1. General information

The microservice being developed is intended for automated processing of "clean" digital images of diagrams obtained from **standardly created schemes** in online services (for example, miro, flip-chart, draw.io, sboard.online, holst.so, pruffme.com, Figma, FigJam, app.mural.co, unidraw.io, mts-link.ru, board.vk.company). **Standardly created schemes** are understood to mean any schemes where situations leading to erroneous data were not intentionally created (for example, strong overlapping of blocks on each other, a "bundle" of arrows creating ambiguity of connections, etc.). Schemes obtained from other resources or sites can be transferred from

distortions - this is acceptable.

The service converts the input image into structured JSON with data on recognized objects (shapes, text, arrows, etc.), preserving color characteristics and also includes runtime metadata each process.

2. Supported formats and pre-processing

2.1 Supported input file formats

• Images: png, jpg, heic, heif, webp

• PDF: Only the first page is processed.

2.2 Converting formats

• All input files are converted to a single internal format (for example, PNG or into a pixel array) for further processing.

2.3 Features

• The service works exclusively with “clean” digital images (for example, taking into account the background grid from draw.io and similar services).

• If there is handwriting or extraneous noise/overlays, leading to incorrect recognition, this is not considered a service error.

3. Types of objects and variations of figures

3.1 Shape

• Supported shapes: circle, rectangle, parallelogram, triangle, diamond.

- If a non-standard shape is detected (e.g. star, cloud, polygon, etc.), it is automatically converted to rectangle.
- Shapes can contain text or be textless.

3.2 Sticker

- Always square in shape.
- May contain text or be empty.
- Stickers should not be perceived as squares that are part of a block-scheme.

3.3 Frame

- It is a rectangle that **is always** labeled.
  The absence of a signature means that the element is treated as normal. rectangle.
- A frame can contain other objects (shapes, arrows, text).

3.4 Arrow

- Can be used as a flowchart element (connecting shapes) or as independent object.
- If the arrow is drawn very thin (for example, 1 px), it is saved the required thickness for correct display.
- Arrows drawn with straight lines and with straight lines are supported corners.

3.5 Text

- Can be present separately or as part of figures, stickers or frame.
- If the text attached to a shape extends beyond its boundaries, the result
  Only the part that is inside the outline of the figure is taken (what goes beyond the boundaries is cut off).

4. Text and style processing

4.1 Text

• All recognized texts are converted into a single font – **Inter** (regardless
from the original font in the original image).

• Main attributes (color, shape/object type, size, size) are preserved
text, outline, color, type, thickness, shape, wavy), except for the font, which is always
replaced by **Inter.**

4.2 Going Beyond Boundaries

• When processing text that extends beyond the boundaries of a shape, a
only the part that is inside.

5. Output and JSON structure

The service generates structured JSON, where each object corresponds to one recognized chart
element. An example of the object structure:

json

```
{
  "id": "unique-identifier",
  "type": "rectangle | circle | parallelogram | triangle | diamond | sticker | frame | arrow |
    text",
  "size": { "width": <ÿÿÿÿÿ>, "height": <ÿÿÿÿÿ> },
  "coordinates": { "x": <number>, "y": <number> },
  "properties": {
    "fillColor": "#RRGGBBAA",
    "text": {
      "value": "Recognized text",
      "font": "Inter",
      "color": "#RRGGBB"
    },
    "outline": {
      "color": "#RRGGBB",
      "width": <number>,
```

```
      "style": "solid"
    }
    // Additional fields for arrows, frames, etc.
  }
}
```

5.1 Clarifications on “properties” properties

Common properties for all objects

1. **fillColor**
    - o Format: string #RRGGBBAA (or #RRGGBB if there is no alpha channel).
    - o Description: The fill color of the object. For text objects, this can be the background color (usually not used if there is only text without a background).
2. **outline**
    - o **color:** string in #RRGGBB(AA) format
    - o **width:** default 1pt
    - o **style:** solid
3. **text** (if the object can contain text - for example, a shape with a caption, a sticker, or the text object itself)
    - o **value:** string (the actual text content).
    - o **font:** string; always "Inter".
    - o **color:** string #RRGGBB(AA) (text color).
4. **zIndex**
    - o Format: number.
    - o Determines the drawing order (on top/below other elements).
5. **parentId / groupId**
    - o Indicates that the object is part of a group or nested within a frame.

Properties by object types

1. Rectangle (shape)

• **fillColor**

• **text** (may be absent or centered)

Also for any other shapes.

2. Sticker

• **fillColor**

• **text:** (the text itself and, for example, placement: "center").

• Additionally, there may be a property or rule that the sticker's width is equal to its height (square).

Example: a colored sticker with text in the center.

3. Arrow

In MVP, "arrows" can be understood as:

• Straight arrow.

• Simple arrow with an angle of 90° (rightAngle) - can be implemented as 2-3 checkpoints.

Properties:

• **arrow** (object):

- o **type:**
  - ÿ straight for a straight line.
  - ÿ rightAngle for a right angle.
- o **startDecorationType:** none
- o **endDecorationType:** arrow
- o **lineStyle:** solid
- o **lineWidth:** default 1pt
- o **path:** An array of (x, y) coordinates specifying the path of the arrow.
  - ÿ Store only start and end points (for straight),
  - ÿ or the starting point of the angle (rightAngle) and the ending point (for rightAngle).

• **fillColor** can be used for the line color (sometimes called lineColor).

4. Text (separate object)

If it is a standalone text (just text on the board, not inside a shape):

• **text.value:** the content of the text.

• **fillColor** can usually be omitted or made transparent.

5. Frame

• **fillColor** — a rectangular area of the frame (can be translucent or white).

• **outline** (if necessary, for example, width: 1, style: "solid").

• **text** - used as a frame title. If there is no text, it is considered a regular rectangle (according to requirements).

In MVP, you can treat a frame as a rectangle with a title inside it there may be nested objects (parentId = frame.id).

Example of JSON structure

```
{
  "id": "unique-id-123",
  "type": "rectangle | circle | parallelogram | triangle | diamond | sticker | frame | arrow | text",
  "coordinates": { "x": 100, "y": 200 },
  "size": { "width": 120, "height": 80 },
  "properties": {
    "fillColor": "#808080ff",
    "outline": {
      "color": "#000000ff",
      "width": 2,
      "style": "solid"
    },
    "text": {
```

```
  "value": "Some text",
  "font": "Inter",
  "size": 14,
  "color": "#000000ff",
  "align": "center",
  "bold": false,
  "italic": false
},
"arrow": {
  "type": "straight",
  "startDecorationType": "none",
  "endDecorationType": "arrow",
  "lineStyle": "solid",
  "lineWidth": 2,
  "path": [
   { "x": 100, "y": 100 },
   { "x": 200, "y": 100 }
  ]
},
"frame": {
  "isHidden": false
},
"rotation": 0,
"zIndex": 1,
"groupId": "",
"parentId": ""
```

```
  }
}
```

5.2 Processing metadata

In addition to the main JSON with the recognized objects, the response includes metadata showing the processing time:

- meta_pre: time for pre-processing and converting the file.
- meta_model: model execution time (YOLO, OCR).
- meta_json: time of JSON code generation.

6. Architecture, API and Integration

6.1 Microservice architecture

- The service is implemented as a separate microservice with REST API (based on **FastAPI** in Python).
- Processing occurs asynchronously: each POST request (with a file in the format multipart/form-data) is processed independently, and JSON with recognized objects and metadata is returned to the user in response.

6.2 Integration

- The service integrates with the main system via REST API.
- Asynchronous processing ensures that each user receives their result for instant display in the appropriate interface.

7. Containerization and deployment

7.1 Docker

- The service must be packaged into a Docker image to ensure isolated and portable environment.
- It is assumed that a Dockerfile will be used, which allows installing all necessary dependencies (YOLO model, Tesseract/EasyOCR, etc.) into the container.

8. Performance and Metrics

8.1 Performance

- There are no strict limits on processing time – the faster the service works, all the better.
- The service should be optimized for concurrent processing multiple requests (concurrent access).

8.2 Metrics

- The service must return metadata with information about the processing time:
  - meta_pre: time for transformation and preprocessing file.
  - meta_model: model runtime (recognition using YOLO, OCR).
  - meta_json: The time the JSON output was generated.
- Once the development and finalization of the model is complete, it is necessary to assemble accuracy and performance metrics (number of correctly/incorrectly recognized shapes, processing speed, possible failures) and provide report.

9. Additional requirements and clarifications

9.1 Processing "standard" diagrams

- By “standard generated schemes” we mean any schemes without intentional complex superpositions and ambiguous visual effects (for example, strong intersection of figures, “bunches” of arrows creating parallel, difficult to distinguish connections, etc.).
- Schemes created in the standard way in the listed services (miro, draw.io, Figma etc.) must be transferred correctly.
- Schemes from other resources may be transferred with distortions – this is acceptable.

9.2 Processing objects

- All possible object options (shape, sticker, frame, arrow, text) taken into account.
- When recognizing text attached to a shape, if the text extends beyond it borders, the part that goes beyond the contour is cut off, and only text inside.

10. Technology stack

The following (or similar in functionality) technology stack is used to develop and operate a microservice:

1. **Programming language:**
    - o Python (recommended)
2. **REST API framework:**
    - o **FastAPI** (recommended)
    - o Alternatives (Flask, Django REST) may be used, but preference – FastAPI.
3. **Computer vision model:**
    - o **YOLO** (current version, for example v5/v7/v8) – for detecting figures, shooter etc.
4. **Library for OCR (text recognition):**
    - o **Tesseract, EasyOCR** or any other OCR library with sufficient accuracy.
5. **Library for working with images:**
    - o **OpenCV, Pillow** (depending on preferences and specific tasks).
6. **Containerization means:**
    - o **Docker** (Dockerfile for building the image).

11. Documentation

The project must create and maintain up-to-date documentation that includes:

1. **Description of the application**
    - o Architecture and basic structure of microservice.
    - o General principles of image processing and sequence of steps (pipeline).
2. **API documentation (Swagger / OpenAPI)**

- o Full description of all available endpoints, HTTP methods, request and response formats (including multipart/form-data for file uploads).
- o Request and response examples (JSON with object structure and metadata).

3. **Metadata**

- o A detailed description of the meta_pre, meta_model, meta_json fields and any other auxiliary fields returned by the service.

4. **Interaction and metrics**

- o Information on integration with other services: example request sequences, authorization if required, etc.
- o Methods of collecting and analyzing metrics (processing time, recognition accuracy, number of errors).

5. **Deployment schemes**

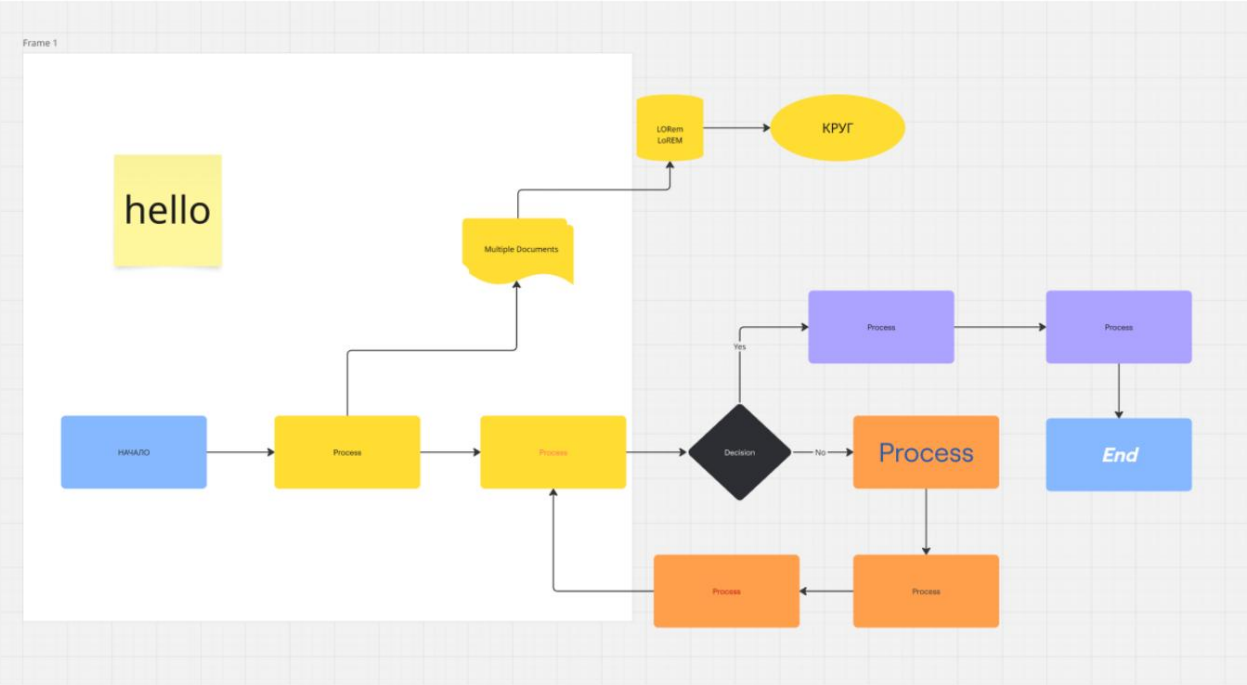
- o Description of building a Docker image and running a microservice.
- o If necessary – use docker-compose, Kubernetes and other scaling and fault tolerance mechanisms.

The documentation should be in the project repository and updated when making any changes that affect the functionality of the service (new endpoints, new versions of models, changes in the JSON format, etc.).

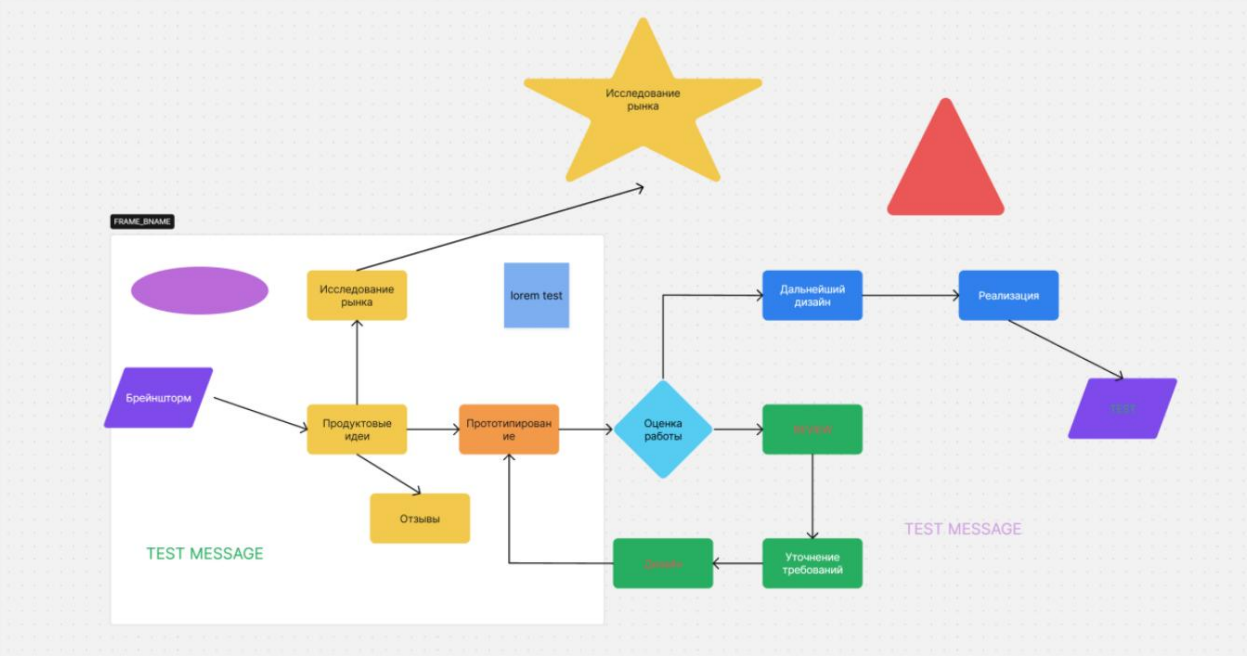
Summary

This Technical Assignment describes the functional, architectural and technological requirements for a microservice that recognizes "pure" digital diagrams. It details the supported object types, requirements for input and output formats, the structure of the resulting JSON, the necessary metadata, the technologies used (stack) and the documentation requirements. Following this Technical Assignment will allow you to create a service that can qualitatively recognize the main elements of flowcharts, correctly visualize them in JSON format and ensure the necessary speed and reliability in production.

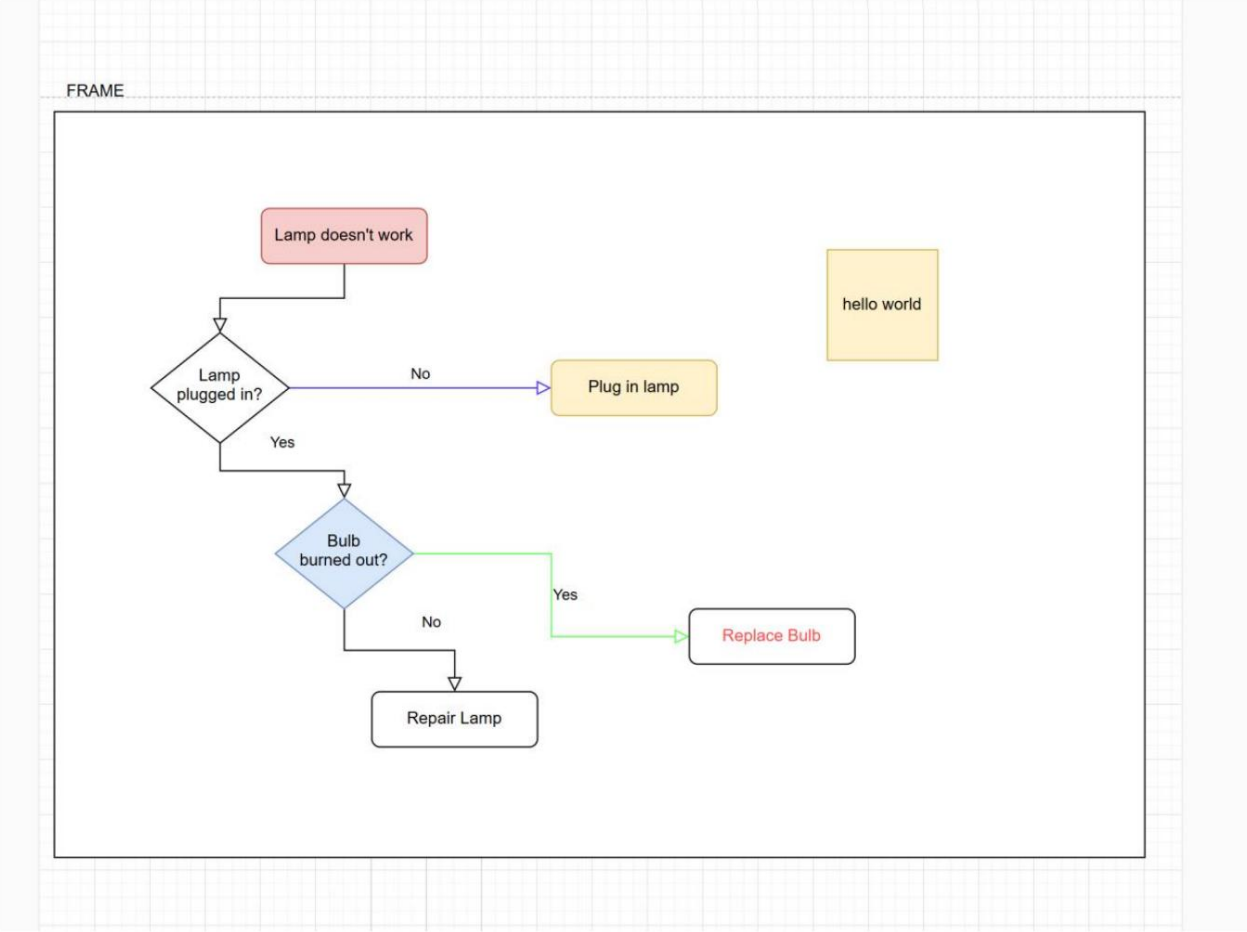
• **Standardized types of flowcharts:**



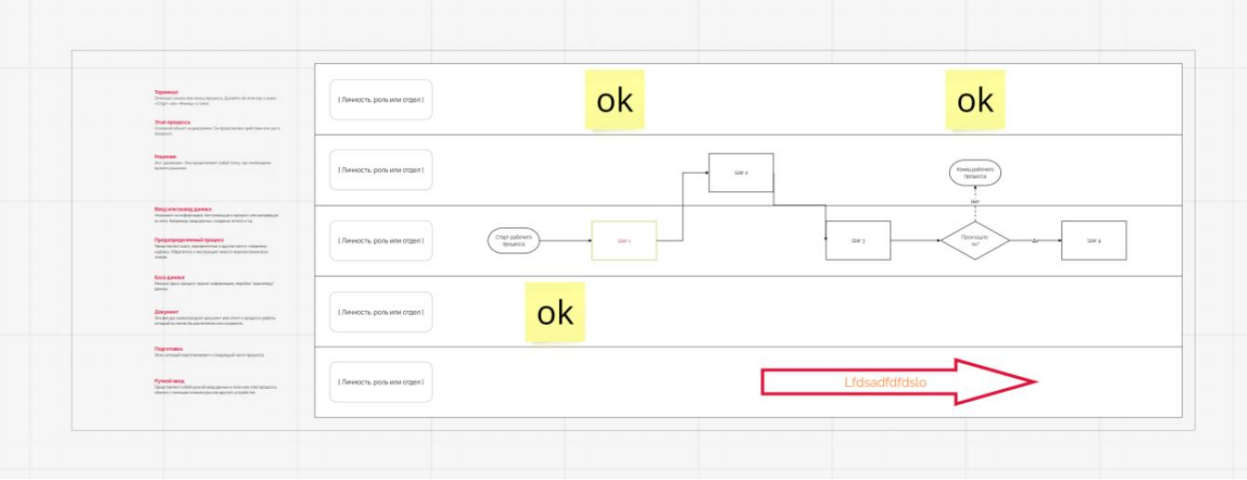
fruit



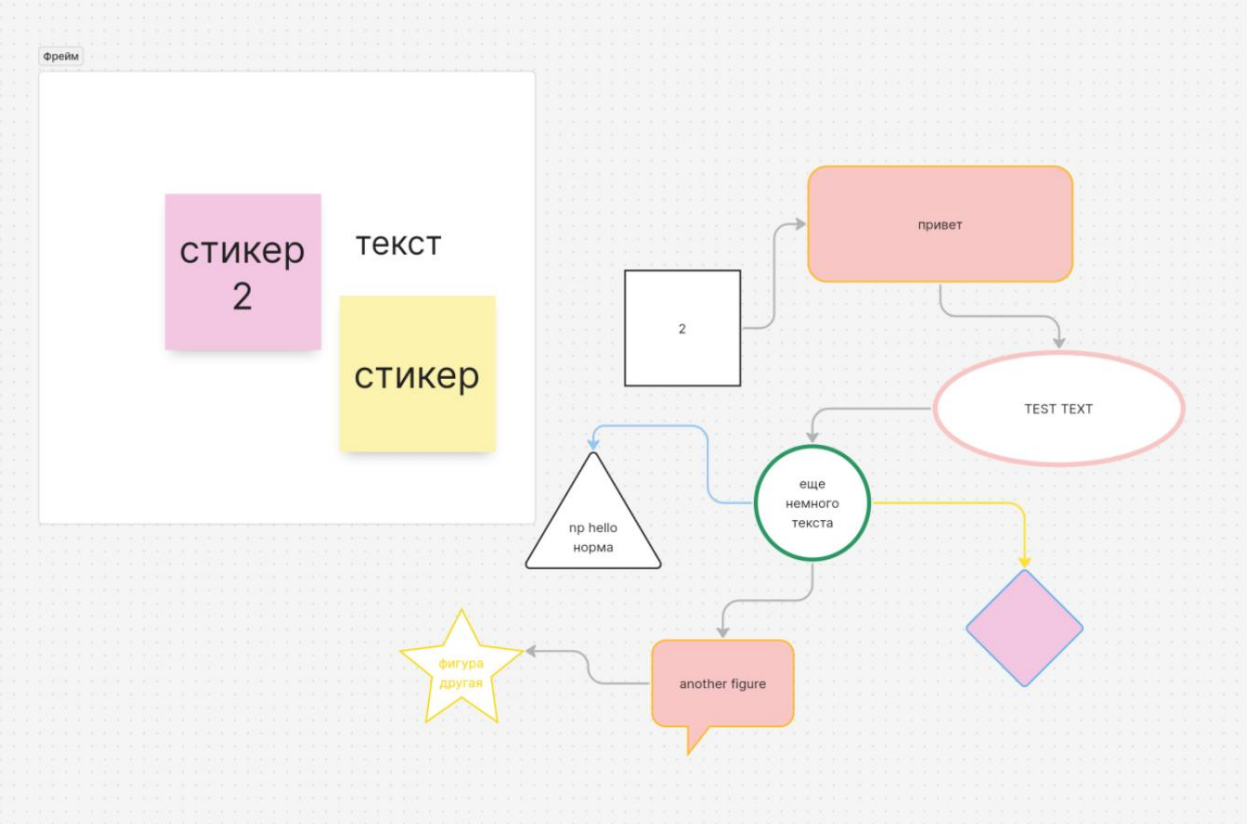
Flipchart



Draw.io

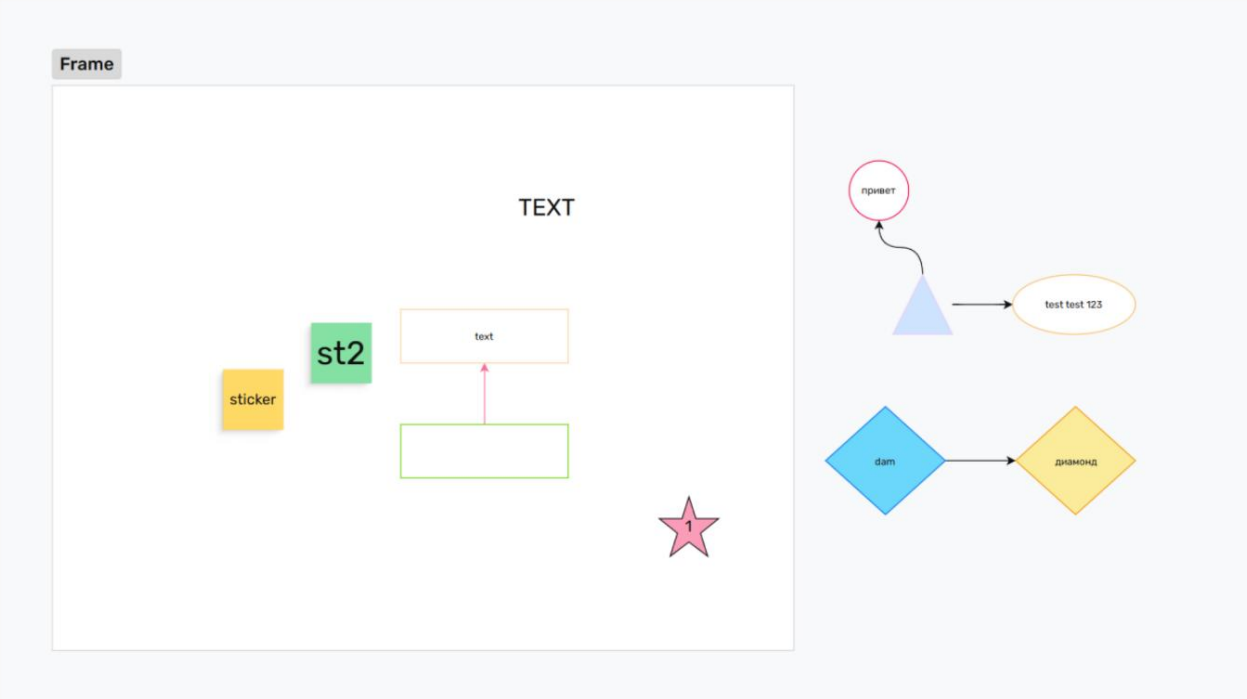


Sboard.online

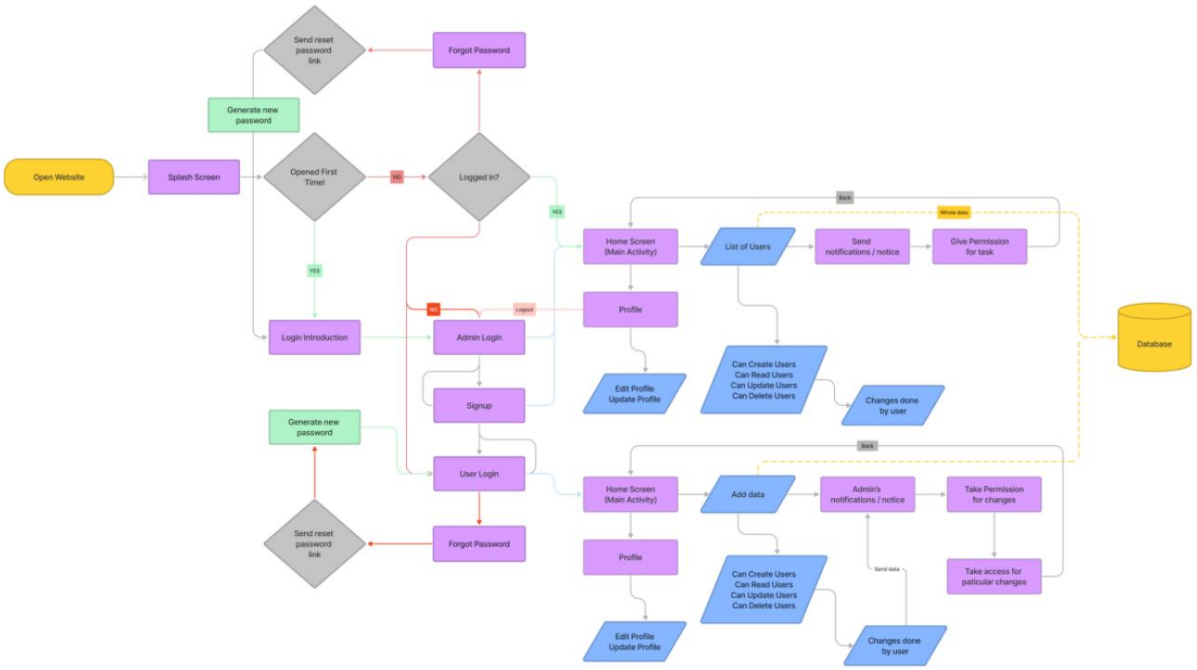


Holst.so

pruffme.com

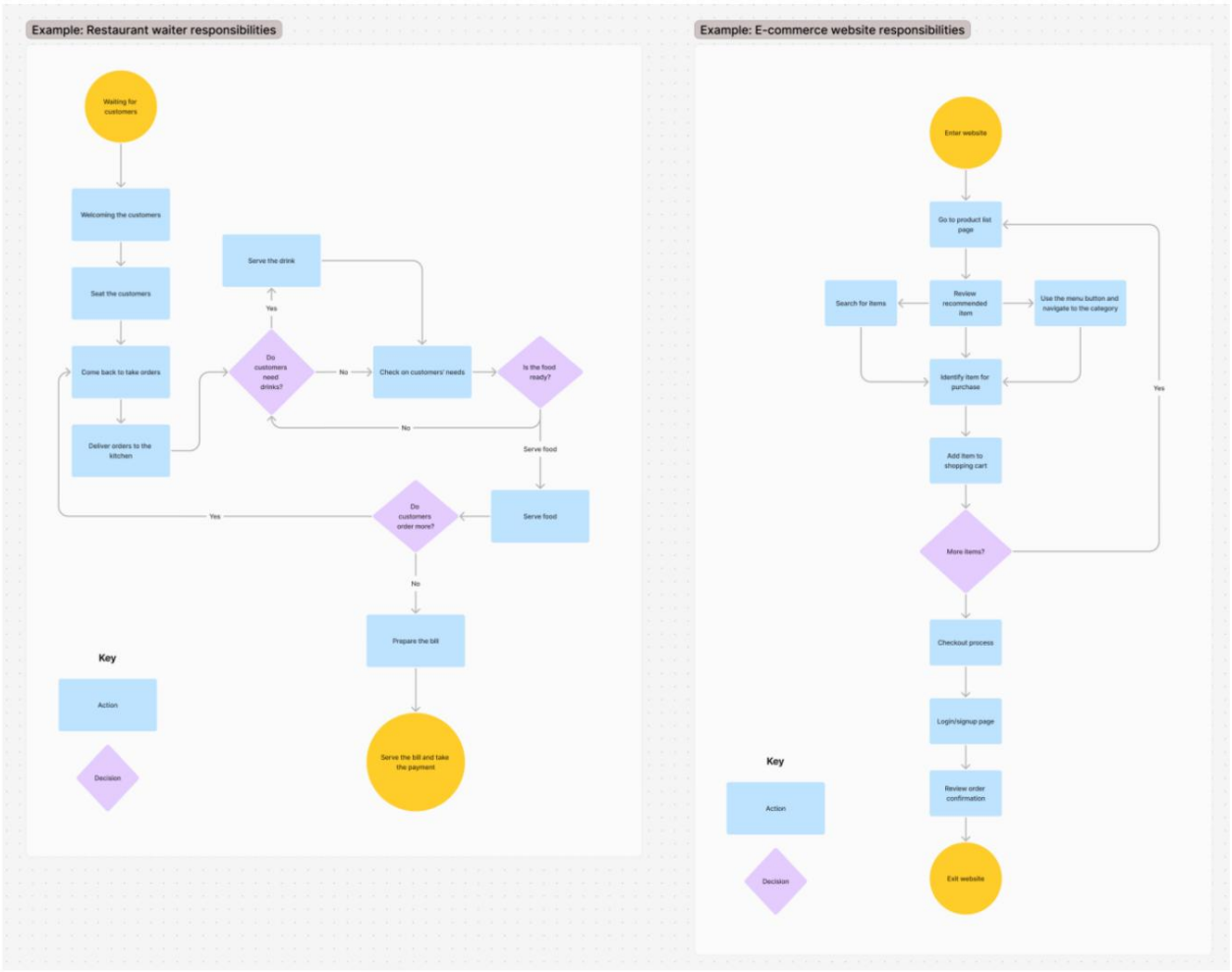


Figma

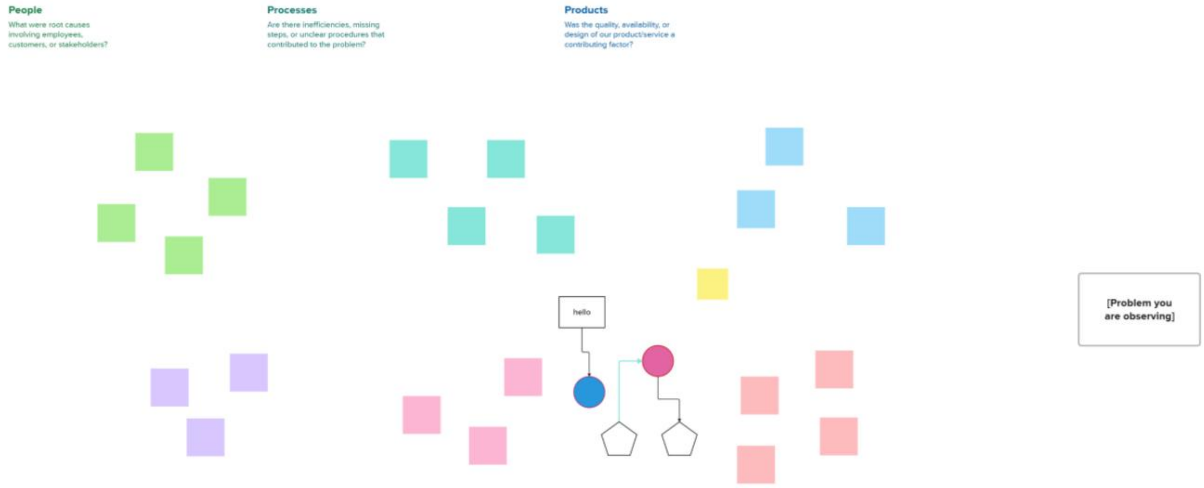


As an example, let the arrows have 2 bends.

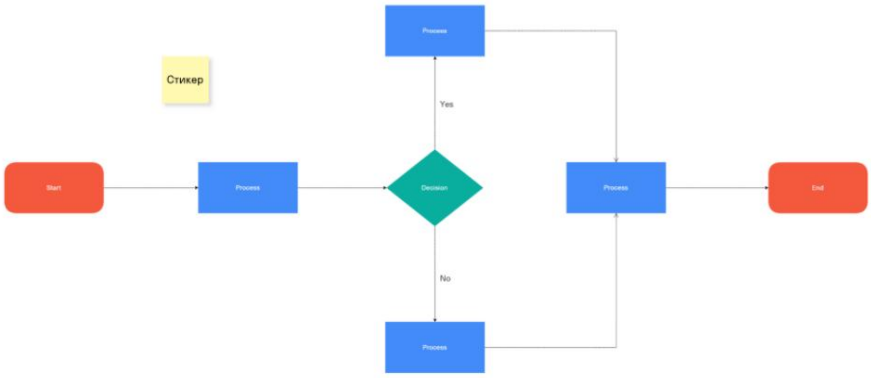
I am a fig.



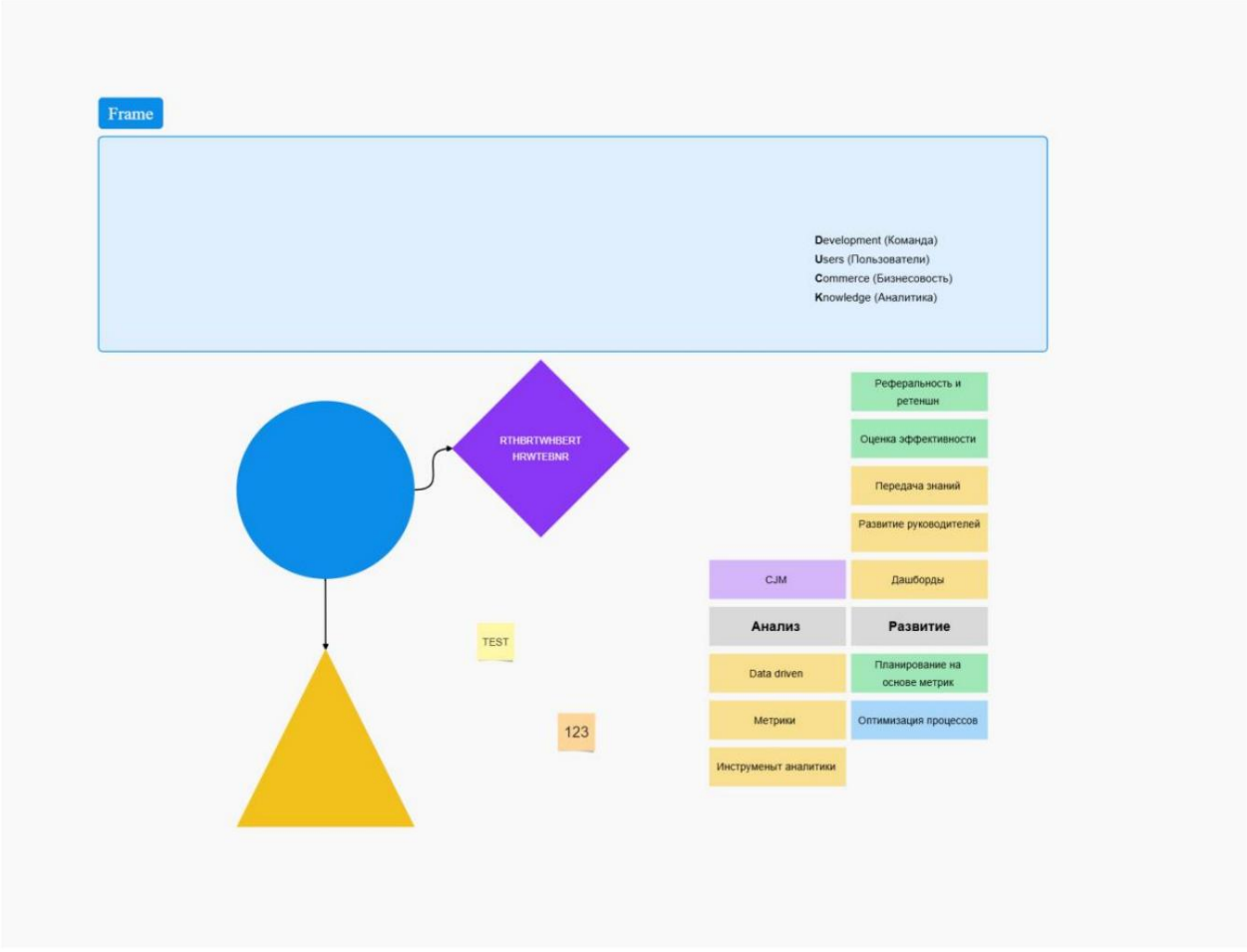
app.mural.co



unidraw.io



mts-link.ru



board.vk.company