3	3.2	3
4	4.1	1
4	4.2	4
5	5.1	5
6	6.1	20
7	7.1	17
7	7.2	19

Table 1: User Acceptance Criteria and Test Matching

2 Unit Testing and Test Driven Development

2.1 Front-end Module

The front-end part of our program refers to the User Interface dashboard that will display the processed videos with a list of camera feed. It also shows relevant information like graphs presenting the number of faces detected in the video over time as well as the analytics data like latency and frame rate discovered from processing the video. Our dashboard is created using Java Swing, a graphical user interface toolkit for Java. The dashboard is instantiated and displayed when the VideoStreamDashboard.java file is called and ran in the terminal after establishing a connection with Kafka, Zookeeper, HDFS and our MongoDB database. This part of our software is mainly tested using

manual testing to ensure the features of the dashboard is functioning properly in correspondence to the back-end part of our system. Each test is tested manually according to the steps specified in the test procedures of each test. The tests are performed thoroughly to ensure the dashboard works with ease along with the system so users will not get any trouble in using the dashboard.

2.1.1 Functionality Testing

1. **Test Case 1** System displays and publish processed frames as videos onto dashboard.

   Description

   Dashboard is playing processed video frames with facial detection feature when input video is fed into system.

   Test procedure

   (a) Configure system parameters accordingly. (Refer to End User Guide)

   (b) Run Kafka server with Zookeeper, HDFS and MongoDB.

   (c) Compile code and run the dashboard with VideoStreamProcessor.java and VideoStreamCollector.java to display processed videos and data analytics.

   (d) Verify that video is displayed on the top left panel of the dashboard and shows facial detection feature.

   Expected outcome

   Video feed is displayed on dashboard with facial detection feature.

   Actual outcome

   Video feed is displayed on dashboard with facial detection feature.

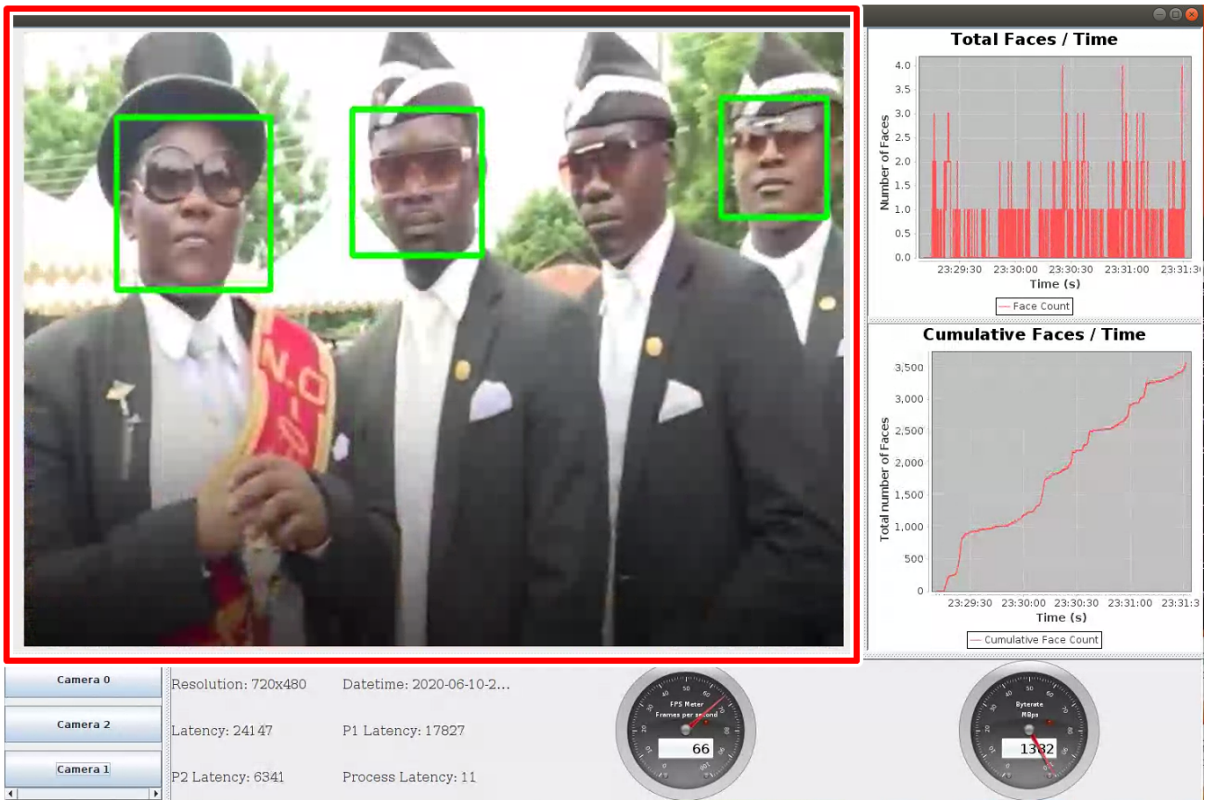


Figure 1: Dashboard with video feed.

2. **Test Case 2** System shows analytical data to dashboard.

   Description

   Dashboard is displaying all the required analytical data as declared in VideoStream-Dashboard.java file.

   Test procedure

   (a) Configure system parameters accordingly. (Refer to End User Guide)

   (b) Run Kafka server with Zookeeper, HDFS and MongoDB.

   (c) Compile code and run the dashboard with VideoStreamProcessor.java and VideoStreamCollector.java to display processed videos and data analytics.

   (d) Verify that all analytical data is present and displayed on dashboard.

   Outcome table

Component	Expected Outcome	Actual Outcome	Pass/Fail
Graph of number of faces detected over time	Graph displayed on dashboard	Graph displayed on dashboard	Pass
Graph of cumulative total number of faces detected over time	Graph displayed on dashboard	Graph displayed on dashboard	Pass
Resolution	Resolution displayed on dashboard	Resolution displayed on dashboard	Pass
Datetime	Datetime displayed on dashboard	Datetime displayed on dashboard	Pass
Latencies	Latencies displayed on dashboard	Latencies displayed on dashboard	Pass
Frames per second	Frames per second displayed on dashboard	Frames per second displayed on dashboard	Pass
Byterate	Byterate displayed on dashboard	Byterate displayed on dashboard	Pass

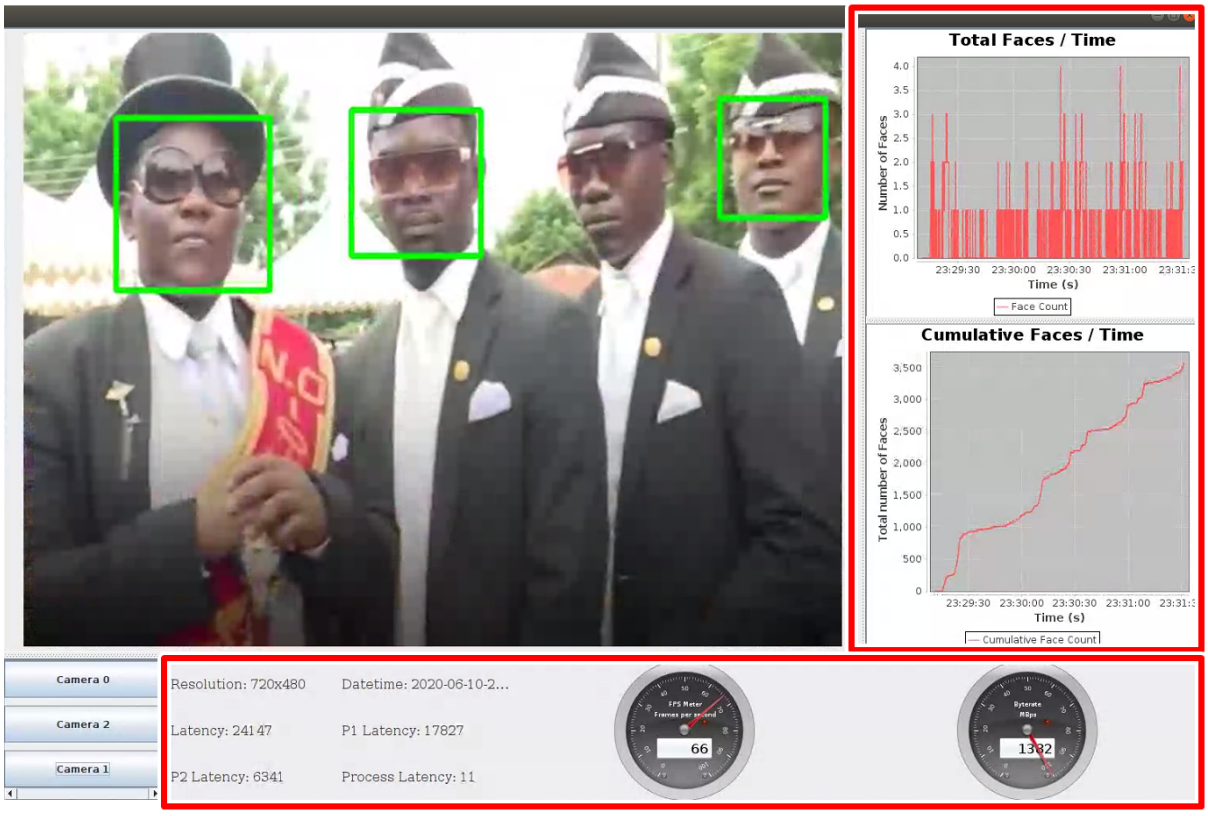


Figure 2: Dashboard with analytical data.

3. **Test Case 3** Navigating between different video feed on dashboard.

Description

Clicking on different camera buttons on the bottom left panel of the dashboard navigates to a different video.

Test procedure

(a) Configure system parameters accordingly. (Refer to End User Guide)

(b) Run Kafka server with Zookeeper, HDFS and MongoDB.

(c) Compile code and run the dashboard with VideoStreamProcessor.java and VideoStreamCollector.java to display processed videos and data analytics.

(d) Click on camera buttons on bottom left panel to switch to another video.

(e) Verify that another video is displayed on the top left panel of the dashboard.

Expected outcome

Different video feed is displayed when camera button is clicked.

Actual outcome

Different video feed is displayed when camera button is clicked.

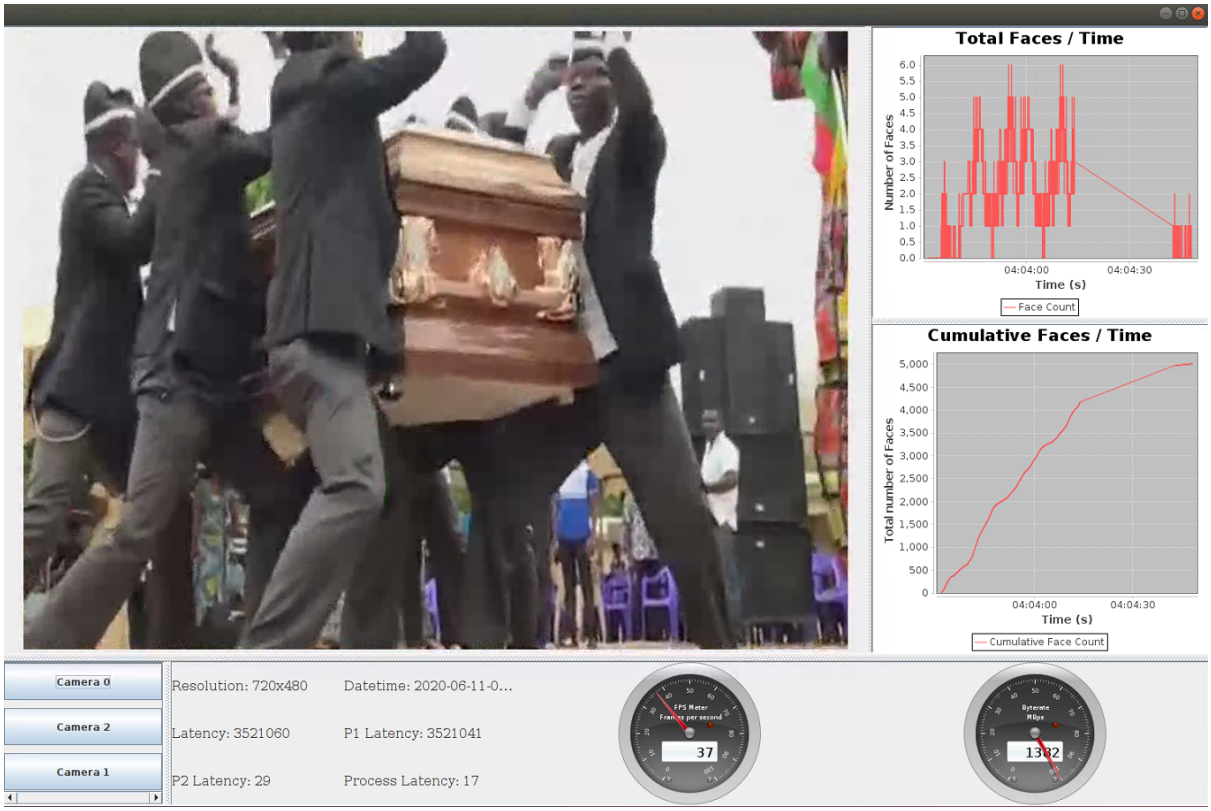


Figure 3: Video shown when camera 0 button is clicked.

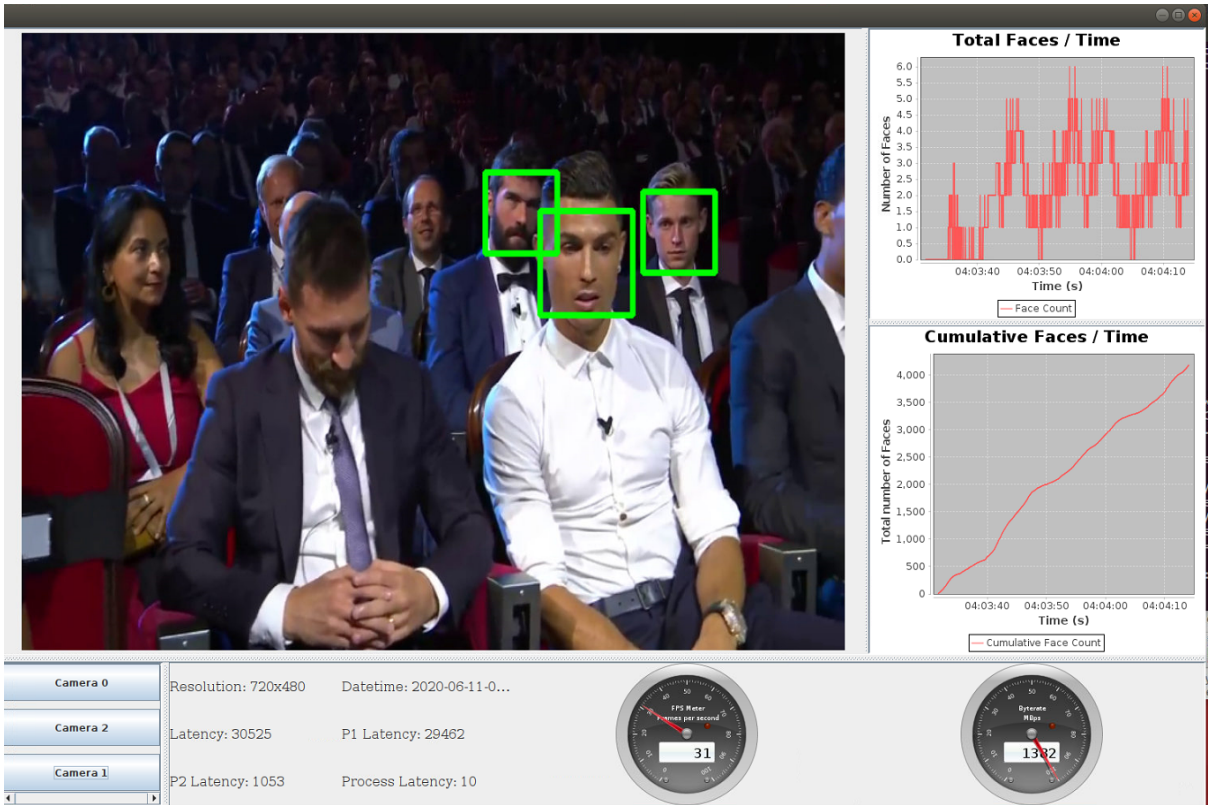


Figure 4: Video shown when camera 1 button is clicked.

4. **Test Case 4** Number of camera buttons shown on dashboard matches number of input videos fed into system.

   Description

   Ensuring that the total camera URLs declared for producer in system matches the number of camera buttons displayed on dashboard.

   Test procedure

   (a) Set camera.url in collector.properties to `camera.url=0,1,2` and ensure there are 3 camera properties file named as `0.properties`, `1.properties` and `2.properties`.

   (b) Run Kafka server with Zookeeper, HDFS and MongoDB.

   (c) Compile code and run the dashboard with VideoStreamProcessor.java and VideoStreamCollector.java to display processed videos and data analytics.

   (d) Verify that the number of buttons shown on dashboard is 3 as specified at step (a).

   Expected outcome

   The number of buttons shown on dashboard matches number of input videos fed into the system.

   Actual outcome

   The number of buttons shown on dashboard matches number of input videos fed into the system.

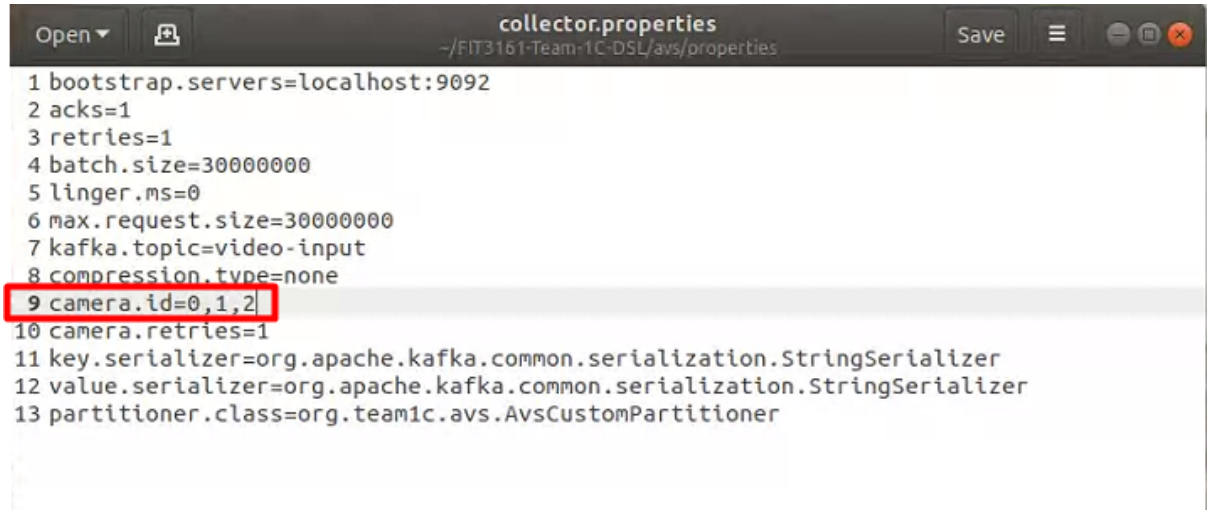


Figure 5: `camera.url` in collector.properties is set to `0,1,2`

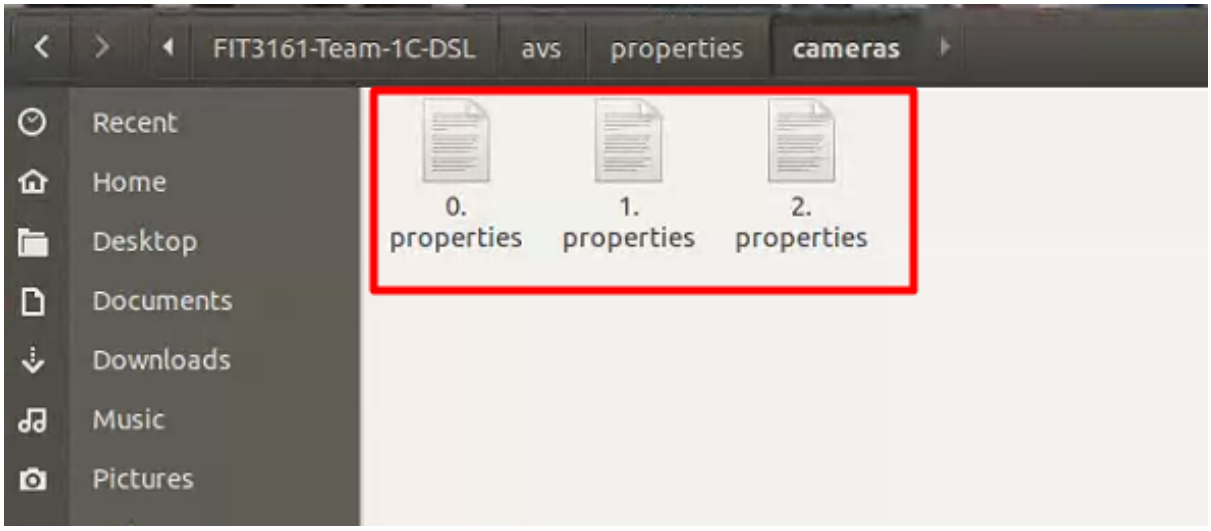


Figure 6: Camera properties files are named according to number of camera URLs.

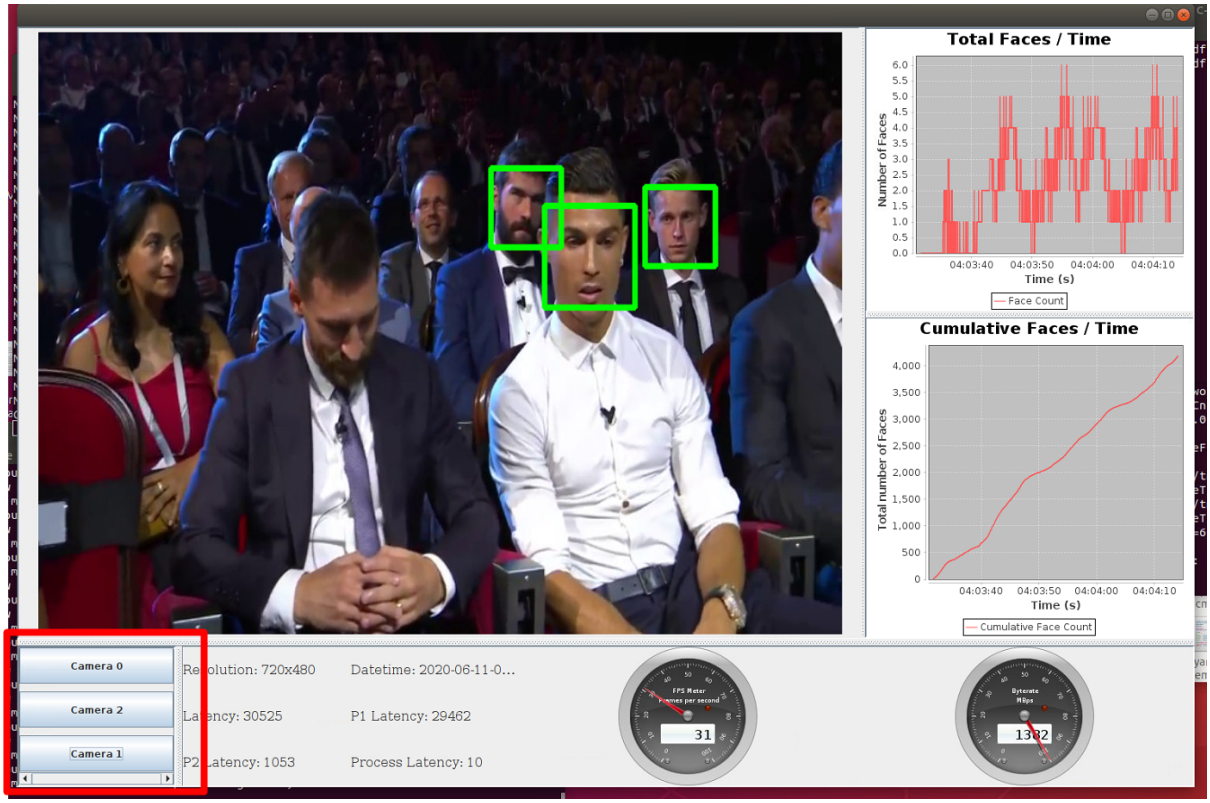


Figure 7: The number of camera buttons shown is 3.

5. **Test Case 5** Data analytics on dashboard are constantly running as input videos are being processed.

   Description

   Data analytics on dashboard shows updated values in real-time as video frames are being processed and published onto dashboard.

   Test procedure

   (a) Configure system parameters accordingly.

   (b) Run Kafka server with Zookeeper, HDFS and MongoDB.

   (c) Compile code and run the dashboard with VideoStreamProcessor.java and VideoStreamCollector.java to display processed videos and data analytics.

   (d) Verify that all analytical data displayed on dashboard are running and updated accordingly in real-time on dashboard.

   Expected outcome

   Analytics data on dashboard is running and constantly updated in real-time.

   Actual outcome

   Analytics data on dashboard is running and constantly updated in real-time.

6. **Test Case 6** Adjusting panels of dashboard.

   Description

   Panels can be adjusted according to users' preference and dashboard performs as usual with no errors.

Test procedure

(a) Configure system parameters accordingly.

(b) Run Kafka server with Zookeeper, HDFS and MongoDB.

(c) Compile code and run the dashboard with VideoStreamProcessor.java and VideoStreamCollector.java to display processed videos and data analytics.

(d) Drag and adjust panels on dashboard.

(e) Verify that dashboard still performs normally with no bugs.

Expected outcome

Dashboard with adjusted panels is running with no problem.

Actual outcome

Dashboard with adjusted panels is running with no problem.

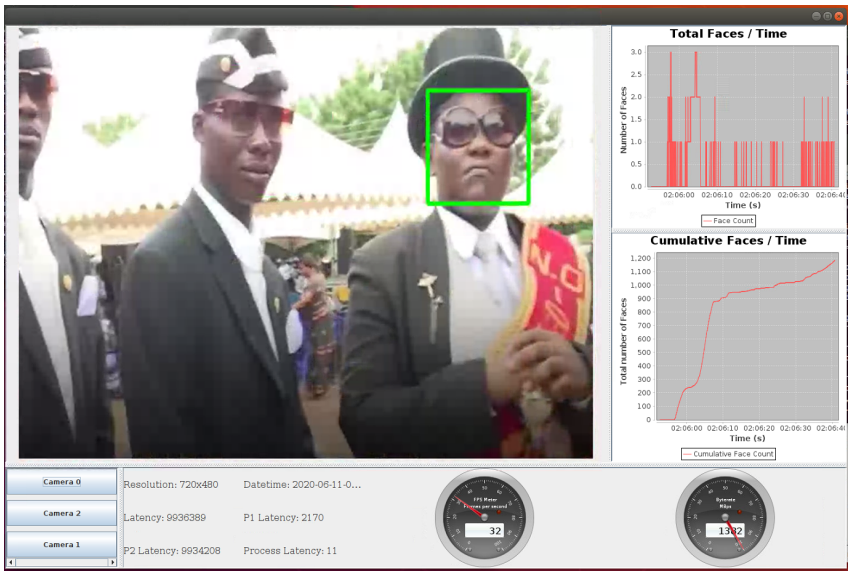


Figure 8: Dashboard before any panels adjusted.

Figure 9: Dashboard with panels adjusted. Data analytics displayed on dashboard resizes according to the panel adjusted.

7. **Test Case 7** Minimising/Maximizing dashboard

   Description

   Dashboard can be minimized and maximized according to user's preference.

   Test procedure

   (a) Configure system parameters accordingly.
   (b) Run Kafka server with Zookeeper, HDFS and MongoDB.
   (c) Compile code and run the dashboard with VideoStreamProcessor.java and VideoStreamCollector.java to display processed videos and data analytics.
   (d) Minimize the dashboard.
   (e) Verify that dashboard can be minimized and still performs normally with no errors.
   (f) Maximize the dashboard.
   (g) Verify that dashboard can be maximized and still performs normally with no errors.

   Outcome table

Test	**Pass/Fail**
Minimizing dashboard	Pass
Maximizing dashboard	Pass

2.2 Back-end Module

The back-end part of our system is where most of our architectural design of big data framework takes place. Our code is developed in one Java project with all the Java class files required for the system to run located in a single source folder in our project directory. Each Java file represents a different component required in our system namely video stream collector, video stream processor, video stream dashboard and video stream archiver and its supporting files that is discussed in the Methodology of the Final Report.

In the back-end of our system, a mixture of both manual and automated testing is done. In the aspect of automated testing, since our code is developed in the Java language, tests will be done using JUnit, an open source testing framework for Java, designed for the purpose of writing and running test cases for Java programs. The automated testing using JUnit is used to test the functionality of static methods that cannot be instantiated and are stateless. For example, most of the tests done are for static methods in Utils.java class that is a common function class where the framework repeatedly uses. These test cases are developed under one test class in the test folder of our source code.

Manual testing is done for testing separate Java class files that make up the component of our system. These tests include running each java file manually on terminal independently to ensure testing is done within the scope of the file. This manual testing process is similar to the integration testing done in Section 3 with the difference of running the java class singly without other classes to ensure the component can run independently without errors.

2.2.1 Automated Testing (JUnit)

All JUnit tests are run and tested in one Java file. The tests are done in IntelliJ IDEA's Test Runner tab. The figure below shows that all test cases for the static methods pass and the time taken to run the tests. There were a total of 15 tests.

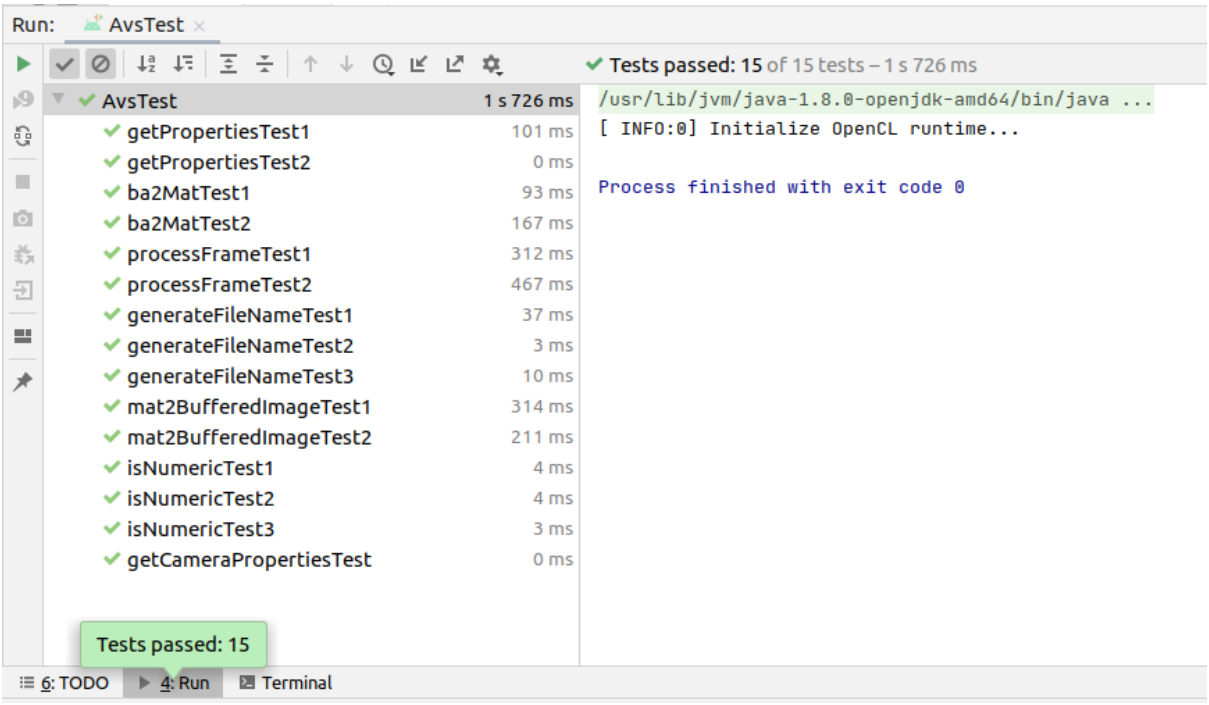


Figure 10: All 15 JUnit tests passed

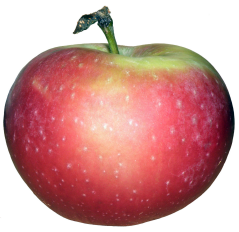
1. **Test Case 8** Test video frame processor for facial detection

   Description:

   Tests on video frame processor processFrame() static function in VideoStream-Processor class which uses Haar Feature-based Cascade Classifiers for facial detection. The function returns number of faces detected in the image. Two tests are conducted with two image frame, one with a image of 4 people which should return 4 and another an image of an apple with no human facial detected which should return 0.

   Input:

(a) Input image with faces

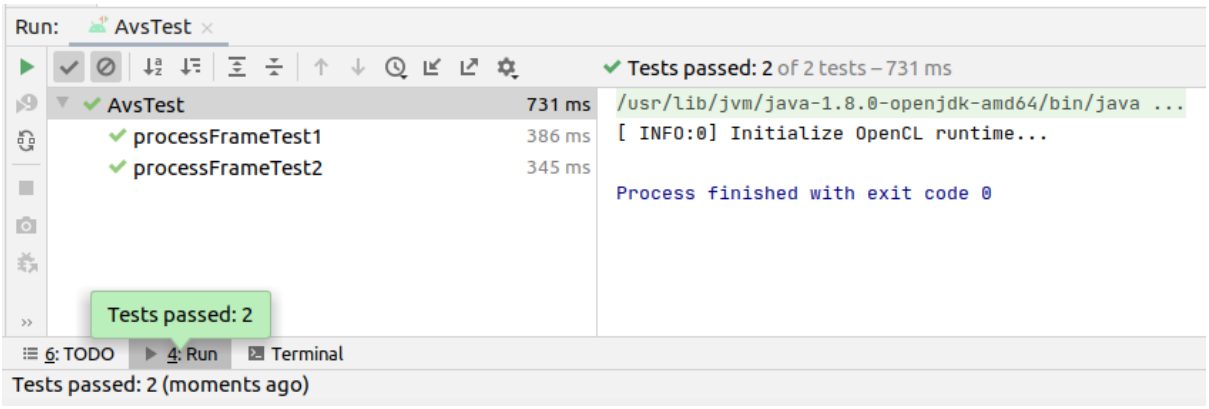


(b) Input image with no faces

   Code:

```java
//VideoStreamProcessor
@Test
public void processFrameTest1() {
    CascadeClassifier faceCascade = new CascadeClassifier();
    faceCascade.load( filename: "/home/gin/opencv/opencv-3.4/data/haarcascades/haarcascade_frontalface_alt.xml");

    Mat familyPhoto = Imgcodecs.imread( filename: "/home/gin/Desktop/FIT3161-Team-1C-master/avs/src/test/java/family.jpg");
    int expectedFaces = VideoStreamProcessor.processFrame(familyPhoto,faceCascade);
    assertEquals(expectedFaces,  actual: 4);
}

@Test
public void processFrameTest2() {
    CascadeClassifier faceCascade = new CascadeClassifier();
    faceCascade.load( filename: "/home/gin/opencv/opencv-3.4/data/haarcascades/haarcascade_frontalface_alt.xml");

    Mat applePhoto = Imgcodecs.imread( filename: "/home/gin/Desktop/FIT3161-Team-1C-master/avs/src/test/java/apple.jpg");
    int expectedFaces = VideoStreamProcessor.processFrame(applePhoto,faceCascade);
    assertEquals(expectedFaces,  actual: 0);

}
```

Result: Pass



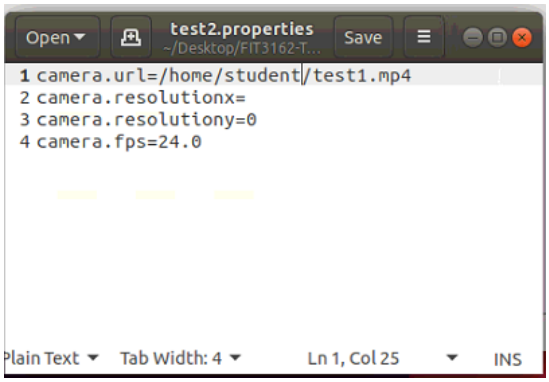
2. **Test Case 9** Test reading a property file into a Java Property object

   Description:

   Tests on getProperties() static function in Util class to ensure the Java Properties object returned has correctly read and load the properties file data. The function returns a Properties object. Two tests are conducted to check if the key and values pair in the Properties objects matches the values in original properties file.

   Input:

(a) Properties file 1

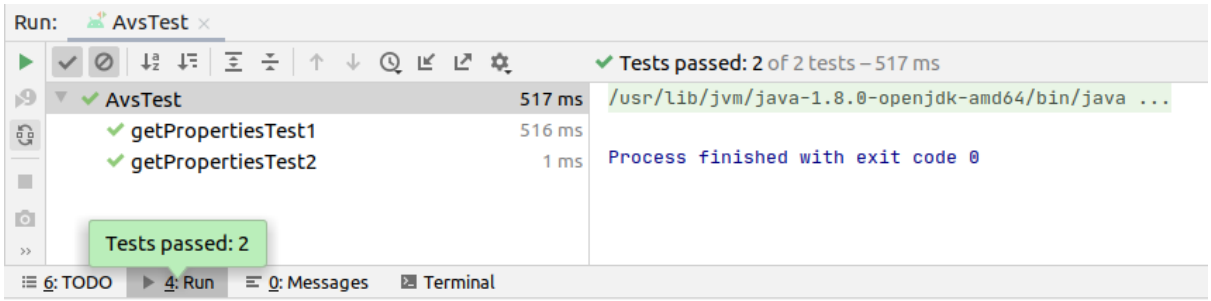


(b) Properties file 2

Code:

```
@Test
public void getPropertiesTest1() throws IOException {
    String filePath = "/home/gin/Desktop/FIT3161-Team-1C-master/avs/src/test/java/test1.properties";
    Properties actualProperties = Util.getProperties(filePath);
    assertEquals(actualProperties.getProperty("camera.resolutionx"), actual: "720");
    assertEquals(actualProperties.getProperty("camera.resolutiony"), actual: "480");
    assertNotEquals(actualProperties.getProperty("camera.fps"), actual: "");
    assertNotEquals(actualProperties.getProperty("camera.nchannels"), actual: "100");
}

@Test
public void getPropertiesTest2() throws IOException {
    String filePath = "/home/gin/Desktop/FIT3161-Team-1C-master/avs/src/test/java/test2.properties";
    Properties actualProperties = Util.getProperties(filePath);
    assertEquals(actualProperties.getProperty("camera.fps"), actual: "24.0");
    assertEquals(actualProperties.getProperty("camera.resolutionx"), actual: "");
    assertNotEquals(actualProperties.getProperty("camera.resolutiony"), actual: "00");
    assertNotEquals(actualProperties.getProperty("camera.nchannels"), actual: "");
}
```

Result: Pass



3. **Test Case 10** Test read and return a Java Property file for a given camera ID

Description:

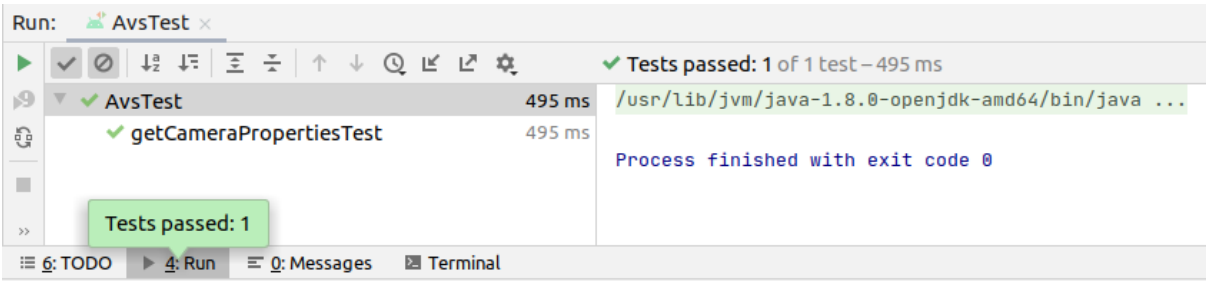
Tests on getCameraProperties() static function in Util class to ensure the Java

Properties object returned has correctly stored the key and value properties values from the camera ID properties file. Test is run to check if the key and value pairs in the returned Properties object matching the value stated in the original properties file of the camera ID.

Code:

```
@Test
public void getCameraPropertiesTest() throws IOException {
    String camID = "0";
    Properties actualProperties = Util.getCameraProperties(camID);
    assertEquals(actualProperties.getProperty("camera.resolutionx"), actual: "720");
    assertEquals(actualProperties.getProperty("camera.resolutiony"), actual: "480");
    assertNotEquals(actualProperties.getProperty("camera.fps"), actual: "");
    assertNotEquals(actualProperties.getProperty("camera.nchannels"), actual: "100");
}
```

Result: Pass



4. **Test Case 11** Test conversion of byte array to Mat object

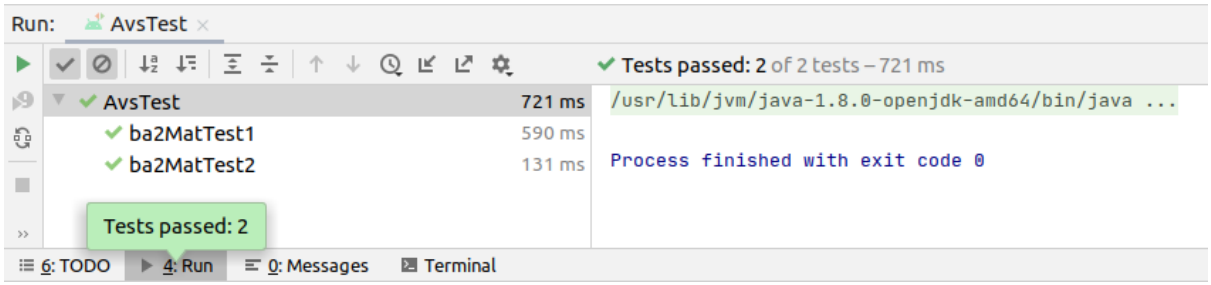
Description:

Tests on ba2Mat() static function in Util class to ensure the Mat object created from byte array has the right dimensions and type. The function returns the Mat object. Two tests are conducted on two images, one of size 1515x1600 and type CV_8UC3 and another of size 720x1280 and type CV_8UC3. The Mat objects returned are asserted to be equal to these values.

Code:

```java
@Test
public void ba2MatTest1(){
    Mat expected = Imgcodecs.imread( filename: "/home/gin/Desktop/FIT3161-Team-1C-master/avs/src/test/java/family.jpg");
    byte[] bytesArr = new byte[expected.channels() * expected.cols() * expected.rows()];
    expected.get( row: 0, col: 0,bytesArr);
    Mat actual = Util.ba2Mat(expected.rows(),expected.cols(), CvType.CV_8UC3,bytesArr);
    assertEquals(actual.rows(),  actual: 720);
    assertEquals(actual.cols(),  actual: 1280);
    assertEquals(actual.type(),  actual: 16);
}


@Test
public void ba2MatTest2(){
    Mat expected = Imgcodecs.imread( filename: "/home/gin/Desktop/FIT3161-Team-1C-master/avs/src/test/java/apple.jpg");
    byte[] bytesArr = new byte[expected.channels() * expected.cols() * expected.rows()];
    expected.get( row: 0, col: 0,bytesArr);
    Mat actual = Util.ba2Mat(expected.rows(),expected.cols(), CvType.CV_8UC3,bytesArr);
    assertEquals(actual.rows(),  actual: 1515);
    assertEquals(actual.cols(),  actual: 1600);
    assertEquals(actual.type(),  actual: 16);
}
```

Result: Pass



5. **Test Case 12** Test conversion of Mat object to BufferredImage object

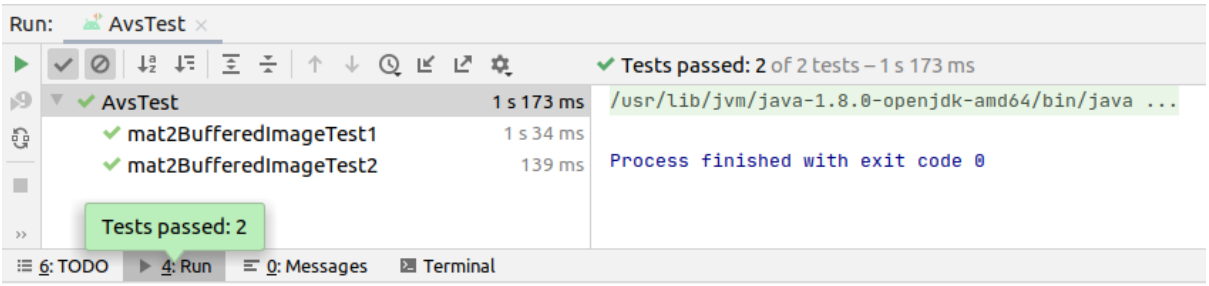
Description:

Tests on mat2BufferedImage() static function in Util class to ensure buffered image created has the right dimensions and type. The function returns a BufferedImage object. Two tests are conducted on two images, one of size 1515x1600 and type CV_8UC3 and another of size 720x1280 and type CV_8UC3. The BufferedImage objects returned are asserted to be equal to these values.

Code:

```java
@Test
public void mat2BufferedImageTest1() {
    Mat m = Imgcodecs.imread( filename: "/home/gin/Desktop/FIT3161-Team-1C-master/avs/src/test/java/family.jpg");
    BufferedImage actual = Util.Mat2BufferedImage(m);
    assertEquals(actual.getHeight(), actual: 720);
    assertEquals(actual.getWidth(), actual: 1280);
    assertEquals(actual.getType(), actual: 5);

}

@Test
public void mat2BufferedImageTest2() {
    Mat m = Imgcodecs.imread( filename: "/home/gin/Desktop/FIT3161-Team-1C-master/avs/src/test/java/apple.jpg");
    BufferedImage actual = Util.Mat2BufferedImage(m);
    assertEquals(actual.getHeight(), actual: 1515);
    assertEquals(actual.getWidth(), actual: 1600);
    assertEquals(actual.getType(), actual: 5);

}
```

Result: Pass



6. **Test Case 13** Test if string is numeric

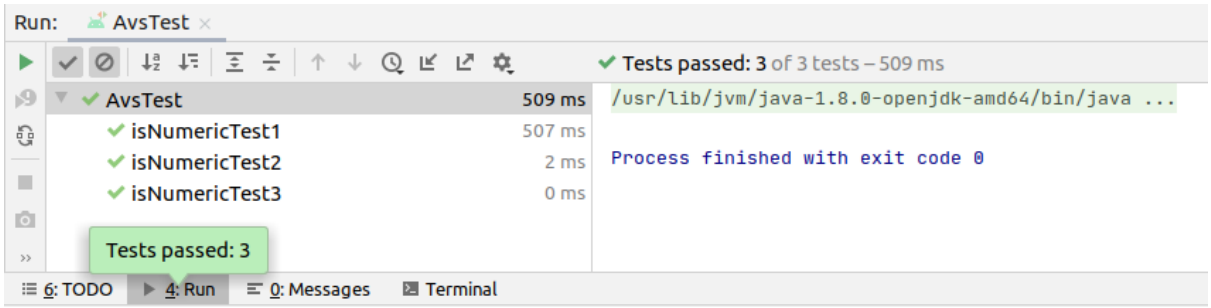
Description:

Tests on isNumeric() static function in Util class to check if a string represent numeric values. The function returns true if it is and false if string is null or not numeric. Three tests are conducted, where the strings are numeric, non-numeric and null and asserted to the right Boolean value.

Code:

```java
@Test
public void isNumericTest1(){
    String s = "100";
    boolean actualOutcome = Util.isNumeric(s);
    assertTrue(actualOutcome);
}

@Test
public void isNumericTest2(){
    String s = "abc";
    boolean actualOutcome = Util.isNumeric(s);
    assertFalse(actualOutcome);
}

@Test
public void isNumericTest3(){
    String s = null;
    boolean actualOutcome = Util.isNumeric(s);
    assertFalse(actualOutcome);
}
```

Result: Pass



7. **Test Case 14** Test if filename generated follows the right expression

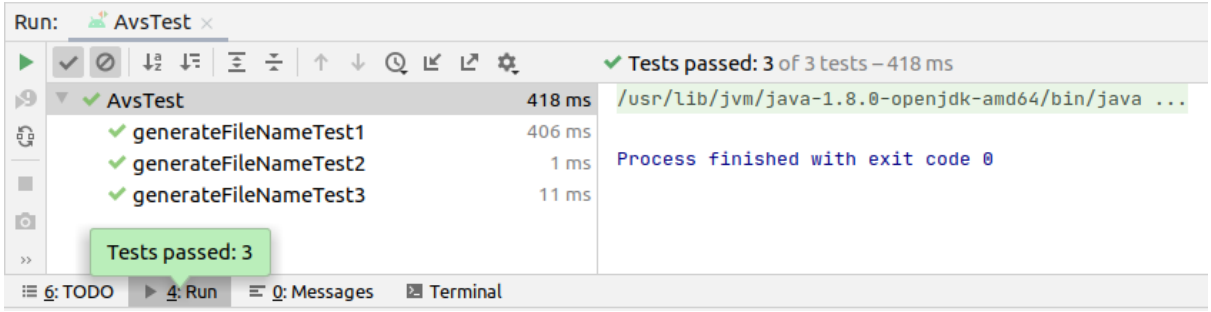
Description:

Tests on generateFileName() function in VideoStreamWriter class to check if the string of the file name generated follows the regular expression of yyyy-MM-dd-HH-mm-ss.avi. The function returns a string and three tests are conducted to assert if the strings matches the expression.

Code:

```
@Test
public void generateFileNameTest1() {
    String output = VideoStreamWriter.generateFileName();
    boolean test1 = output.matches( s: "\\d{4}-\\d{2}-\\d{2}-\\d{2}-\\d{2}-\\d{2}.avi");
    assertTrue(test1);
}

@Test
public void generateFileNameTest2() {
    String output = VideoStreamWriter.generateFileName();
    boolean test2 = output.matches( s: "\\d{4}-\\d{2}-\\d{2}-\\d{2}-\\d{2}-\\d{2}");
    assertFalse(test2);
}

@Test
public void generateFileNameTest3() {
    String output = VideoStreamWriter.generateFileName();
    boolean test3 = output.matches( s: "\\d{4}-\\d{2}-\\d{2}-\\d{4}-\\d{2}-\\d{2}.avi");
    assertFalse(test3);
}
```

Result: Pass



2.2.2 Manual Testing

Each component of the framework are tested independently via manual testing to ensure that each of them are working as expected when the corresponding command to run the Java files or the software is executed in the terminal (shown in the User Guides under Code Report). Before executing the commands for running the Java files, the code package is compiled using the command `mvn clean package` or `sudo mvn clean package` to clear the target directory where Maven builds our project. The tests below are running in the specific order as below because some command has to be executed first in order to see the results in others. This is equal to a gradual testing where components are added in to see if the component will break the system.

1. **Test Case 15** Running Zookeeper and Kafka.

   (a) A terminal is opened inside installed Kafka folder.

(b) The following command is executed in terminal to run Zookeeper.
`bin/zookeeper-server-start.sh config/zookeeper.properties`

(c) The same process is repeated to start Kafka in another terminal with the command.
`bin/kafka-server-start.sh config/server.properties`

(d) **Result:** Zookeeper and Kafka are started and runs as expected.

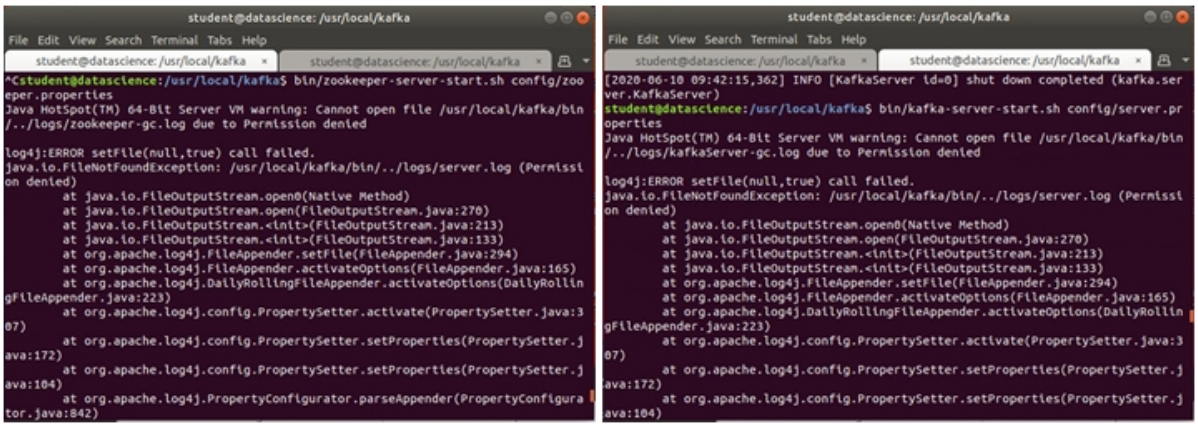


Figure 13: Zookeeper and Kafka running.

2. **Test Case 16** Running VideoStreamDashboard .

(a) A terminal is opened inside the project folder.

(b) The following command is executed in terminal.
```
mvn exec:java \
-Dexec.mainClass=org.team1c.avs.VideoStreamDashboard
```

(c) Verification that the dashboard does not show any data yet until Processor and Collector are run.

(d) **Result:** Dashboard window is started. No data shown on dashboard as expected.

Figure 14: Empty dashboard running when no data in Kafka cluster

3. **Test Case 17** Running VideoStreamArchiver.

   (a) Start VideoStreamArchiver to start video writing by opening a new terminal and running:
   ```
   mvn exec:java \
   -Dexec.mainClass=org.team1c.avs.VideoStreamArchiver
   ```

   (b) **Result:** The system did not crash by connecting VideoStreamArchiver.

4. **Test Case 18** Running VideoStreamProcessor and VideoStreamCollector.

   (a) A terminal is opened inside the project folder for VideoStreamProcessor.

   (b) The following command is executed in terminal for VideoStreamProcessor.
   ```
   mvn exec:java \
   -Dexec.mainClass=org.team1c.avs.VideoStreamProcessor
   ```

   (c) A terminal is opened inside the project folder for VideoStreamCollector.

   (d) The following command is executed in terminal for VideoStreamCollector.
   ```
   mvn exec:java \
   -Dexec.mainClass=org.team1c.avs.VideoStreamCollector
   ```

   (e) **Result:** Video stream collector and processor are running and processing video frames. Dashboard starts showing data of framework as expected.

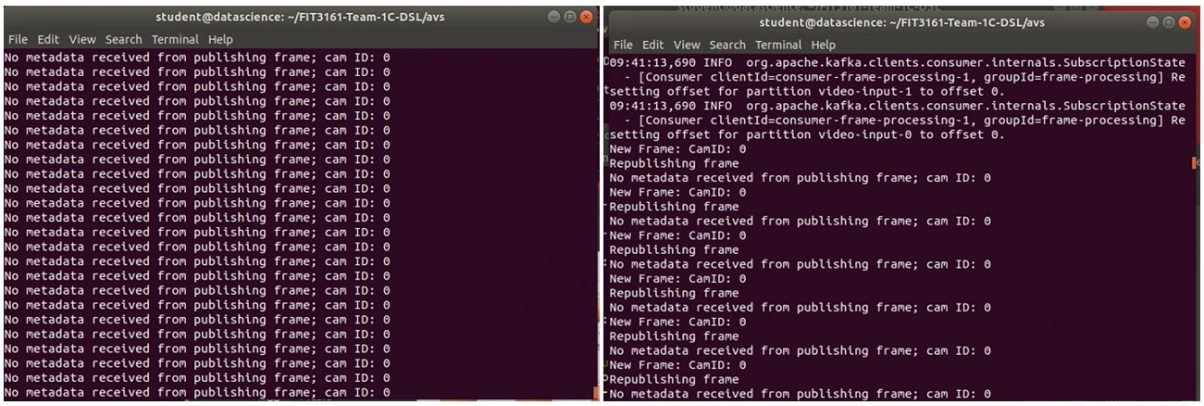


Figure 15: VideoStreamCollector and VideoStreamProcessor running.

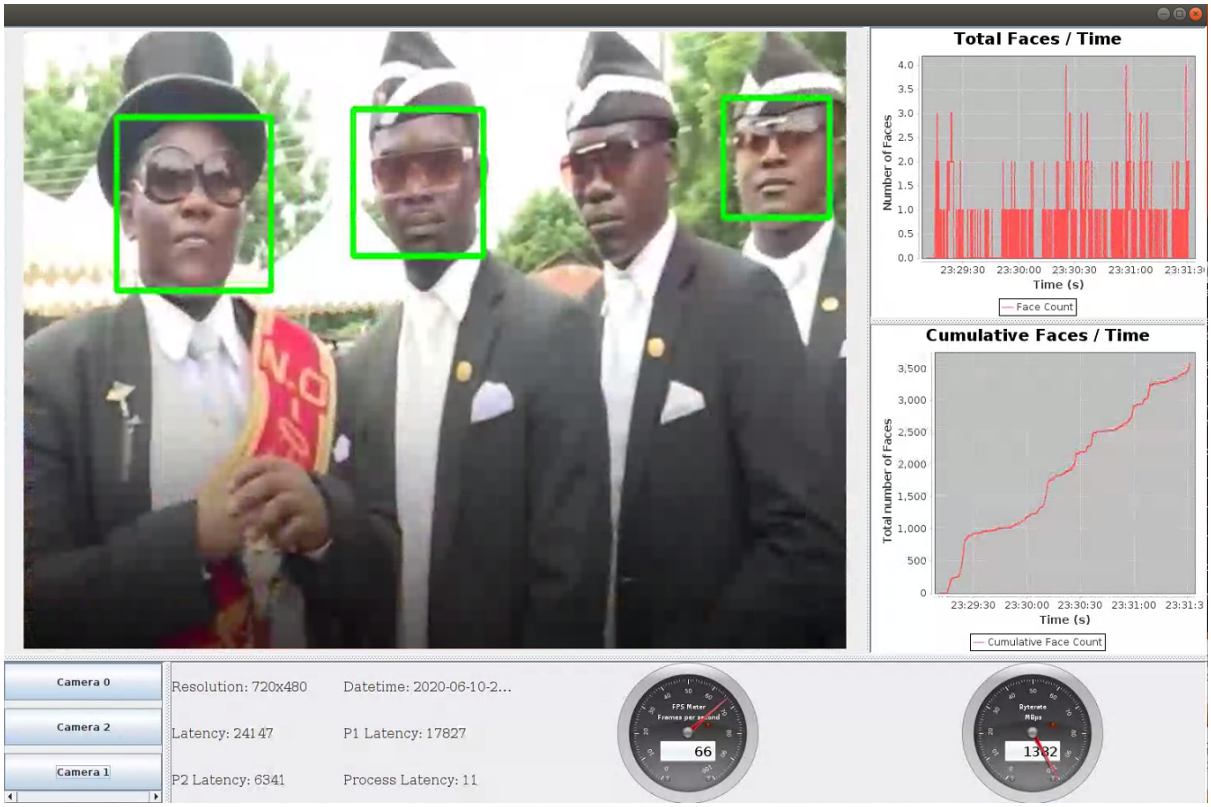


Figure 16: Dashboard showing video and data after data is passed from collector

5. **Test Case 19** Testing archiving in HDFS

   (a) A new terminal window is started and the following command is executed in terminal
   `hdfs dfs -ls /`

   (b) Verify that the log files of the data analytics are stored in Hadoop's subdirectory folder.

   (c) **Result:** Processed video data found in HDFS is listed on terminal and log files of data are stored in Hadoop's subdirectory folder.

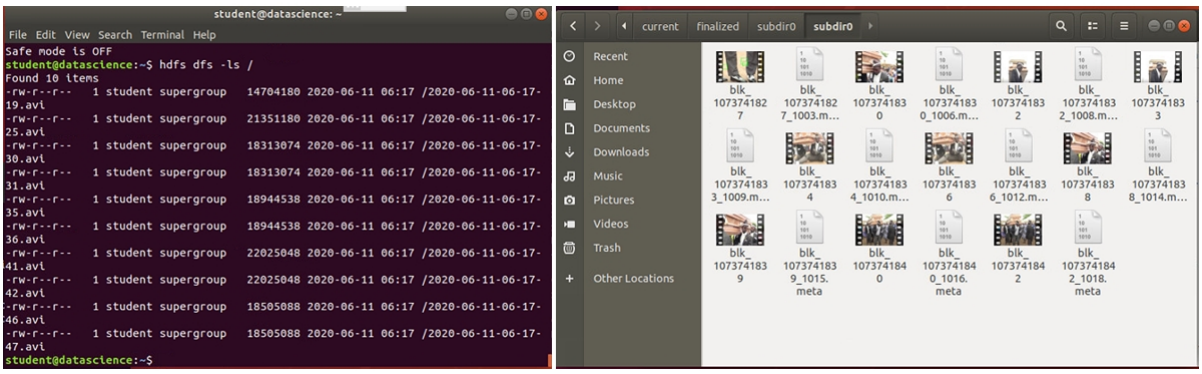


Figure 17: Processed video data found in HDFS listed in terminal and log files in Hadoop's subdirectory folder.

6. **Test Case 20** Testing archiving in MongoDB database.

   (a) A new terminal window is started and the following commands are executed in terminal to enter the mongo shell:
   `mongo`

   (b) The following commands are executed to access the database of our project:
   `show dbs`
   `use avs`
   `db.avgFaces.find().pretty()`

   (c) **Result:** Data analytics from processed video store in MongoDB database is listed on terminal.

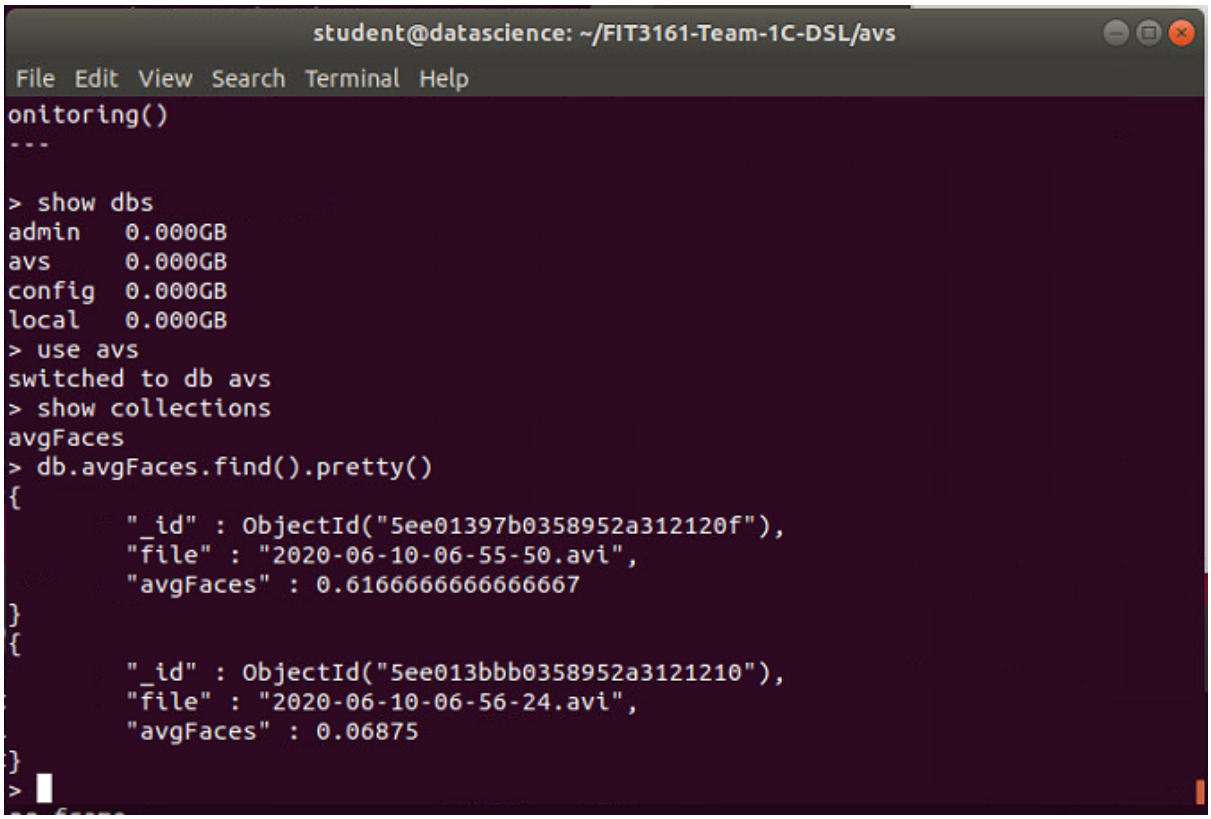


Figure 18: Data from processed video in MongoDB

3 Integration and System Testing

3.1 Integration Testing

In this section of our testing methodology, we tested the individual components as a group. The integration testing is conducted to ensure all requirements of this project are satisfied and the system behaves as expected. Each independently developed units of the system are combined and run together to validate that they can work together as a whole program and tested against different inputs. The performance and scalability of our system are evaluated using latency and frames per seconds (FPS) against different inputs. The procedure of our integration tests closely follows the steps of running the software as provided under Technical User Guide of our Code Report with the difference being the number of input videos fed into the system and the number of processors to process the videos. The main steps in our tests are simplified as below:

1. Start Zookeeper
2. Start Kafka
3. Run VideoStreamArchiver
4. Run VideoStreamDashboard
5. Run VideoStreamProcessor
6. Run VideoStreamCollector

In each of these tests, the FPS as well as three types of latencies are measured: the latency between producer group and processing group (P1), the latency between processing group and dashboard (P2) and the latency in processing the video frame known as the processing latency.

3.1.1 Testing One Video Source with One Processor

Test Case 21:

This test is conducted to measure the latency of the system when one video source is processed by one processor. This test is done without adding any artificial delays to the producer.

Results:

Table 2 extracted from Section 5.3 of Discussion in Final Report shows the FPS and latency of one video and one processor with no artificial delay introduced. P1 shows an overwhelming large increase in latency compared to P2 and processing latency. Based on the values in Table 2, the FPS value is calculated by averaging the FPS measurements in the test which is estimated to be 36.62 FPS.

Time(s)	**P1**	**P2**	**Pr**	FPS
1	956	34	6	29
10	5462	27	9	32
20	12197	26	8	39
30	19108	22	17	32
40	25787	24	16	38
50	32801	27	14	35

Table 2: Induced latency in ms, 1 video, 1 processor, no delay

3.1.2 Testing One Video Source with One Processor with Induced Delay

Test Case 22:

To reduce the stress on Kafka producer by limiting the frequency of sending, we introduced a delay to the producer using a sleep function with 32 milliseconds in the next test. This test is also performed on one video source processed by one processor.

Results:

Table 3 extracted from Section 5.4 of Discussion in Final Report shows the results of the test with induced latency. Results show that the delay has significantly resolved the bottleneck in P1 making P1 latency stable and low. FPS is also stable at an average rate of 24 frames per second. The improvement in terms of latency is due to optimization of the message production rate to the Kafka partition that matches the processing speed.

Time(s)	**P1**	**P2**	**Pr**	**FPS**
1	34	27	13	29
10	21	27	22	24
20	26	25	8	25
30	26	31	17	24
40	36	33	18	21
50	26	22	14	25

Table 3: Induced latency in ms, 1 video, 1 processor, 32ms delay

3.1.3 Testing Three Video Source with One Processor with Induced Delay

Test Case 23:

A test where 3 videos streams is processed by one processor is carried out to estimate the capabilities of the processor. This time, an artificial delay of 72 milliseconds is induced in between frames based on the estimation from the calculation of delay.

Results:

Table 4 extracted from Section 5.4 of Discussion in Final Report displays the results and as expected, the processor can handle 12 frames per second for the three videos while keeping latency stable overall. This shows that our processor can handle 3 videos streams at 12 FPS.

Time(s)	P1	P2	Pr	FPS
1	21	26	8	12
10	48	32	20	12
20	39	37	21	12
30	47	33	21	12
40	35	21	15	12
50	46	30	15	12

Table 4: Induced latency in ms, 3 videos, 1 processor, 72ms delay

3.1.4 Testing Three Video Source with Three Processor

Test case 24:

This test measures the latency of 3 video sources with 3 processors to estimate the scalability of our system when the number of processors are increased.

Results:

Table 5 extracted from Section 5.5 of Discussion in Final Report shows similar results to the tests running one video steam with one processor with the difference being some large latency spikes are observed from the Figure 19. One reasoning for this could be message requests from Kafka's clients are sent at once to the node which caused a spike in latency when the node is being flooded with requests.

Time(s)	P1	P2	Pr	FPS
1	64	32	5	32
10	25	54	9	24
20	59	25	11	25
30	27	22	8	24
40	23	24	6	26
50	31	32	12	20

Table 5: Induced latency in ms, 3 videos, 3 processor, 32ms delay

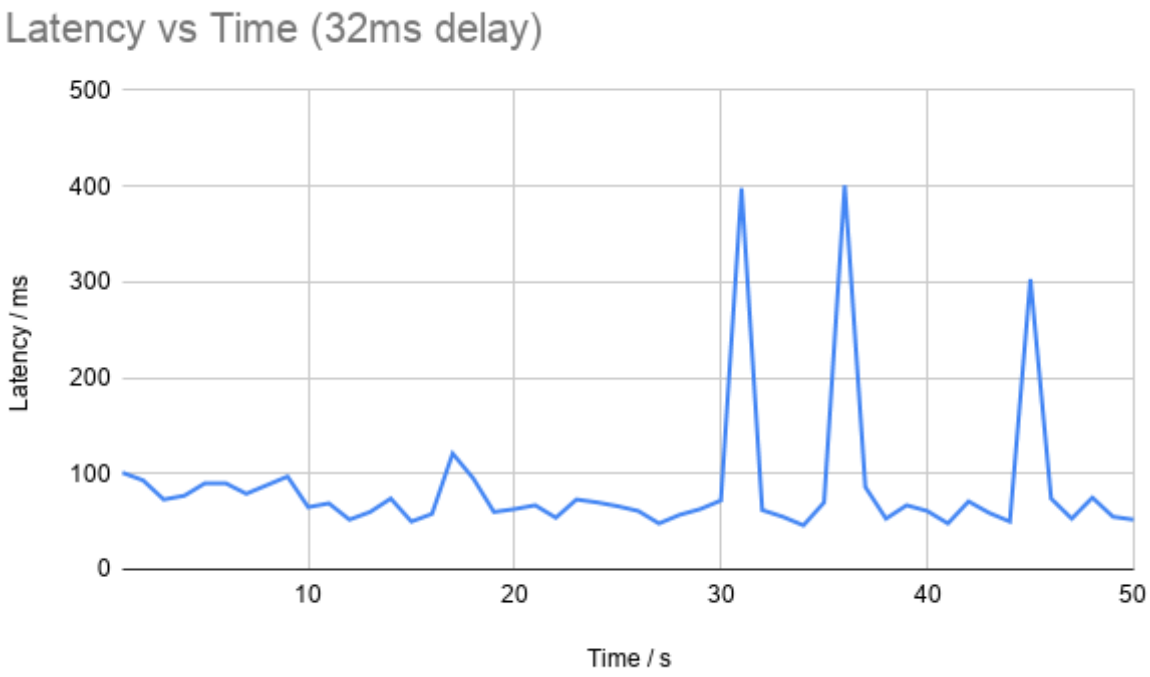


Figure 19: Overall latency of 3 videos, 3 processor, 32ms delay

3.1.5 Testing Three Video Source with Three Processor with Producer Batching

Test case 25:

Time-based batching of multiple frames into one request for producer is tested using interval of 50 milliseconds and 100 milliseconds to attempt to reduce the latency spikes observed in Test case 24.

Results:

Table 6 and Table 7 shows the results of latency and FPS for both cases. The average latency has increase when batching is implemented. Batch interval of 100 milliseconds show even greater increase in average latency than batching interval of 50 milliseconds. This is highly due to the increase in size of the message causing latency in producer. On a good note, as observed from Figure 20 and Figure 21, the latency spikes are not as notable when batching is implemented.

Time(s)	P1	P2	Pr	FPS
1	87	86	7	32
10	93	157	20	22
20	99	89	14	22
30	80	96	13	24
40	94	97	10	24
50	86	117	12	24

Table 6: Induced latency in ms, 3 videos, 3 processor, 32ms delay, 50ms batch interval

Time(s)	P1	P2	Pr	FPS
1	110	256	6	28
10	124	135	10	25
20	142	148	8	24
30	136	139	11	26
40	108	226	8	25
50	190	160	4	24

Table 7: Induced latency in ms, 3 videos, 3 processor, 32ms delay, 100ms batch interval

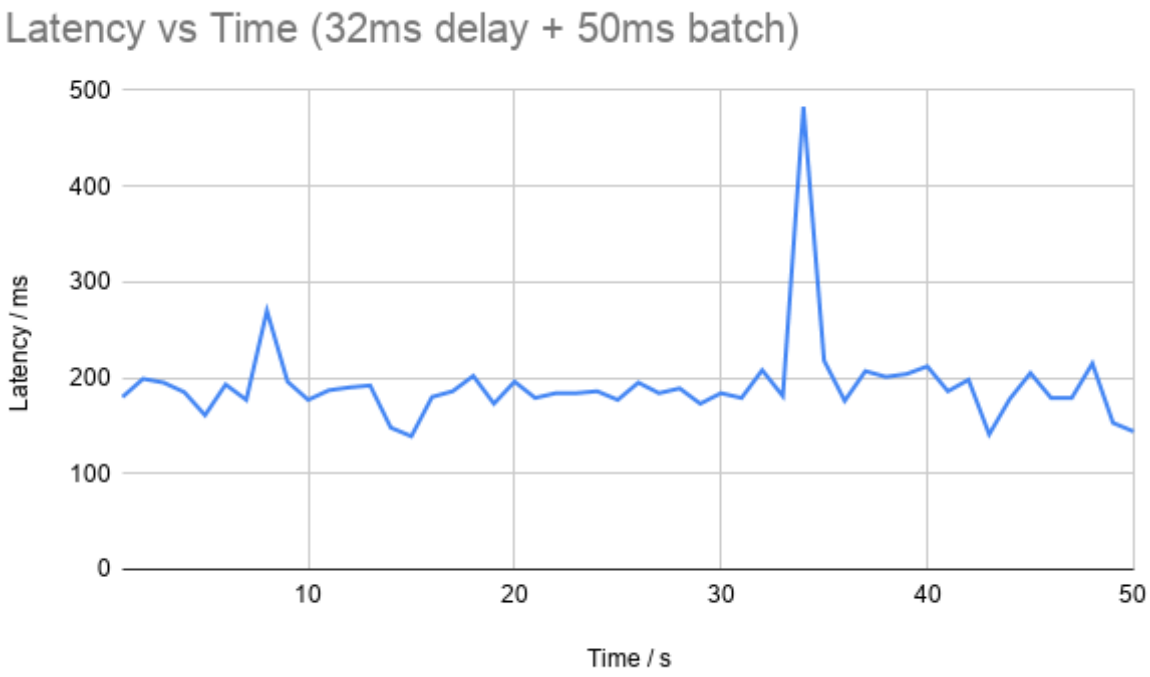


Figure 20: Overall latency of 3 videos, 3 processor, 32ms delay, 50 batch

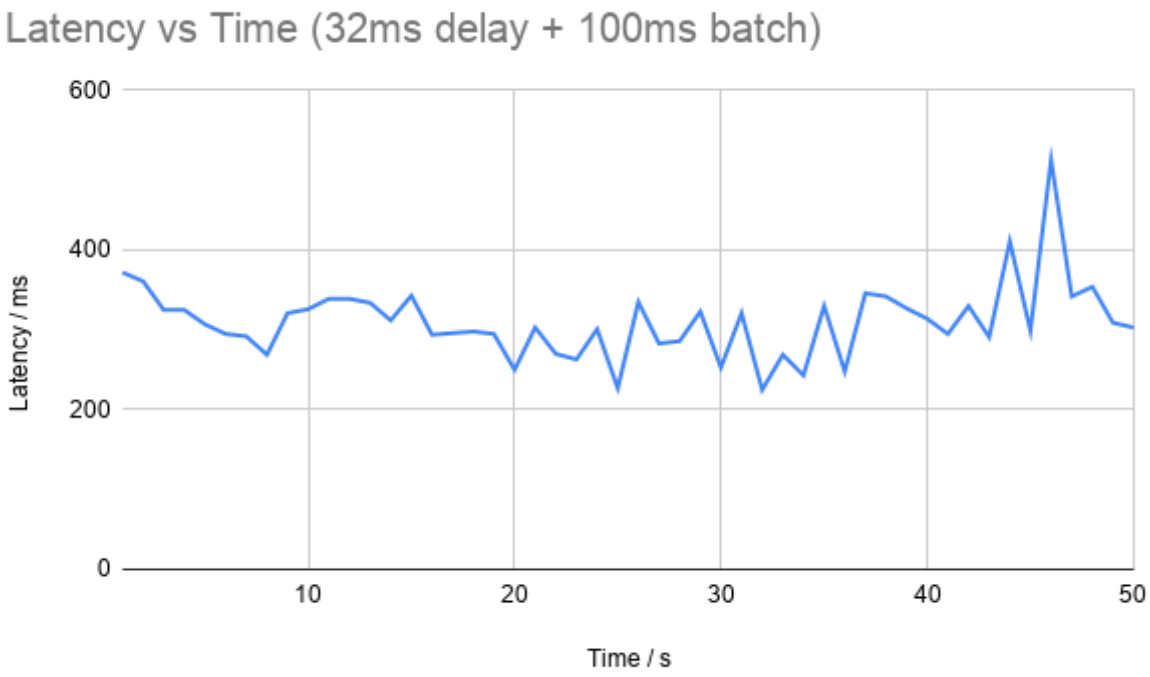


Figure 21: Overall latency of 3 videos, 3 processor, 32ms delay, 100 batch

3.2 System Testing

In system testing methodology of our project, we mostly focused on compatibility testing of our system on different operating systems. Due to hardware limitations, our system is only tested on Ubuntu Linux and Windows 10. The installation and configuration of necessary software setup as specified in code report are completed in these two operating systems. Next, following the procedures of integration testing, the framework of our project is tested on Ubuntu Linux 18.04.03 LTS (Bionic) machine and a Windows 10 machine of different specifications listed below in Table 8.

Operating System	**Ubuntu Linux 18.04**	**Windows 10**
Processor	Intel Xeon W-2145 @ 3.7 GHz	AMD Ryzen 1600 @ 3.2Ghz
Logical (Physical) Number of Processors	16 (8)	12 (6)
Memory Size (RAM)	64 GB	16 GB

Table 8: Machines Specifications

Table 8 have showed that we have tested the system on different operating systems with varying specifications especially on the difference between processing power and memory space. Obviously, the machine with better processing power and memory space have shown better results in the performance of the system. However, the machine running Windows 10 is capable of getting performances similar to the machine running Ubuntu when it runs each daemon independently instead of simultaneously for the entire framework. It is possible to run each component separately since Kafka will store all logs in the log file and retrieving data is just a means of retrieving the correct log files stored.

4 Performance, Scalability, Usability

4.1 Performance & Scalability

Latency and frames per second are used to evaluate the performance of our system. From the test conducted in integration testing, the potential bottleneck of our system is more apparent in the end-to-end latency between producer group and the processing group. To mitigate this problem, we introduced artificial delay into the producer to drop its message production rate to about 24 FPS. Following this, the result shows great improvement in terms of overall latency of the system. The latency between producer group and the processing group that was initially the bottleneck is no longer incrementing because the processor can now handle the current throughput.

In terms of scalability, we ran tests to increase the number of processors to match the number of video source. Each video is now sourced into one processor to maximise FPS and at the same time, minimize system latency. Some spikes of latency are observed when Kafka node is blasted with message requests from clients simultaneously.

This problem can be alleviated by batching multiple frames into one request for producer. Our system implements time-based batching method and two tests were carried out, one with batching interval of 50 milliseconds and another with 100 milliseconds. The spikes of latency observed previously are no longer as prominent after producer batching. However, this improvement came with price of increase in overall latency. Size of message is now larger so the producer holding message for some time has caused average latency to increase. This is because messages are not sent directly when it is produced but waits for the batching time.

4.2 Usability

The functionality of our dashboard is designed to be as user friendly as possible. The dashboard is designed with a large panel showing the video feed clearly and easily comprehensible graph plot of face count and cumulative face count on the right. The data analytics displayed on the bottom panel shows clear annotations and are straight to the point. Gauges are used to display byterate and frame per second (FPS) of the video for a touch of fun and aesthetics on our dashboard.

Besides, users can drag and adjust the panels to their liking and resize them to view the data clearly. The camera buttons on the bottom left section can be scrolled and user can navigate to other video feeds with just a single click on the buttons.

The features of the dashboard are user friendly, however, running the whole system can be complicated. Our system requires a number of terminals with specific command to run the main components of our framework, namely Kafka, Zookeeper, archiver, dashboard, collector and processor. Aside from that, as of right now, our current system does not allow user to query data stored in database from the dashboard itself. Data and processed videos can be retrieved from the database shell terminal. Users with no prior knowledge in IT may find it challenging and complicated to run our system.

5 Limitations

Performance of our system shows high latency when more video input is fed into the system. The latency from data publishing to consuming from event generators to the consumer processor group is approximately 99% of the total latency of the frames passing from the generator to the dashboard in the entire framework. At the time of writing, our performance tests showed that input video streams can be smoothly processed for 3 videos but any more than that, the system will start showing an increase in latency and video processed displayed on dashboard will start showing lag.

An improvement we made is we limit the number of bytes and frequency of sending by using a sleep function for 32 milliseconds. This reduces the stress on Kafka cluster and minimizes the latency. However, the drawback is frames per second (FPS) of output video is also reduced at the same time.

Our system only supports one partition processor for one video input due to Kafka's inability to change partition at run-time. A video input that's already fed into one partition processor cannot be sending its video frames to another partition processor. This is so we can maintain ordering of the video streams within the partitions. Since testing the scalability of the system is limited to only one computer, the extend of how scalable our system and testing the system's capability is restricted due to the lack of hardware to run a cluster computing system.

The producers and consumers of Kafka have their own buffer size settings than can limit the number of bytes to be sent over. A smaller buffer size than optimal will increase overhead when data is constantly sent. On the other hand, a larger buffer size will increase throughput but increase latency at the same time due to accumulating data waiting in the buffer to be filled before it can be sent over.

Finally, all testing are done on the basis and assumption that the input videos are video files that can be read by OpenCV without error. If text files are passed through the system, OpenCV will just repeatedly try to read from the text file and which causes system to fail.