1. **Compilation & Implementation Information**:

Platform: Node.js

Version: v16.20.1

Download: https://nodejs.org/en/download/

Installation: Follow the installation instructions on the Node.js website for your operating system. Node.js is required to run the JavaScript code for the server and worker processes.

Configuration: After installation, we will use npm to install all the required dependencies. This is detailed in the next step. Any other specific package versions are contained in the *package.json* in the project directory and will be downloaded by npm.

1. **How to Compile Our Code**:

Since we used Node.js applications we do not need compilation. Instead, you ensure all dependencies are installed:

1. Navigate to the project directory in the command prompt (Either “Selected” or “Unselected”, publisher-subscriber architecture and blackboard style respectively).
2. Run *npm install* to install all dependencies specified in the package.json file.
3. **How to Execute Our System**:

* Start the Server:
  + Open a new command prompt window and navigate to the project directory where server.js is located.
  + Option 1:
    - Run “*npm run start-server*”
  + Option 2:
    - Run “*node server.js*”
* Start the Worker:
  + Open a new command prompt window and navigate to the project directory where worker.js is located.
  + Option 1:
    - Run “*npm run start-worker*”
  + Option 2:
    - Run “*node worker.js*”
* Access the Application:
  + Open a web browser and navigate to the “*index.html*” file in the project directory to access the application.

1. **Differences Between Architecture Designs and the Rationales For Our Final Selection: Publisher-Subscriber architectural style**

After thorough evaluation and implementation of both the Publisher-Subscriber and Blackboard architectural styles, our team chose to select the Publisher-Subscriber architecture for the final implementation of the project. Here are the detailed rationales behind this choice:

* Decoupling of Components: The Publisher-Subscriber architecture provides a high level of decoupling between components. This separation ensures that changes in one part of the system (for ex, the article fetching mechanism) do not directly impact other components (for ex, the article summarization or scoring services).
* Scalability: This architecture allows for easier scalability. As the system grows and the volume of articles increases, additional instances of subscribers can be deployed to handle increased loads, facilitating load balancing and improving system responsiveness.
* Flexibility in Processing: Subscribers can independently process messages at their own pace. This is particularly beneficial for operations like article summarization and credibility scoring, which may not always execute at uniform speeds.
* Reliability: The use of a message broker (we used RabbitMQ) ensures message delivery reliability. In case a subscriber fails, messages can be re-queued for processing, ensuring that no data is lost.
* Asynchronous Communication: This architecture supports asynchronous processing, allowing the system to efficiently handle incoming articles without blocking other operations.

While the Blackboard architecture also has its merits, especially in terms of collaborative problem-solving, the Publisher-Subscriber architecture was deemed more suitable for the project's needs, particularly concerning scalability and component independence.

1. **Any Changes in Architecture Options and Summary**:

We did not change our architecture options from the project proposal to our final deliverables.

Summary: While the Blackboard architecture offers significant benefits for collaborative problem-solving and dynamic interaction, the Publisher-Subscriber architecture was chosen for its alignment with the project's specific goals, its superior support for scalability, decoupling, flexibility, and fault tolerance. These factors collectively ensured that the Publisher-Subscriber architecture was the optimal choice for the Credibility Project's requirements.

1. **Additional Information (Major Differences in the Source Code For the Different Styles)**:

Pub-Sub Style:

The Pub-Sub style heavily relies on message queues, with RabbitMQ being a central component. The source code includes the establishment of a connection to RabbitMQ, defining exchange types, and setting up queues. Publishers post messages to an exchange, and subscribers listen to the queues bound to these exchanges. This setup is evident in the code where *amqp* library functions are utilized to connect, send, and receive messages. The publisher sends articles to the queue without waiting for the processing outcome. Subscribers independently consume messages from the queue and process them. This is particularly visible in the callback functions and promise handling in the Node.js environment. In the Pub-Sub source code, there is a clear separation between the publisher and subscribers. They interact through the message broker without direct references to each other, emphasizing component decoupling.

Blackboard style:

The Blackboard implementation does not use a message broker but instead interacts directly with a centralized database, acting as the blackboard. The source code includes numerous MongoDB operations where different components read from and write to the shared database. Components in the Blackboard style continuously poll the database to check for new or updated data to process. This contrasts with the Pub-Sub style, where data flows through the system via messages. In the Blackboard style, you'll see periodic database polling and updates, reflecting a more collaborative and dynamic approach to data processing. The Blackboard architecture's source code is structured around a shared knowledge base, with different services updating the same data records. This is a stark contrast to the Pub-Sub style, where data is passed along through the queue without a central shared state.