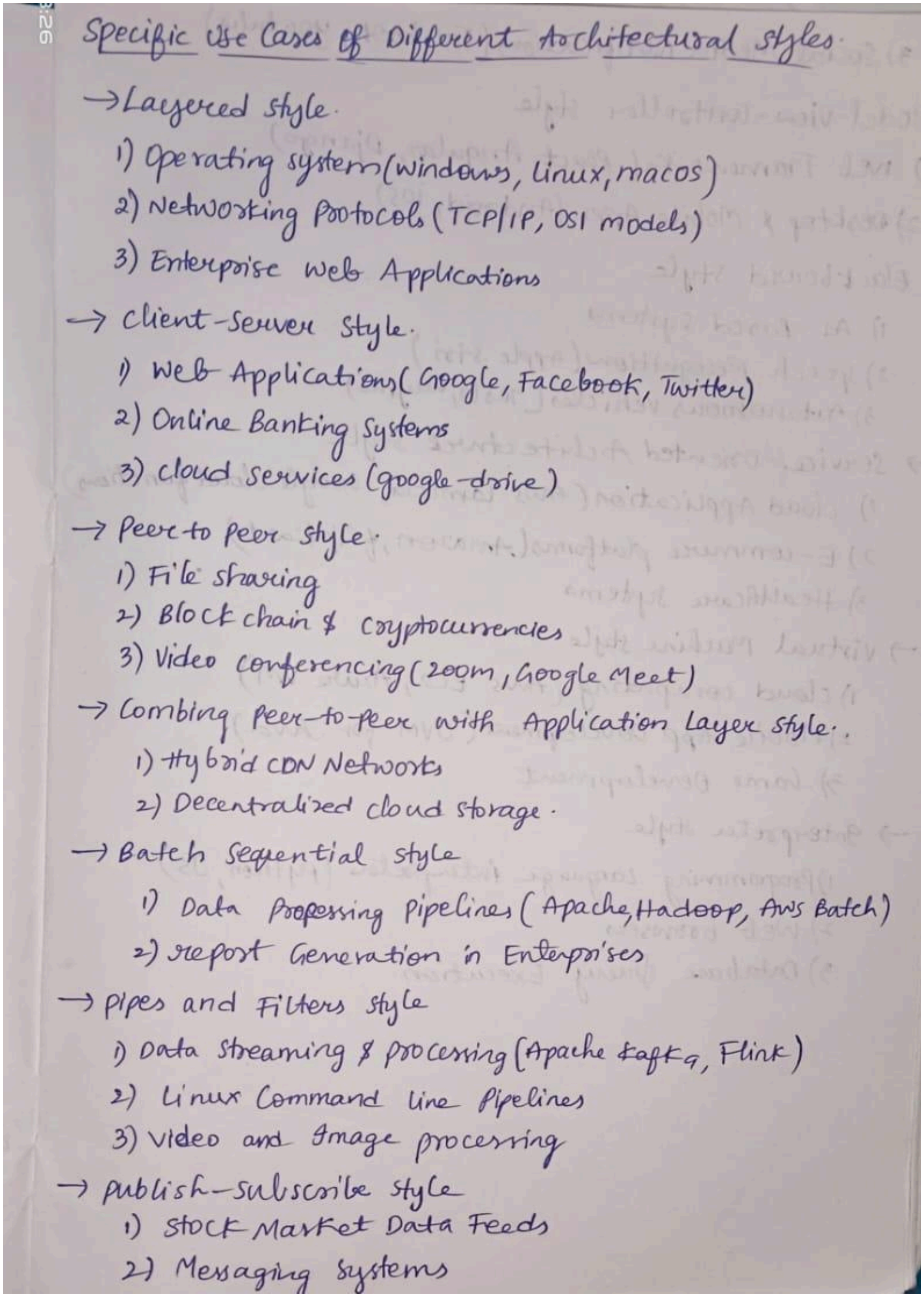
Distributed Systems

Assignment

Name : A Sahithi
Roll No : 100522733001
Course : BE-CSE VI sem

- **Applications of Distributed Architectures**



Specific use Cases of Different Architectural styles.

→ Layered style.

1) Operating system(windows, linux, macos)
2) Networking Protocols (TCP/IP, OSI models)
3) Enterprise Web Applications

→ Client-Server Style.

1) Web Applications(Google, Facebook, Twitter)
2) Online Banking Systems
3) cloud services (google-drive)

→ Peer-to peer style.

1) File sharing
2) Block chain & cryptocurrencies
3) Video conferencing(zoom, Google Meet)

→ Combing peer-to-peer with Application Layer style.

1) Hybrid CDN Networks
2) Decentralized cloud storage.

→ Batch Sequential style

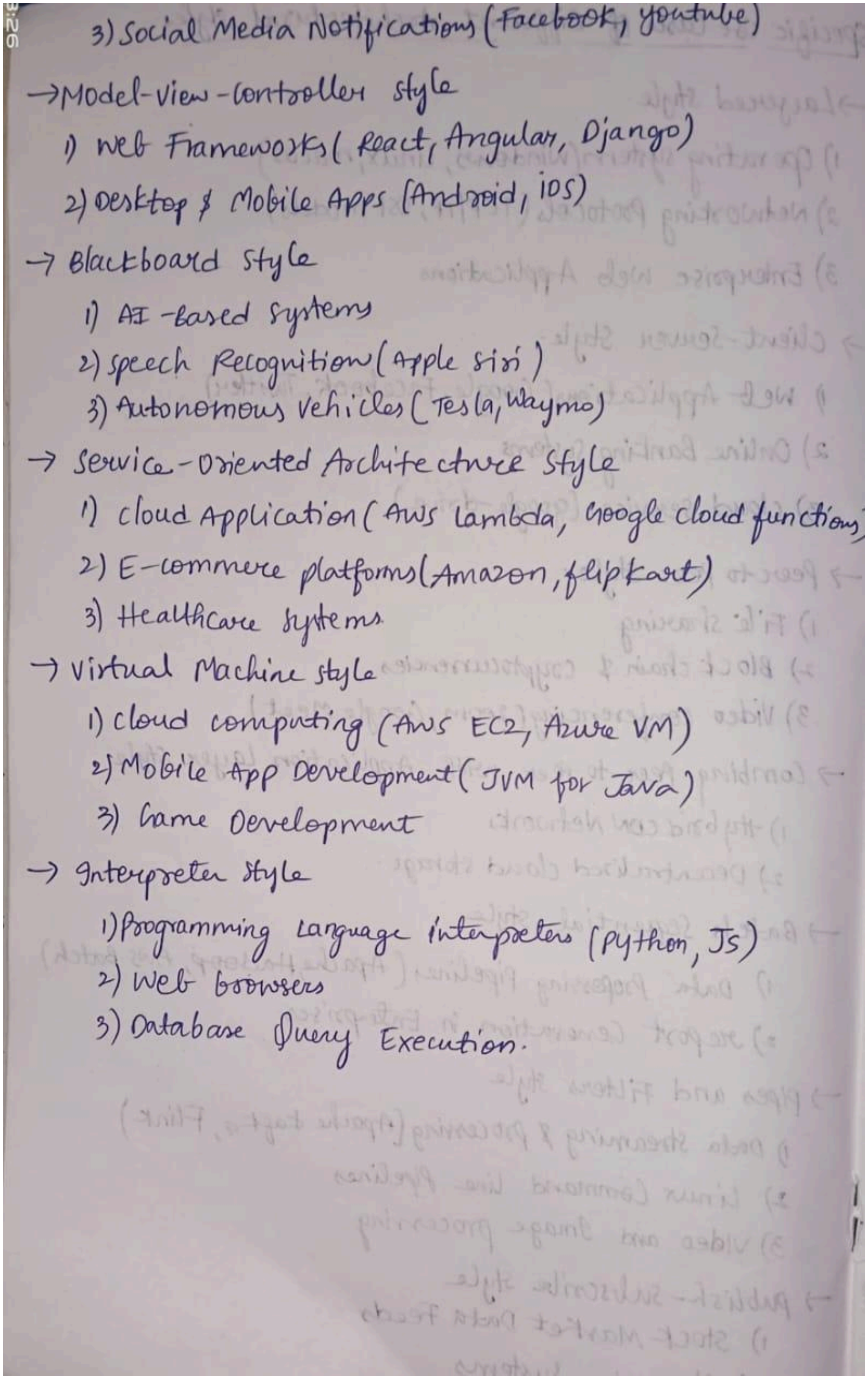
1) Data Processing Pipelines (Apache, Hadoop, AWS Batch)
2) report Generation in Enterprises

→ pipes and Filters style

1) Data Streaming & processing (Apache Kafka, Flink)
2) Linux Command line Pipelines
3) video and Image processing

→ publish-subscribe style

1) Stock Market Data Feeds
2) Messaging systems



3) Social Media Notifications (Facebook, youtube)

→ Model-view-controller style

1) Web Frameworks (React, Angular, Django)
2) Desktop & Mobile Apps (Android, iOS)

→ Blackboard Style

1) AI -based Systems
2) speech Recognition (Apple siri)
3) Autonomous vehicles (Tesla, Waymo)

→ Service-Oriented Architecture Style

1) cloud Application (AWS Lambda, Google cloud functions
2) E-commerce platforms (Amazon, flipkart)
3) Healthcare systems

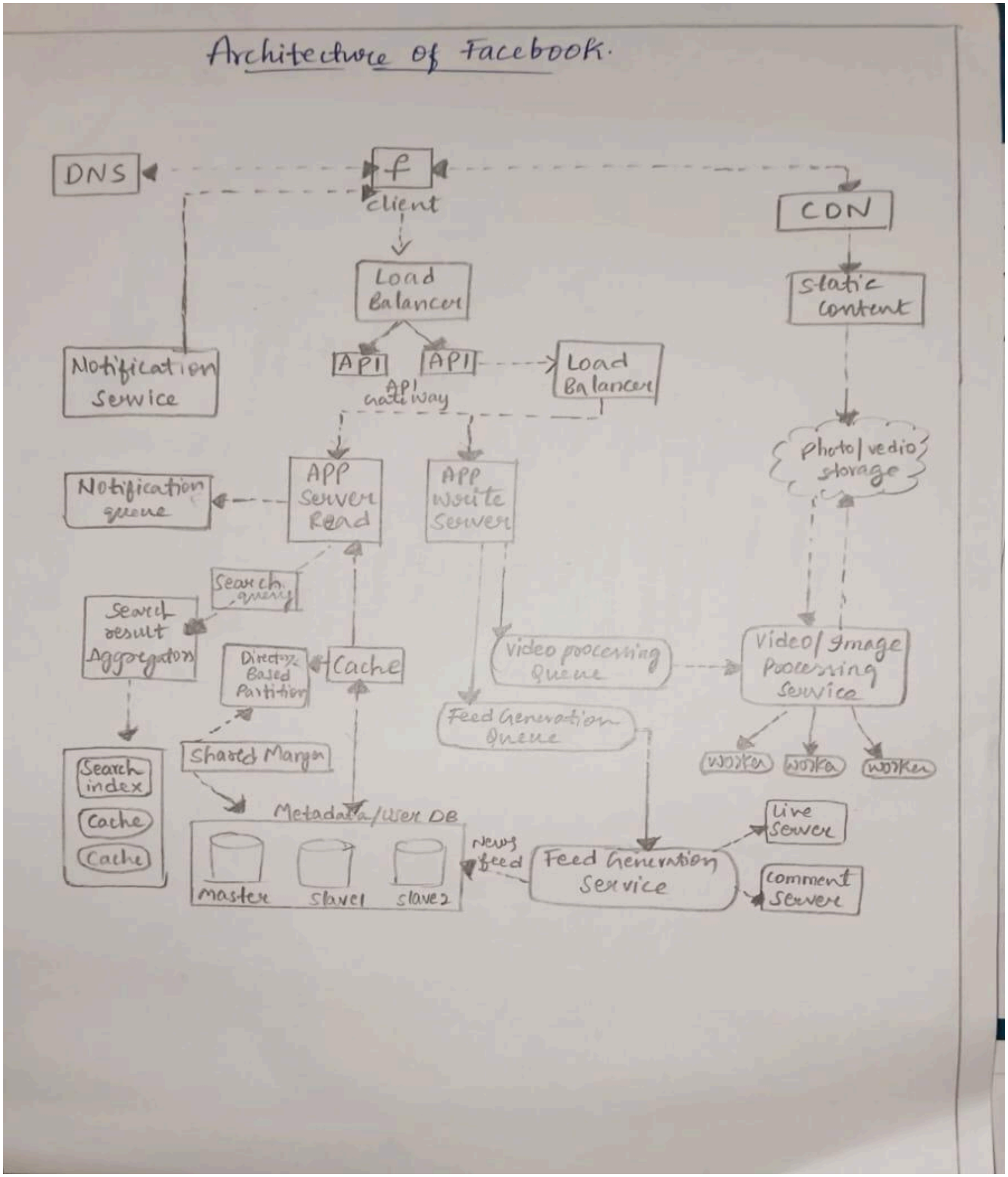
→ Virtual Machine style

1) cloud computing (AWS EC2, Azure VM)
2) Mobile App Development (JVM for Java)
3) Game Development

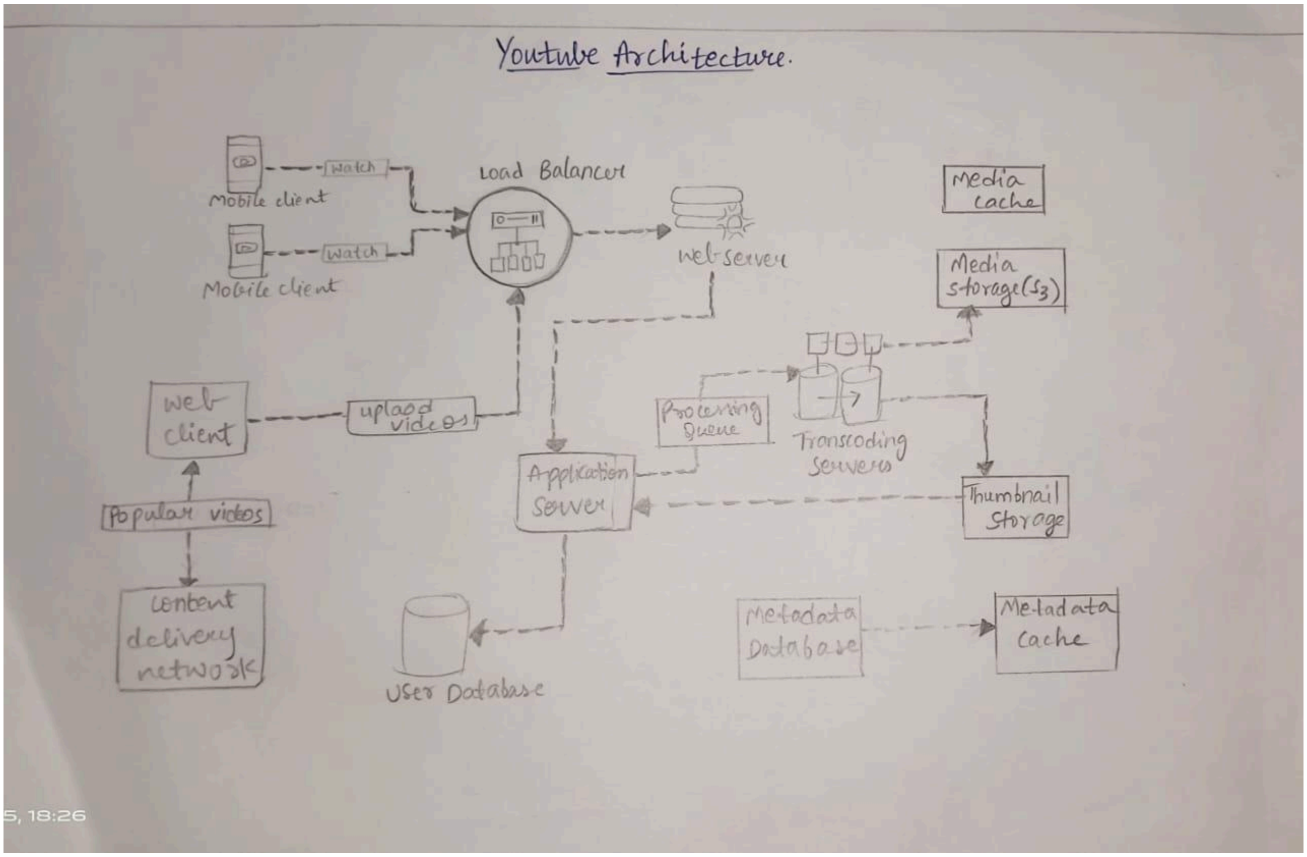
→ Interpreter style

1) Programming Language interpreters (Python, Js)
2) Web browsers
3) Database Query Execution.

- **Architecture Diagram of Facebook**



- **Architecture Diagram of Youtube**



- # MAP REDUCE - WORD COUNT

MapReduce is a programming paradigm introduced by Google for processing and generating large data sets with a parallel, distributed algorithm on a cluster. It consists of two primary functions:

- Map Function: Takes a set of input key/value pairs and produces a set of intermediate key/value pairs. The map function is applied in parallel to each input data block.

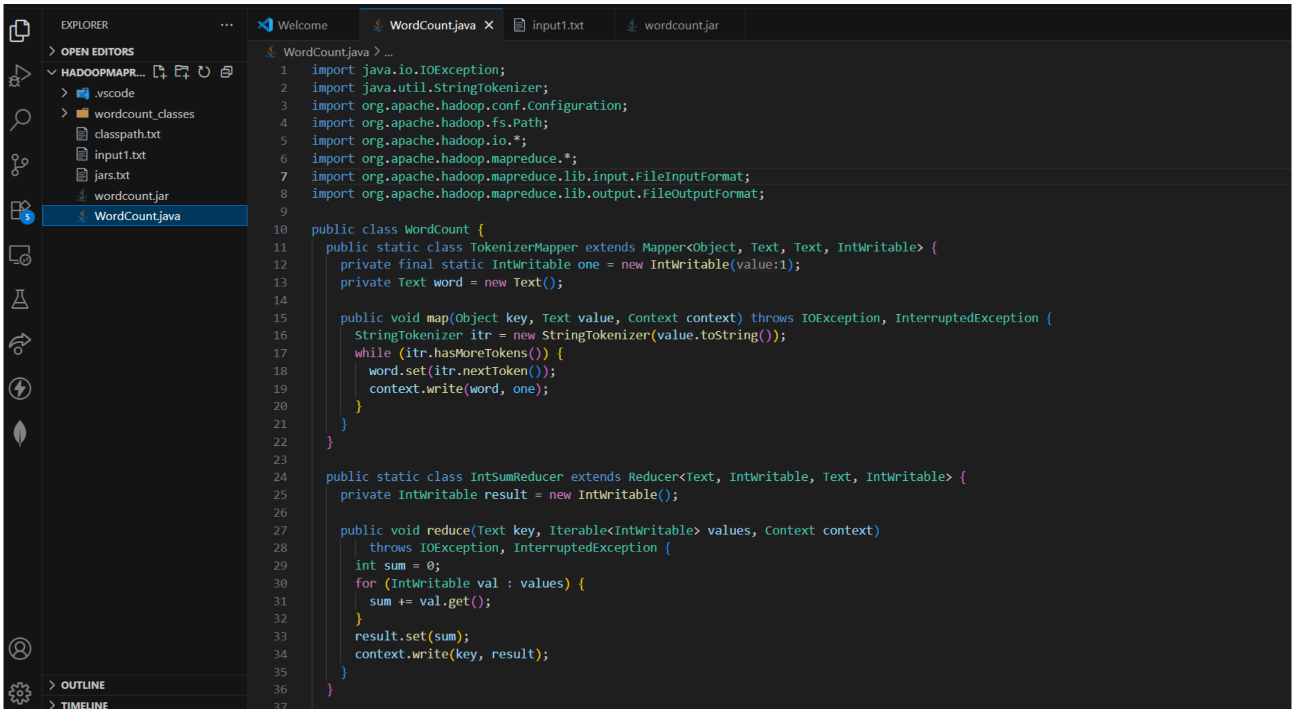
- Reduce Function: Merges all intermediate values associated with the same intermediate key. The reduce function processes these values to produce the final output.

Word Count using MapReduce

The **Word Count** program is a classic example used to demonstrate the MapReduce framework.

- **Map Phase**: Each line of input text is parsed into words. For every word, the mapper emits a key-value pair in the form `(word, 1)`.

- **Reduce Phase**: The reducer sums all the values for each word key and outputs the final count in the form `(word, total_count)`.



```java
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {
  public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(value:1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }
```

Java Program :

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {
  public static class TokenizerMapper extends Mapper<Object, Text, Text,
IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context) throws
IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer extends Reducer<Text, IntWritable,
Text, IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values, Context
context)
        throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Ouput :

```
C:\Users\adlas\Documents\HadoopMapReduce>hdfs dfs -cat /output1/part-r-00000

(bps).  1
1/1000th        1
16-,    1
1945,   1
40      1
64-bit  1
8-bit   1
Along   2
And     1
Arduino 1
As      1
CPU,    1
CPUs    1
CPUs,   1
For     1
From    1
In      3
Initially,      1
```

```
LANs	1
Larger	1
Local-area	1
Moreover,	1
Of	1
Packed	1
Parallel	1
Pi	1
Raspberry	1
Starting	1
States	1
The	3
These	1
This	1
United	1
WANs	1
Well-known	1
Wide-area	1
With	1
a	11
able	1
about	1
actually	2
actuators,	1
adapting	1
advances	1
again	1
ago,	1
airplanes,	1
all	2
allow	2
also	2
amounts	2
an	1
and	10
another.	1
any	1
are	4
as	3
at	3
attack.	1
```