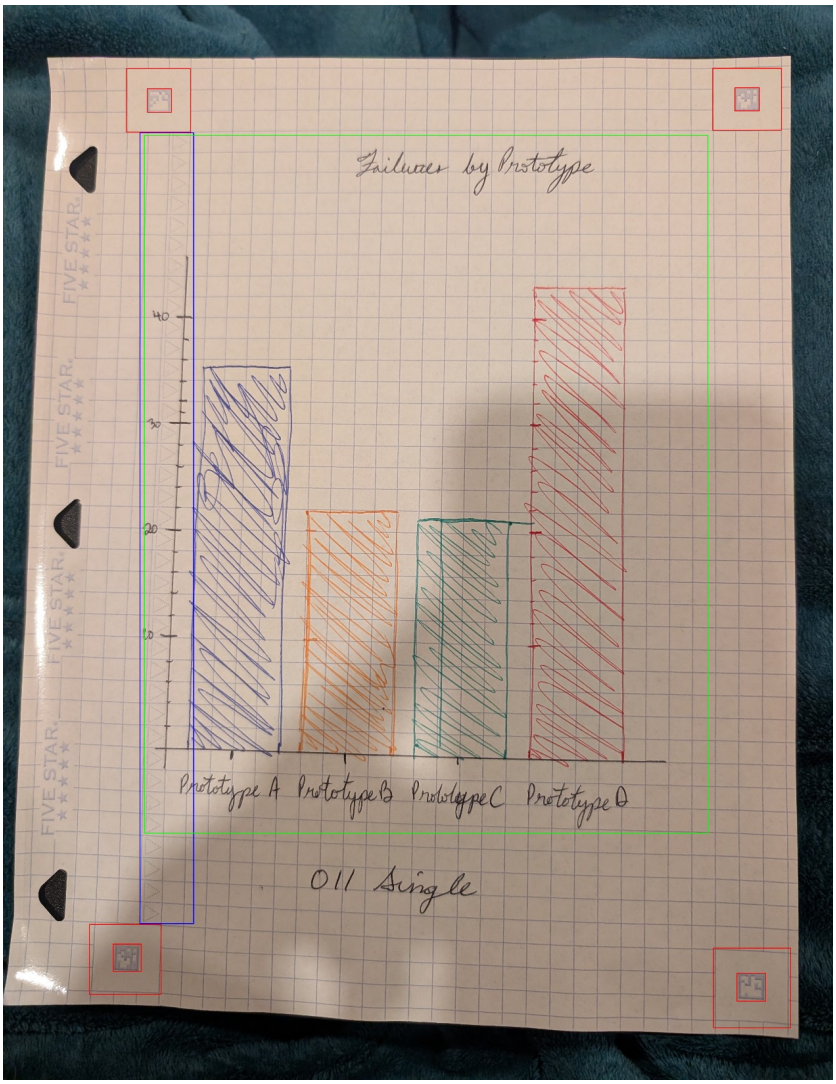
FieldChartOCR: Extraction of Handwritten Charts and Tables



Project Proposal by: Chris Brechin

Prepared for: DTSA-5506

Date: November 18th, 2025

0. Abstract

Handwritten charts and technical notes are rich with quantitative insight yet remain locked inside scanned pages. FieldChartOCR extends ChartOCR to converting hand-drawn charts and tables into structured datasets. The procedure goes through chart classification, element detection, and handwriting OCR, to information extraction. The focus is on delivering a practical tool that field technicians and researchers can rely on without exhaustive manual transcription.

1. Introduction

Digitizing handwritten technical notebooks is labour-intensive. Engineers, students, and analysts rely on annotations, sketches, and mixed layouts that resist automated extraction. Existing OCR tools, like ChartOCR, provide plain text at best; they ignore hand-drawn informatics in order to optimize accuracy. On the other hand, professionals (such as field technicians) must manually re-create charts in tools such as Excel, Desmos, or CAD systems—a significant barrier to iteration and collaboration.

The goal of FieldChartOCR is to build a unified pipeline that ingests handwritten graphs, tables, and mechanical annotations and produces structured artifacts suitable for analytics and visualization. By coupling deep learning with rule-based reasoning, I hope to generalize across chart families while maintaining transparency in intermediate outputs. Delivering this capability advances document understanding, accelerates knowledge sharing, and supports accessibility by providing machine-readable alternatives to complex figures.

2. Related Work

- **Chart Understanding Frameworks:** ChartOCR's hybrid keypoint-and-rule architecture demonstrates the effectiveness of combining deep detection with domain logic. ChartSense and ReVision apply rule-based heuristics but struggle with diverse layouts. DVQA and FigureQA focus on question answering but assume neatly rendered charts.
- **Diagram & Table Recognition:** DiagramParseNet and other graph-parsing models detect nodes and edges in structured diagrams. DeepDeSRT and PubTabNet handle printed tables yet rely on sharp lines and consistent fonts, unlike the irregular strokes in hand-drawn notes.
- **Handwriting OCR & Math Recognition:** Transformer-based handwriting recognizers and MathOCR systems convert cursive and symbolic notation into LaTeX. They typically operate on grayscale inputs and overlook complex layouts.
- **Document Normalization & Imaging:** Research on illumination correction and phase-based representations inspires our preprocessing layer. Multispectral imaging studies inform our roadmap for future sensing modalities.

3. Proposed Work

3.1 Data Sources

- handwritten bar chart diagrams
- synthesized datasets in comma separated lists
- bar charts generated with `matplotlib`

3.2 System Architecture

1. **Preprocessing & Layout Normalization**
    - Perform denoising, skew correction, grid suppression, and edge enhancement tailored to notebook paper.
    - Standardize contrast and dynamic range to support downstream detectors.
    - **Note:** I may move to scanning the images if this part proves too onerous.
2. **Chart Type Classification**
    - Lightweight CNN-transformer hybrid predicts dominant chart categories for each region.
        - The goal is to match the 3 charts recognized by ChartOCR: bar charts, line charts, and pie charts.
    - Supports mixed-content pages by assigning probabilities per region.
3. **Element Identification & OCR**
    - Shared detection backbone with heads for axes, scales, titles, data marks, and table structure.
4. **Semantic Assembly & Output**
    - Associates text blocks with detected elements, enforces basic geometric constraints, and exports Markdown summaries plus CSV datasets.

3.3 Development Plan

- **Phase 1:** Build preprocessing pipeline, layout normalization, and table-to-CSV baseline.
- **Phase 2:** Train chart classifier and core element detectors; integrate handwriting OCR.
- **Phase 3:** Finalize semantic assembly, polish outputs, and iterate on error analysis.

4. Evaluation Plan

- **Effectiveness Metrics**
  - Chart classification macro-F1 across core categories (bar, line, scatter, tables).
  - Element detection precision/recall for axes, titles, and data marks.
  - OCR character error rate (CER) and table reconstruction F1.
- **Efficiency Metrics**
  - End-to-end runtime per page and memory footprint.
- **Experimental Setup**
  - Stratified train/validation/test split by chart type and handwriting style.
  - Baselines: ChartOCR, generic OCR plus heuristic extraction, and manual transcription samples.
  - Ablations on preprocessing and semantic assembly.

5. Timeline & Milestones

Phase	Milestone
1	Finalize annotation schema, label initial dataset.
2	**Implement normalization prototype, integrate baseline handwriting OCR and chart classifier; evaluate on sample pages. (CURRENT)**
3	Deploy element detectors and table recognizer; stand up Markdown/CSV writers.
4	Iterate on semantic assembly, run core evaluations, and prepare presentation materials.

Current progress: synthesized dataset, building dataset for QC square recognition using GIMP commandline.

6. Risks and Mitigation

- **Limited Hand-Drawn Training Data**
  - Mitigation: remove the information extraction steps and keep only the classification.
- **Complex Page Layouts**
  - Mitigation: flag difficult cases for human review.
- **Lighting Noise**
  - Mitigation: use normalization targets, augment brightness/contrast, and monitor detector confidence.

7. Conclusion and Future Work

Currently, FieldChartOCR is stuck in the early stages of OCR work. The page corner detection requires more fine-tuning to become robust. On an ideal sample photo, 4 corners are detected and the image can be cropped. However, average case scenarios where 1 or more corners are not detected proves difficult to remedy. I am improving the corner detection function by augmenting samples of the QC squares with computer-edited samples.

References

1. Liu, X., Chen, L., Wei, Y., Luo, Y., Zhang, H., & Zhou, S. (2021). ChartOCR: Data extraction from chart images via a deep hybrid framework. WACV 2021. https://openaccess.thecvf.com/content/WACV2021/papers/Luo_ChartOCR_Data_Extraction_From_Charts_Images_via_a_Deep_Hybrid_WACV_2021_paper.pdf

2. Savva, M., Kong, N., Chhoun, C., Agrawala, M., & Heer, J. (2011). ReVision: Automated classification, analysis, and redesign of chart images. UIST '11, 393–402. https://doi.org/10.1145/2047196.2047247

3. Jung, D., Kim, W., Song, H., Hwang, J.-I., Lee, B., Kim, B., & Seo, J. (2017). ChartSense: Interactive data extraction from chart images. CHI '17, 6706–6717. https://www.microsoft.com/en-us/research/wp-content/uploads/2017/02/ChartSense-CHI2017.pdf

4. Rowtula, V., Oota, S. R., & Jawahar, C. V. (2019). Towards automated evaluation of handwritten assessments. (Conference details unavailable). https://cvit.iiit.ac.in/images/ConferencePapers/2019/PID6008523.pdf

5. Huang, K.-H., Chan, H. P., Fung, Y. R., Qiu, H., Zhou, M., Joty, S., Chang, S.-F., & Ji, H. (2024). From pixels to insights: A survey on automatic chart understanding in the era of large foundation models. IEEE Transactions on Knowledge and Data Engineering. https://arxiv.org/pdf/2403.12027v4

6. Li, Z., et al. (2024). Improving handwritten mathematical expression recognition via integrating convolutional neural network with transformer and diffusion-based data augmentation. IEEE Access. https://www.researchgate.net/publication/380548640_Improving_Handwritten_Mathematical_Expression_Recognition_via_Integrating_Convolutional_Neural_Network_with_Transformer_and_Diffusion-Based_Data_Augmentation

Appendices

A1: Bar Chart Examples

A2: Page Normalization

A3: Custom Page Types and Features

A4: Example of Custom Anti-Cheat Paper

A1 Bar Chart Examples

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Palette

For uniformity in hand-drawing and barcharts, I have selected a colour-palette that resembles the pens that I have purchased for this endeavor

In [2]:
```
# Define palette for grouped bar charts
grouped_palette = [
    '#1f77b4',  # Blue
    '#ff7f0e',  # Orange
    '#2ca02c',  # Green
    '#d62728',  # Red
    '#9467bd',  # Purple
    '#555555',  # Grey
    '#e377c2',  # Pink
    '#2495C4',  # Light Blue
]

# Visualise the palette

plt.figure(figsize=(10, 1))
plt.bar(range(len(grouped_palette)), [1]*len(grouped_palette), color=grouped_palett
# Add labels to the bars
for i, color in enumerate(grouped_palette):
    plt.text(i, 0.5, color, ha='center', va='center', fontsize=10)
# Remove the y-axis labels
plt.yticks([])
# Remove the x-axis labels
plt.xticks([])
# Remove the plot outline
plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)
plt.gca().spines['bottom'].set_visible(False)
plt.gca().spines['left'].set_visible(False)
plt.show()
```

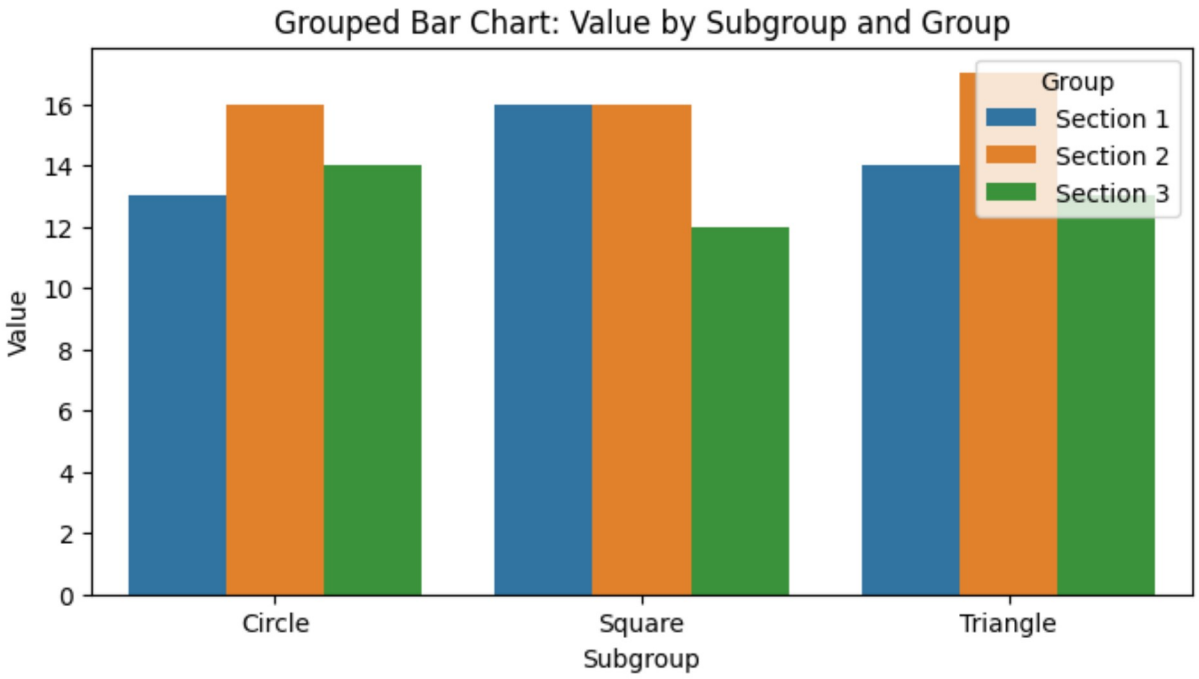
016 Grouped

In [3]:
```
# 1. Simple Bar Chart Example
bc_016_grouped = pd.read_csv('./bar_charts/bar_grouped_016.csv')

bc_016_grouped
```

Out[3]:

	Group	Subgroup	Value
0	Section 1	Circle	13
1	Section 1	Square	16
2	Section 1	Triangle	14
3	Section 2	Circle	16
4	Section 2	Square	16
5	Section 2	Triangle	17
6	Section 3	Circle	14
7	Section 3	Square	12
8	Section 3	Triangle	13

In [35]:
```
plt.figure(figsize=(8, 4))
sns.barplot(
    data=bc_016_grouped,
    x='Subgroup',
    y='Value',
    hue='Group',
    palette=grouped_palette[:3]
)
plt.xlabel('Subgroup')
plt.ylabel('Value')
plt.title('Grouped Bar Chart: Value by Subgroup and Group')
plt.legend(title='Group')
plt.show()
```



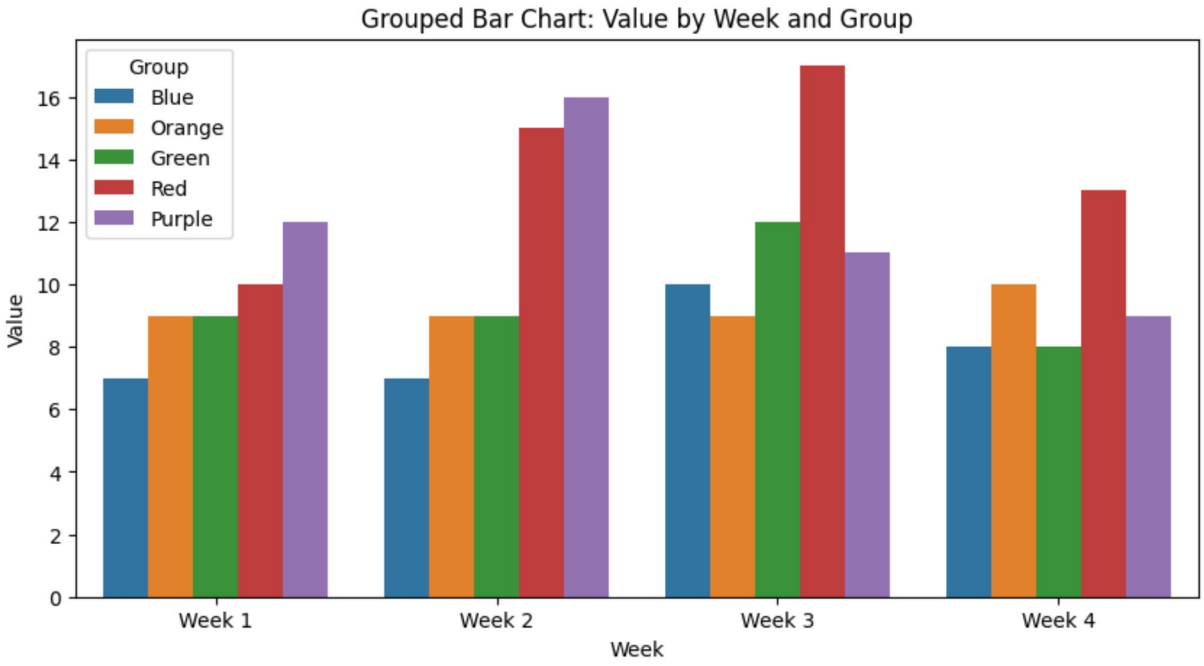
017 Grouped

In [36]:
```
## 002

bc_017_grouped = pd.read_csv('./bar_charts/bar_grouped_017.csv')

bc_017_grouped
```

Out[36]:

	Group	Subgroup	Value
0	Blue	Week 1	7
1	Blue	Week 2	7
2	Blue	Week 3	10
3	Blue	Week 4	8
4	Orange	Week 1	9
5	Orange	Week 2	9
6	Orange	Week 3	9
7	Orange	Week 4	10
8	Green	Week 1	9
9	Green	Week 2	9
10	Green	Week 3	12
11	Green	Week 4	8
12	Red	Week 1	10
13	Red	Week 2	15
14	Red	Week 3	17
15	Red	Week 4	13
16	Purple	Week 1	12
17	Purple	Week 2	16
18	Purple	Week 3	11
19	Purple	Week 4	9

In [6]:
```
# Plot the bar chart
plt.figure(figsize=(10, 5))
import seaborn as sns

sns.barplot(
    data=bc_017_grouped,
    x='Subgroup',
    y='Value',
    hue='Group'
)
plt.xlabel('Week')
plt.ylabel('Value')
plt.title('Grouped Bar Chart: Value by Week and Group')
plt.legend(title='Group')
plt.show()
```



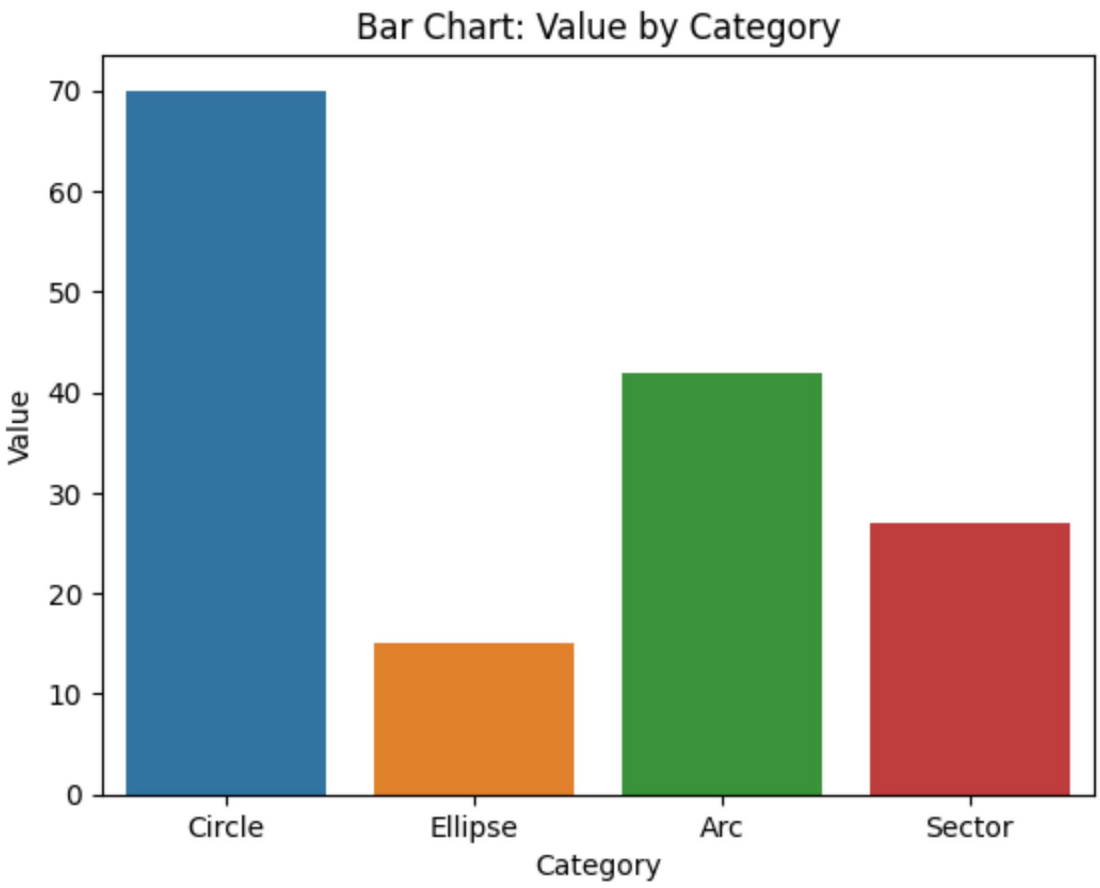
001 Single

In [7]:
```
bc_001_single = pd.read_csv('./bar_charts/bar_single_001.csv')

bc_001_single
```

Out[7]:

	Category	Value
0	Circle	70
1	Ellipse	15
2	Arc	42
3	Sector	27

In [8]:
```
sns.barplot(
    data=bc_001_single,
    x='Category',
    hue='Category',
    y='Value',
    palette=grouped_palette[:len(bc_001_single)]
)
plt.xlabel('Category')
plt.ylabel('Value')
plt.title('Bar Chart: Value by Category')
plt.show()
```



002 Single

In [9]:
```
bc_002_single = pd.read_csv('./bar_charts/bar_single_002.csv')

bc_002_single
```

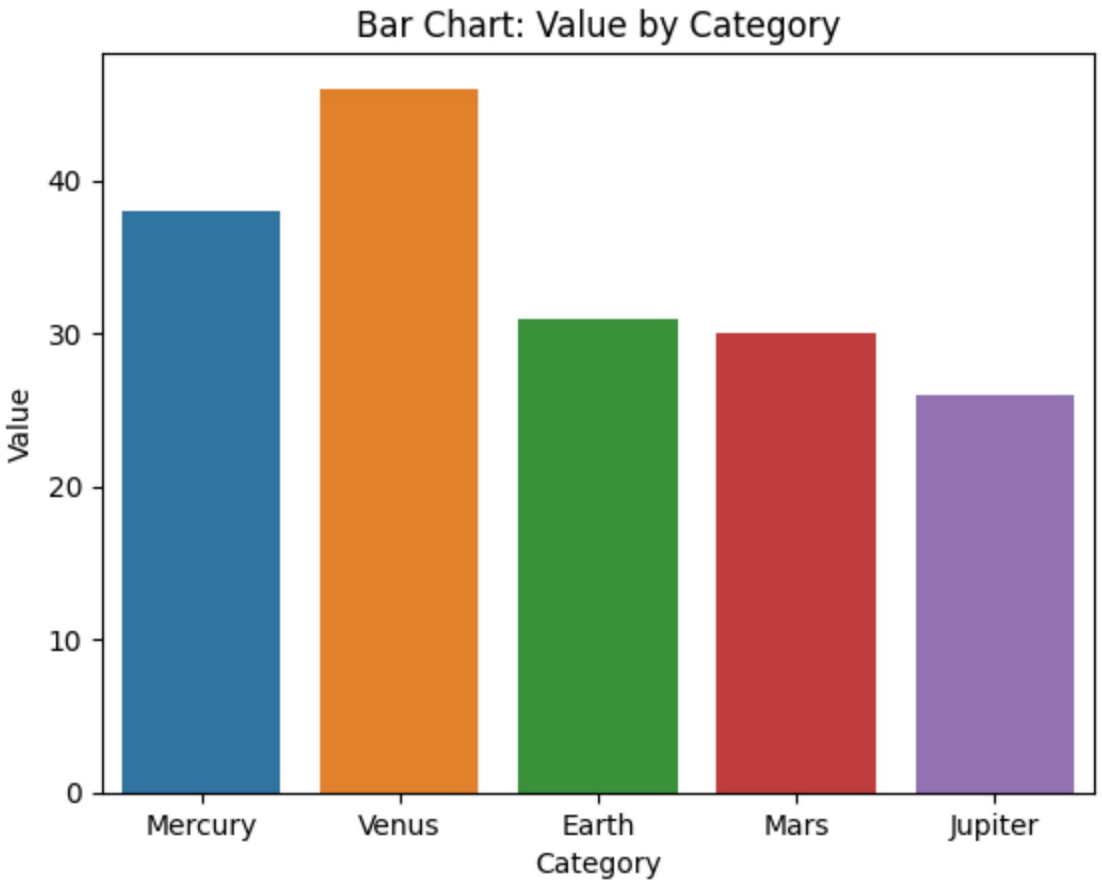
Out[9]:

	Category	Value
0	Mercury	38
1	Venus	46
2	Earth	31
3	Mars	30
4	Jupiter	26

In [10]:
```
sns.barplot(
    data=bc_002_single,
    x='Category',
    hue='Category',
    y='Value',
    palette=grouped_palette[:len(bc_002_single)]
)
plt.xlabel('Category')
plt.ylabel('Value')
```

```
plt.title('Bar Chart: Value by Category')
```

Out[10]: Text(0.5, 1.0, 'Bar Chart: Value by Category')



003 Single

In [11]:
```
bc_003_single = pd.read_csv('./bar_charts/bar_single_003.csv')

bc_003_single
```

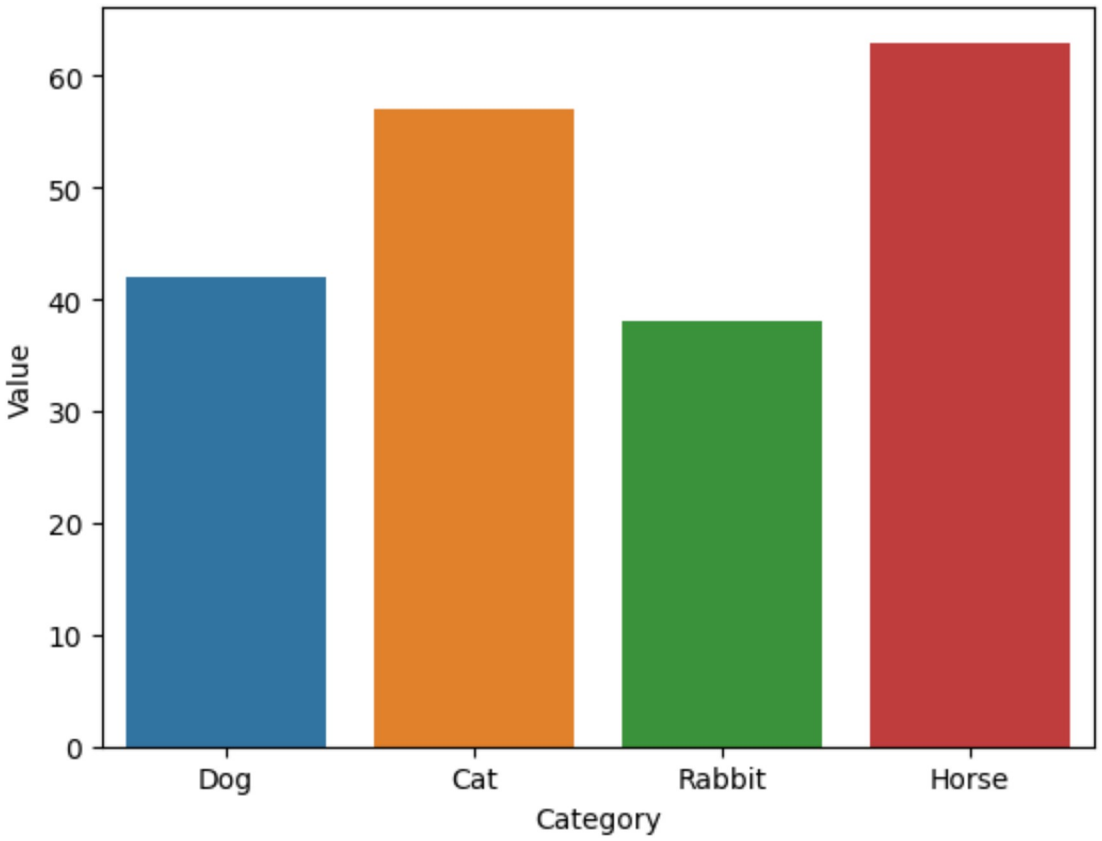
Out[11]:

	Category	Value
0	Dog	42
1	Cat	57
2	Rabbit	38
3	Horse	63

In [12]:
```
sns.barplot(
    data=bc_003_single,
    x='Category',
    hue='Category',
    y='Value',
    palette=grouped_palette[:len(bc_003_single)]
)
```

```
plt.xlabel('Category')
plt.ylabel('Value')
```

Out[12]: Text(0, 0.5, 'Value')



004 Single

In [13]:
```
bc_004_single = pd.read_csv('./bar_charts/bar_single_004.csv')

bc_004_single
```

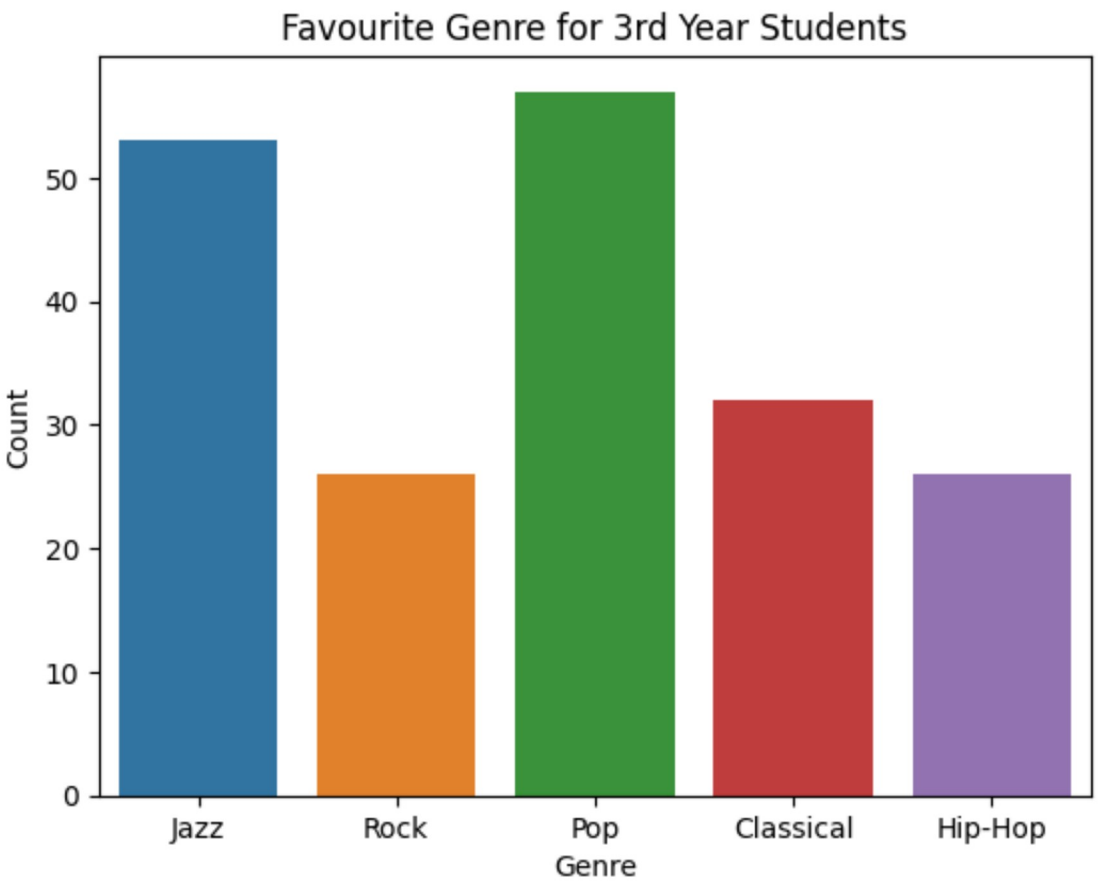
Out[13]:

	Genre	Count
0	Jazz	53
1	Rock	26
2	Pop	57
3	Classical	32
4	Hip-Hop	26

In [14]:
```
sns.barplot(
    data=bc_004_single,
    x='Genre',
    hue='Genre',
    y='Count',
    palette=grouped_palette[:len(bc_004_single)]
```

```
)
plt.xlabel('Genre')
plt.ylabel('Count')
plt.title('Favourite Genre for 3rd Year Students')
```

Out[14]: Text(0.5, 1.0, 'Favourite Genre for 3rd Year Students')



028 Grouped

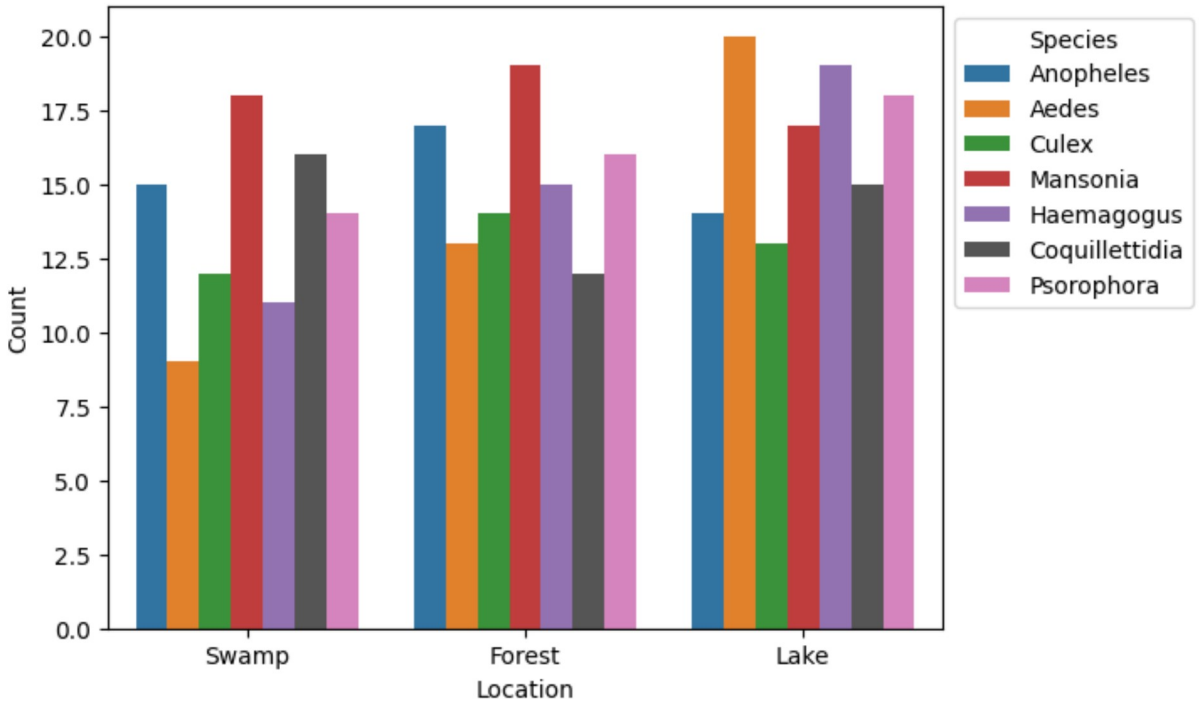
In [15]:
```
bc_028_grouped = pd.read_csv('./bar_charts/bar_grouped_028.csv')

bc_028_grouped
```

Out[15]:

	Location	Species	Count
0	Swamp	Anopheles	15
1	Swamp	Aedes	9
2	Swamp	Culex	12
3	Swamp	Mansonia	18
4	Swamp	Haemagogus	11
5	Swamp	Coquillettidia	16
6	Swamp	Psorophora	14
7	Forest	Anopheles	17
8	Forest	Aedes	13
9	Forest	Culex	14
10	Forest	Mansonia	19
11	Forest	Haemagogus	15
12	Forest	Coquillettidia	12
13	Forest	Psorophora	16
14	Lake	Anopheles	14
15	Lake	Aedes	20
16	Lake	Culex	13
17	Lake	Mansonia	17
18	Lake	Haemagogus	19
19	Lake	Coquillettidia	15
20	Lake	Psorophora	18

In [16]:
```
sns.barplot(
    data=bc_028_grouped,
    x='Location',
    y='Count',
    hue='Species',
    palette=grouped_palette[:bc_028_grouped['Species'].nunique()]  # Set palette si
)
plt.xlabel('Location')
plt.legend(loc='upper left', bbox_to_anchor=(1, 1), title='Species') # Fix the lege
```

Out[16]:
```
<matplotlib.legend.Legend at 0x203595b3d90>
```



029 Grouped

In [17]:
```
bc_029_grouped = pd.read_csv('./bar_charts/bar_grouped_029.csv')

bc_029_grouped
```

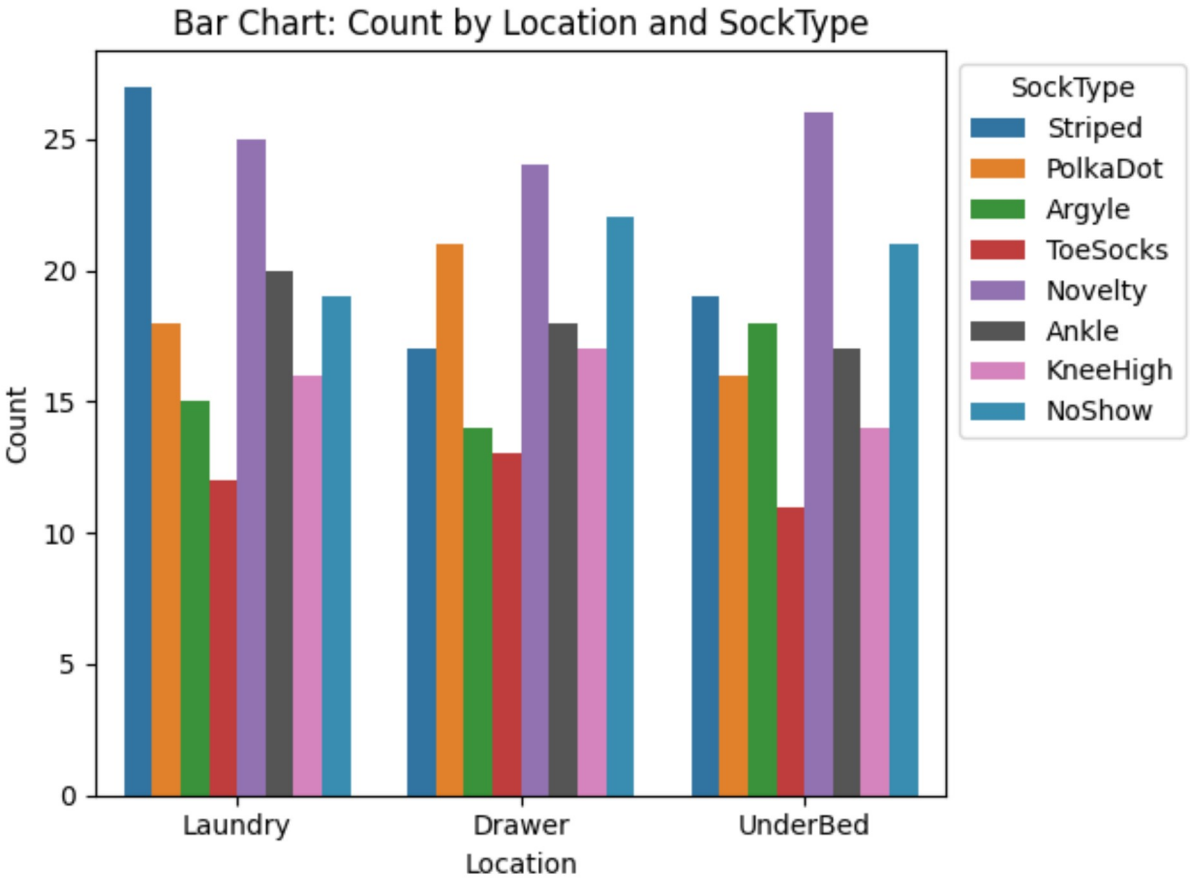
Out[17]:

	Location	SockType	Count
0	Laundry	Striped	27
1	Laundry	PolkaDot	18
2	Laundry	Argyle	15
3	Laundry	ToeSocks	12
4	Laundry	Novelty	25
5	Laundry	Ankle	20
6	Laundry	KneeHigh	16
7	Laundry	NoShow	19
8	Drawer	Striped	17
9	Drawer	PolkaDot	21
10	Drawer	Argyle	14
11	Drawer	ToeSocks	13
12	Drawer	Novelty	24
13	Drawer	Ankle	18
14	Drawer	KneeHigh	17
15	Drawer	NoShow	22
16	UnderBed	Striped	19
17	UnderBed	PolkaDot	16
18	UnderBed	Argyle	18
19	UnderBed	ToeSocks	11
20	UnderBed	Novelty	26
21	UnderBed	Ankle	17
22	UnderBed	KneeHigh	14
23	UnderBed	NoShow	21

In [18]:

```
sns.barplot(
    data=bc_029_grouped,
    x='Location',
    y='Count',
    hue='SockType',
    palette=grouped_palette  # Apply palette to the hue groups
)
plt.xlabel('Location')
plt.ylabel('Count')
plt.title('Bar Chart: Count by Location and SockType')
```

```
plt.legend(
    title='SockType',
    bbox_to_anchor=(1, 1),
    loc='upper left'
)
plt.tight_layout()
plt.show()
```



005 Single

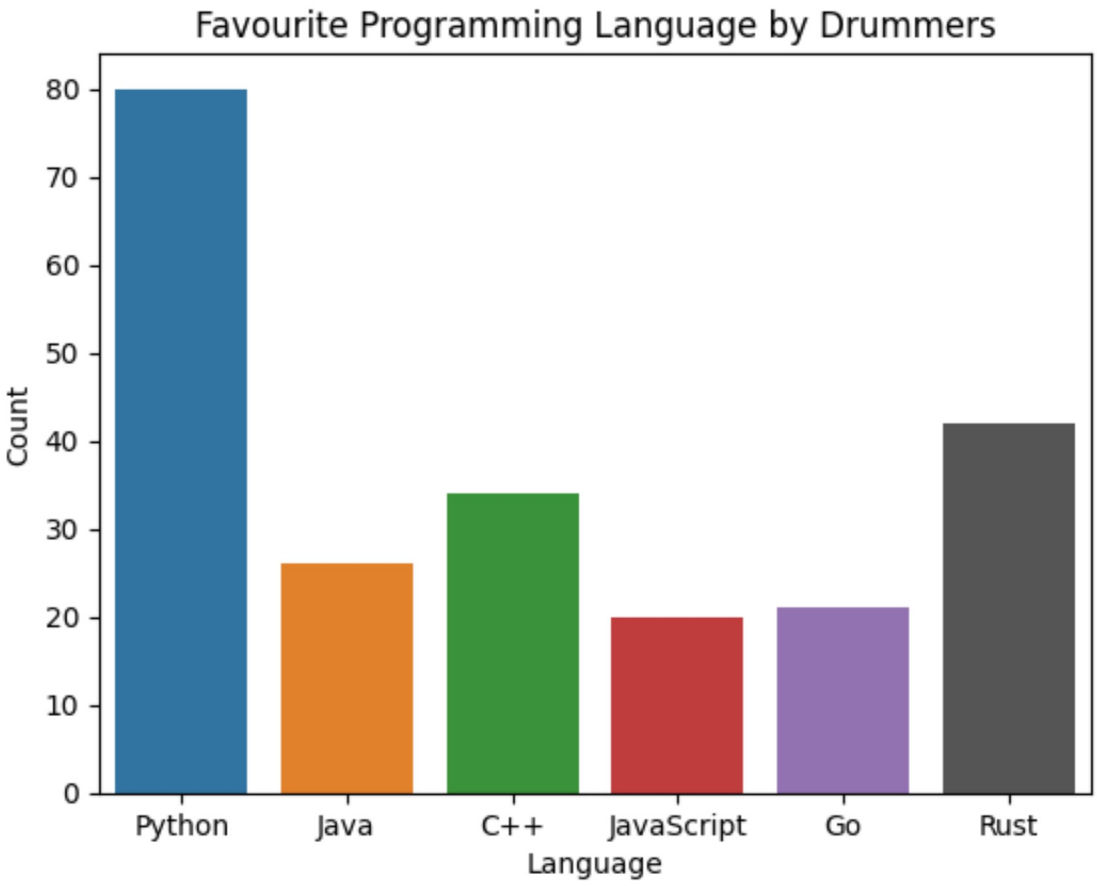
In [19]:
```
bc_005_single = pd.read_csv('./bar_charts/bar_single_005.csv')

bc_005_single
```

Out[19]:

	Language	Count
0	Python	80
1	Java	26
2	C++	34
3	JavaScript	20
4	Go	21
5	Rust	42

In [20]:
```
sns.barplot(
    data=bc_005_single,
    x='Language',
    hue='Language',
    y='Count',
    palette=grouped_palette[:len(bc_005_single)]
)
plt.xlabel('Language')
plt.title('Favourite Programming Language by Drummers')
```

Out[20]:
```
Text(0.5, 1.0, 'Favourite Programming Language by Drummers')
```



006 Single

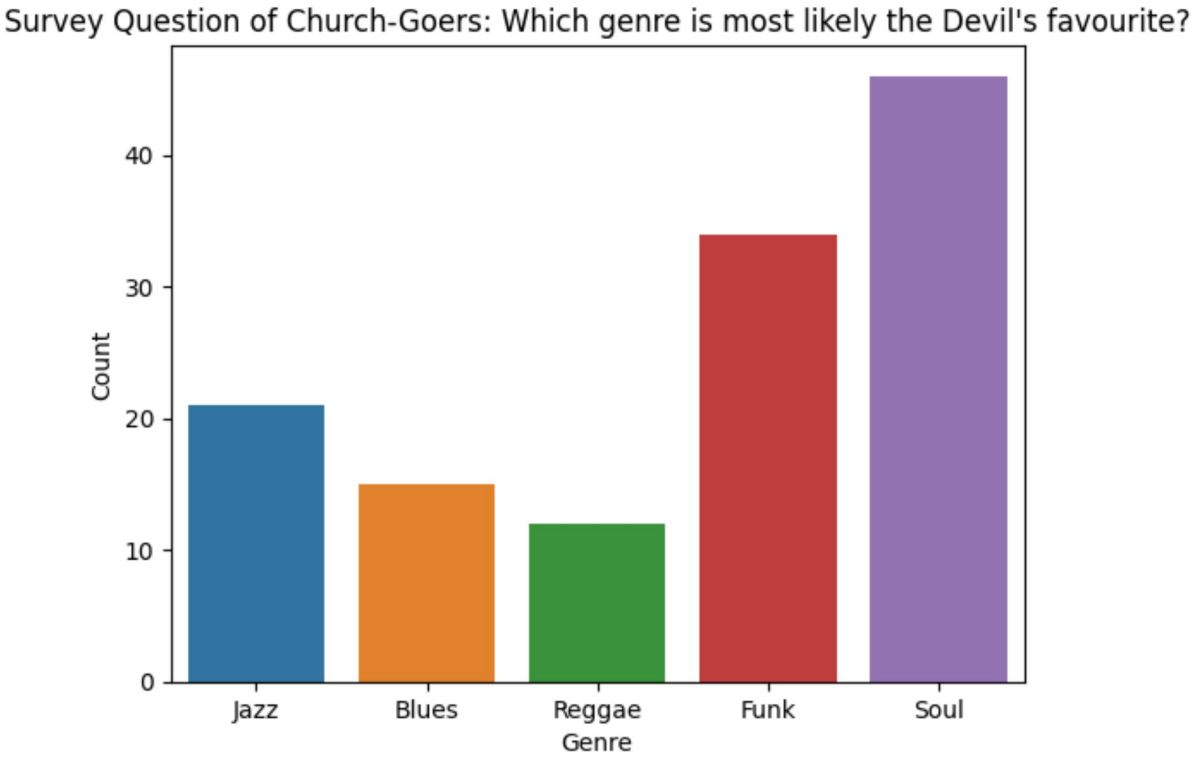
In [21]:
```
bc_006_single = pd.read_csv('./bar_charts/bar_single_006.csv')

bc_006_single
```

Out[21]:

	Genre	Count
0	Jazz	21
1	Blues	15
2	Reggae	12
3	Funk	34
4	Soul	46

In [22]:
```
sns.barplot(
    data=bc_006_single,
    x='Genre',
    hue='Genre',
    y='Count',
    palette=grouped_palette[:len(bc_006_single)]
)
plt.xlabel('Genre')
plt.title('Survey Question of Church-Goers: Which genre is most likely the Devil\'s
```

Out[22]:
```
Text(0.5, 1.0, "Survey Question of Church-Goers: Which genre is most likely the De
vil's favourite?")
```



007 Single

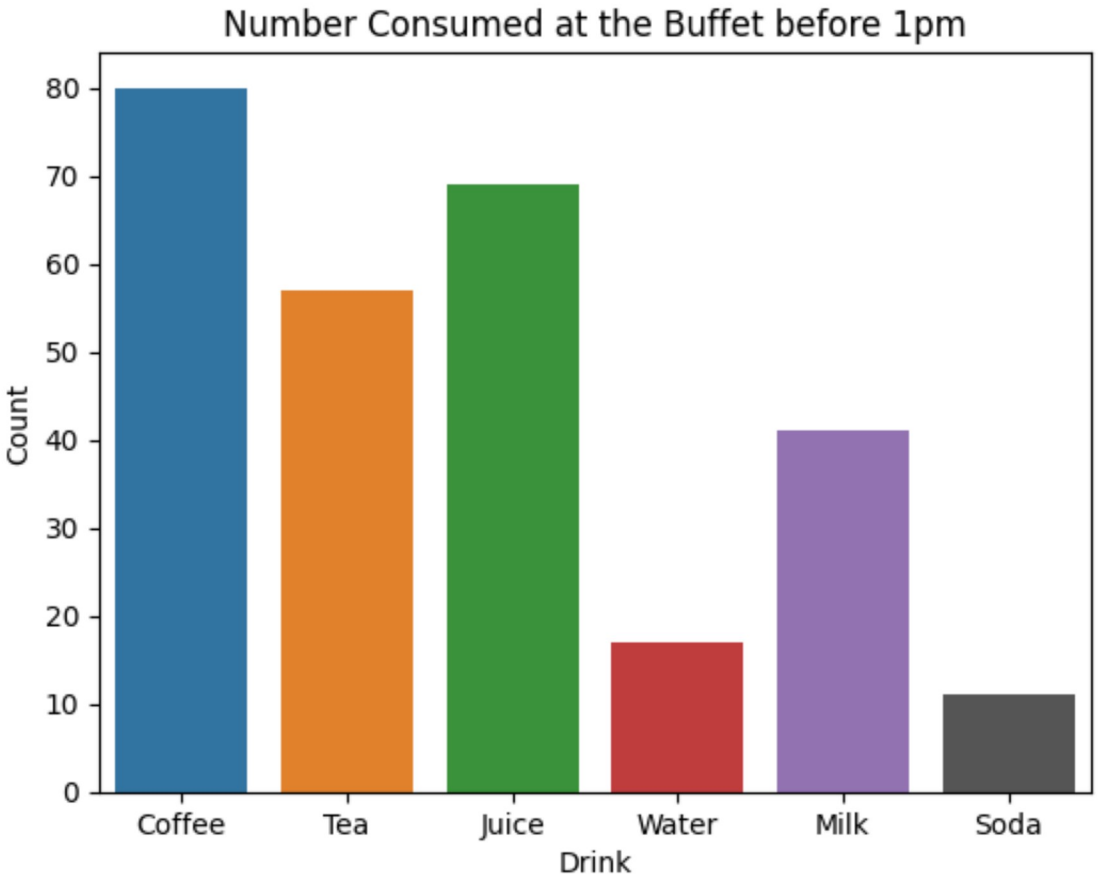
In [23]:
```
bc_007_single = pd.read_csv("bar_charts/bar_single_007.csv")

bc_007_single
```

Out[23]:

	Drink	Count
0	Coffee	80
1	Tea	57
2	Juice	69
3	Water	17
4	Milk	41
5	Soda	11

In [24]:
```
sns.barplot(
    data=bc_007_single,
    x='Drink',
    hue='Drink',
    y='Count',
    palette=grouped_palette[:len(bc_007_single)]
)
plt.xlabel('Drink')
plt.title('Number Consumed at the Buffet before 1pm')
```

Out[24]:
```
Text(0.5, 1.0, 'Number Consumed at the Buffet before 1pm')
```



008 Single

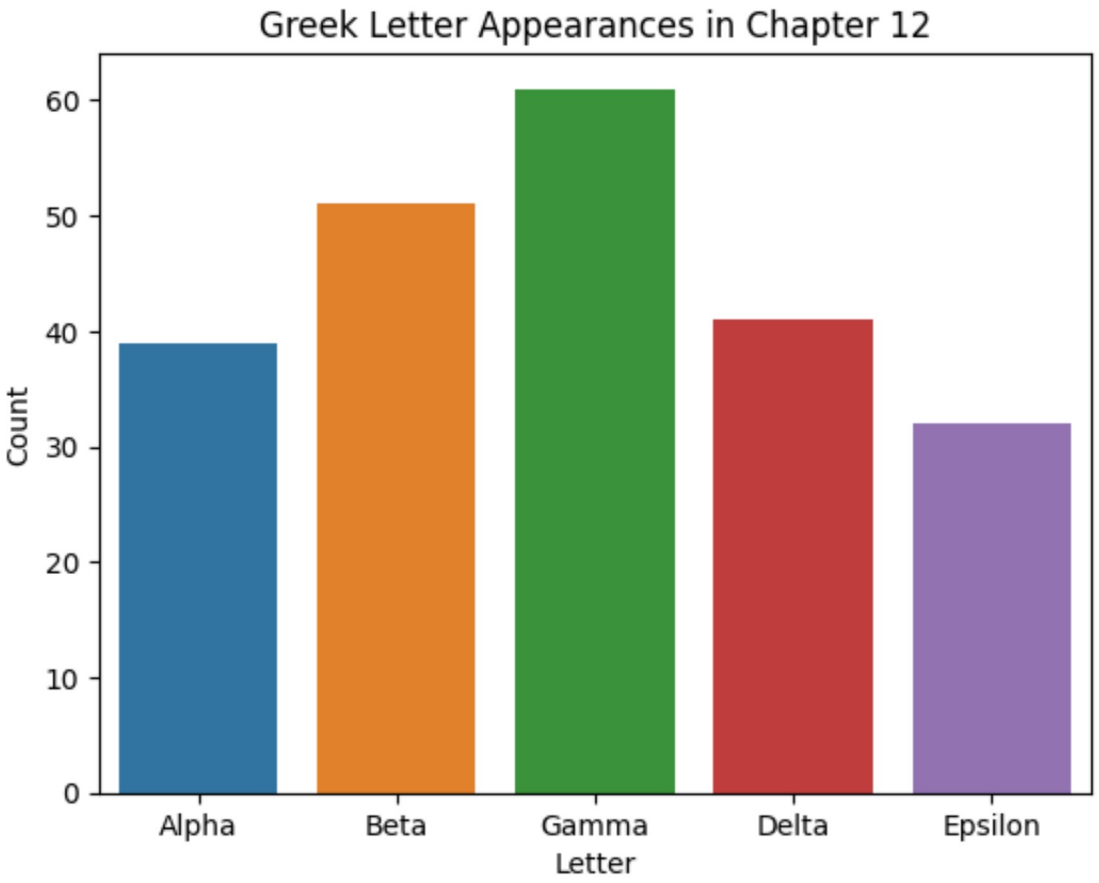
In [25]:
```
bc_008_single = pd.read_csv("bar_charts/bar_single_008.csv")

bc_008_single
```

Out[25]:

	Letter	Count
0	Alpha	39
1	Beta	51
2	Gamma	61
3	Delta	41
4	Epsilon	32

In [33]:
```
sns.barplot(
    data=bc_008_single,
    x='Letter',
    hue='Letter',
    y='Count',
    palette=grouped_palette[:len(bc_008_single)]
)
plt.xlabel('Letter')
plt.title('Greek Letter Appearances in Chapter 12')
```

Out[33]:
```
Text(0.5, 1.0, 'Greek Letter Appearances in Chapter 12')
```



009 Single

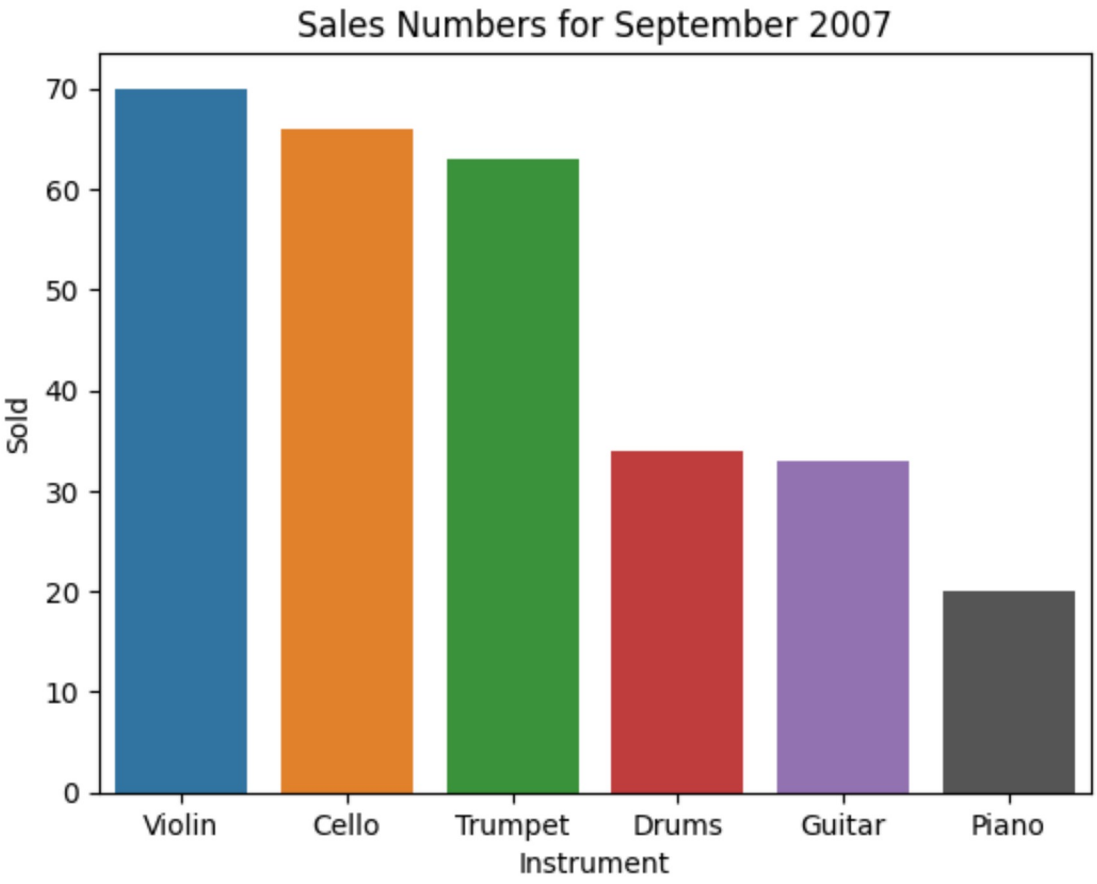
In [27]:
```
bc_009_single = pd.read_csv("bar_charts/bar_single_009.csv")

bc_009_single
```

Out[27]:

	Instrument	Sold
0	Violin	70
1	Cello	66
2	Trumpet	63
3	Drums	34
4	Guitar	33
5	Piano	20

In [28]:
```
sns.barplot(
    data=bc_009_single,
    x='Instrument',
    hue='Instrument',
    y='Sold',
    palette=grouped_palette[:len(bc_009_single)]
)
plt.xlabel('Instrument')
plt.title('Sales Numbers for September 2007')
```

Out[28]:
```
Text(0.5, 1.0, 'Sales Numbers for September 2007')
```



010 Single

In [29]:
```
bc_010_single = pd.read_csv("bar_charts/bar_single_010.csv")

bc_010_single
```

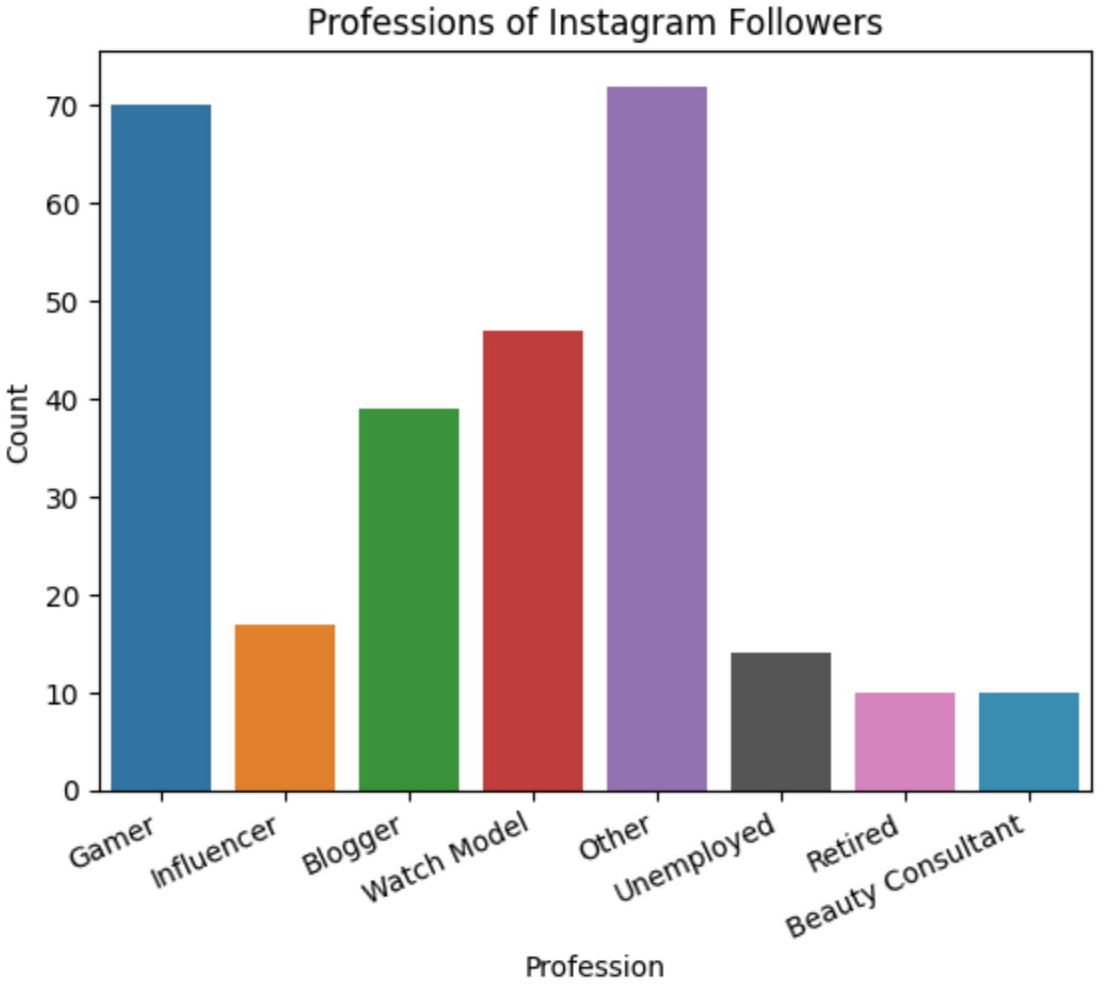
Out[29]:

	Profession	Count
0	Gamer	70
1	Influencer	17
2	Blogger	39
3	Watch Model	47
4	Other	72
5	Unemployed	14
6	Retired	10
7	Beauty Consultant	10

In [30]:
```
sns.barplot(
    data=bc_010_single,
    x='Profession',
```

```
    hue='Profession',
    y='Count',
    palette=grouped_palette[:len(bc_010_single)]
)
plt.xlabel('Profession')
plt.title('Professions of Instagram Followers')
plt.xticks(rotation=25, ha='right')
```

Out[30]:
```
([0, 1, 2, 3, 4, 5, 6, 7],
 [Text(0, 0, 'Gamer'),
  Text(1, 0, 'Influencer'),
  Text(2, 0, 'Blogger'),
  Text(3, 0, 'Watch Model'),
  Text(4, 0, 'Other'),
  Text(5, 0, 'Unemployed'),
  Text(6, 0, 'Retired'),
  Text(7, 0, 'Beauty Consultant')])
```



011

In [31]:
```
bc_011_single = pd.read_csv("bar_charts/bar_single_011.csv")

bc_011_single
```

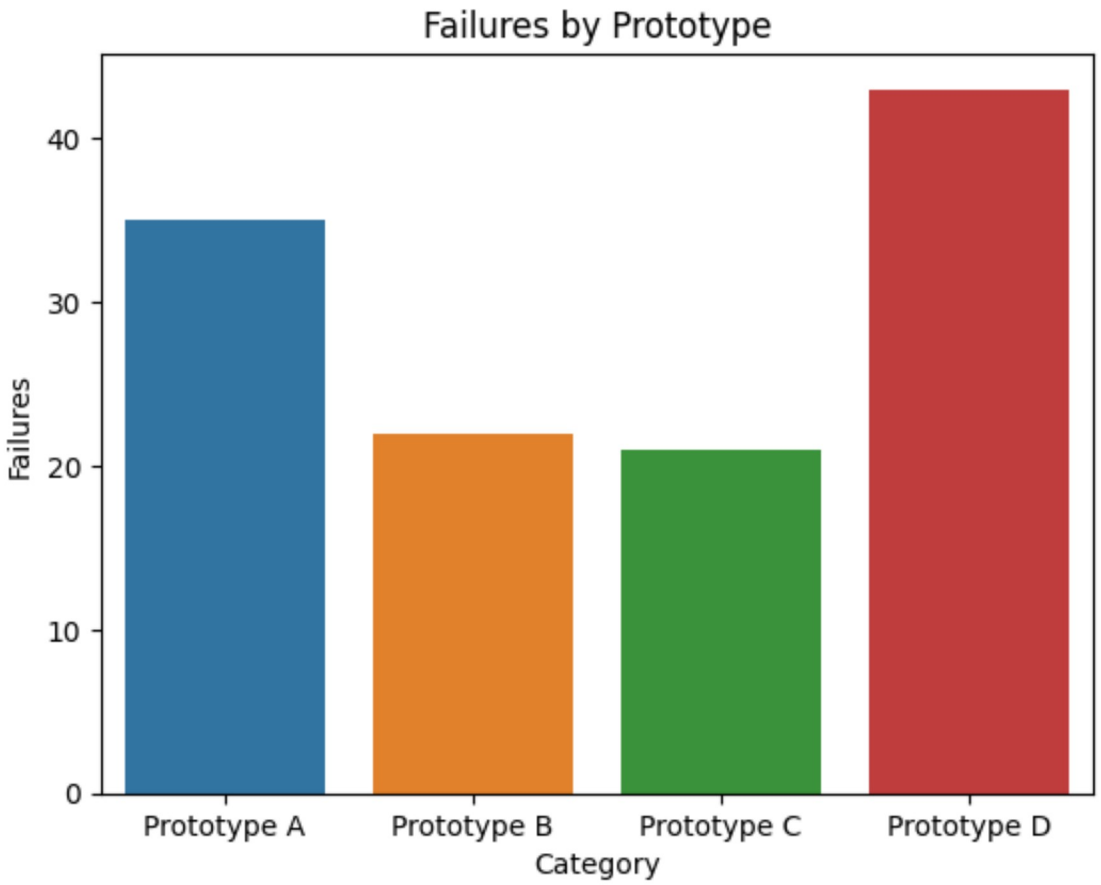
Out[31]:

	Category	Failures
0	Prototype A	35
1	Prototype B	22
2	Prototype C	21
3	Prototype D	43

011 Single

In [32]:
```
sns.barplot(
    data=bc_011_single,
    x='Category',
    hue='Category',
    y='Failures',
    palette=grouped_palette[:len(bc_011_single)]
)
plt.xlabel('Category')
plt.title('Failures by Prototype')
```

Out[32]: Text(0.5, 1.0, 'Failures by Prototype')

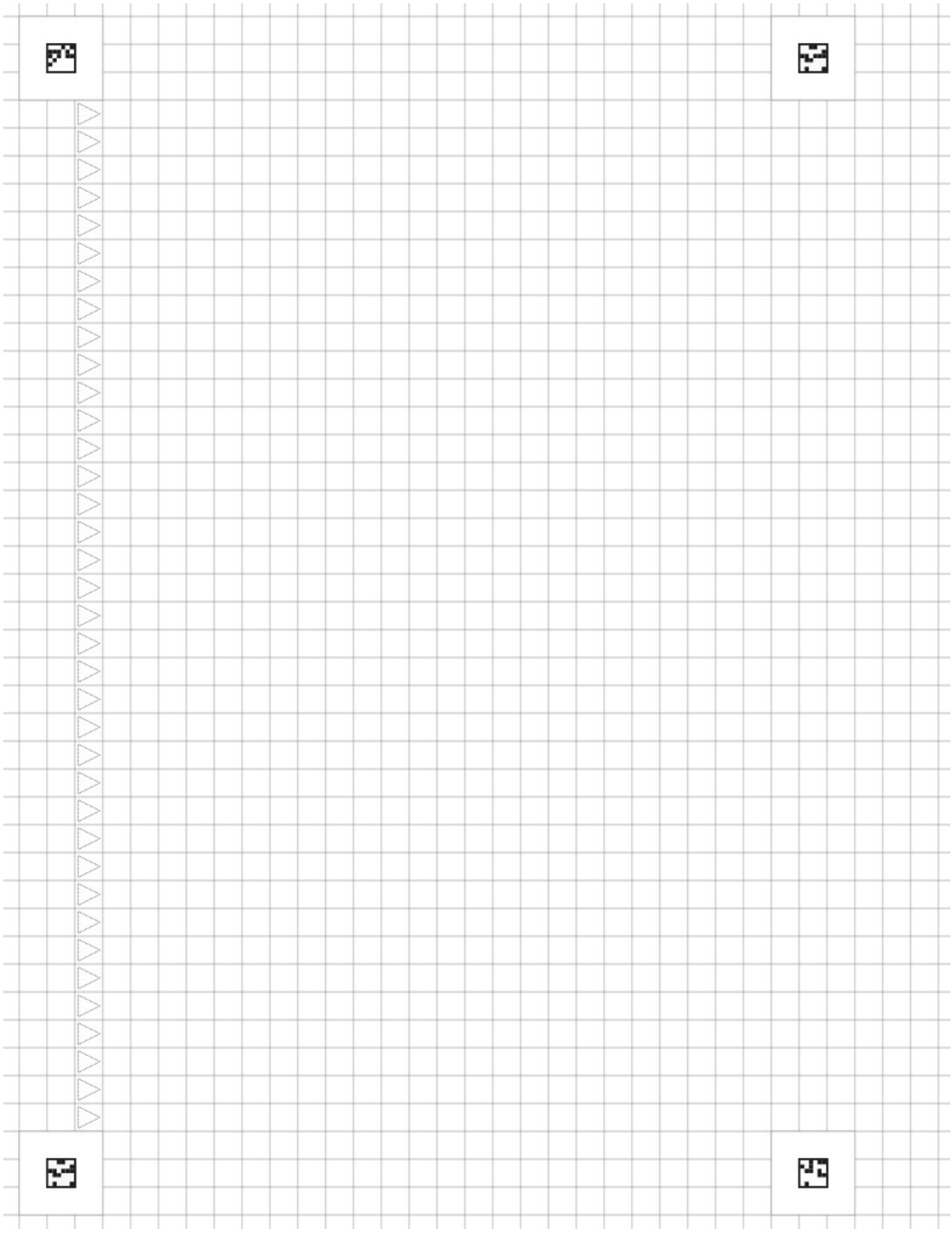


A2 Image Normalization

Each image that is taken with the camera phone will need to be normalized.

In [4]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Page Zero Display



Page One Display

Calibration Page One Display

Other Page Variants

Now that the calibration cubes have been assessed, we turn our attention to another form of

noise.

- Dot Matrix
- Lined
- Blank
- Hex

Thoughts

It is not lost on me that there is an infinite amount of paper types that one could imagine in order to convey ideas. Personally, the type for which I am the most nostalgic is dot matrix. An example below:

Dot Matrix Page One

Describing the Calibration Features

Quick Calibration Squares: `6 x 6` square in every corner. Each cell in the square is `0.975mm+/- 0.500mm` wide. There are 4 distinct styles:

- Top Left and Bottom Right QC square are the same per page (Page 0: Rear; Page 1; Front)
  - see:
    - Page Zero and One Bottom Right
    - Page Zero and One Top Left
- The Top Right and Bottom Left of each page is the same
  - see:
    - Page Zero Top Right and Bottom Left
    - Page One Top Right and Bottom Left

QC Square Envelope: Each QC square has an evelope around it (as seen in any appended images of the sqaures). The envelope is `0.75in`.

Left Margin Arrows Each page has directional triangles in a straight line along the Left Margin of the drawing area. There are `37` of them pointing with the tips towards the drawing area. The ratio of the triangle is such that the left edge, which is perpendicular to the column direction is `0.75in - 2mm` and the two other edges, which are congruent, are also `0.75in - 2mm`. The triangles are centered within each cell within each column and row.

Left Margin

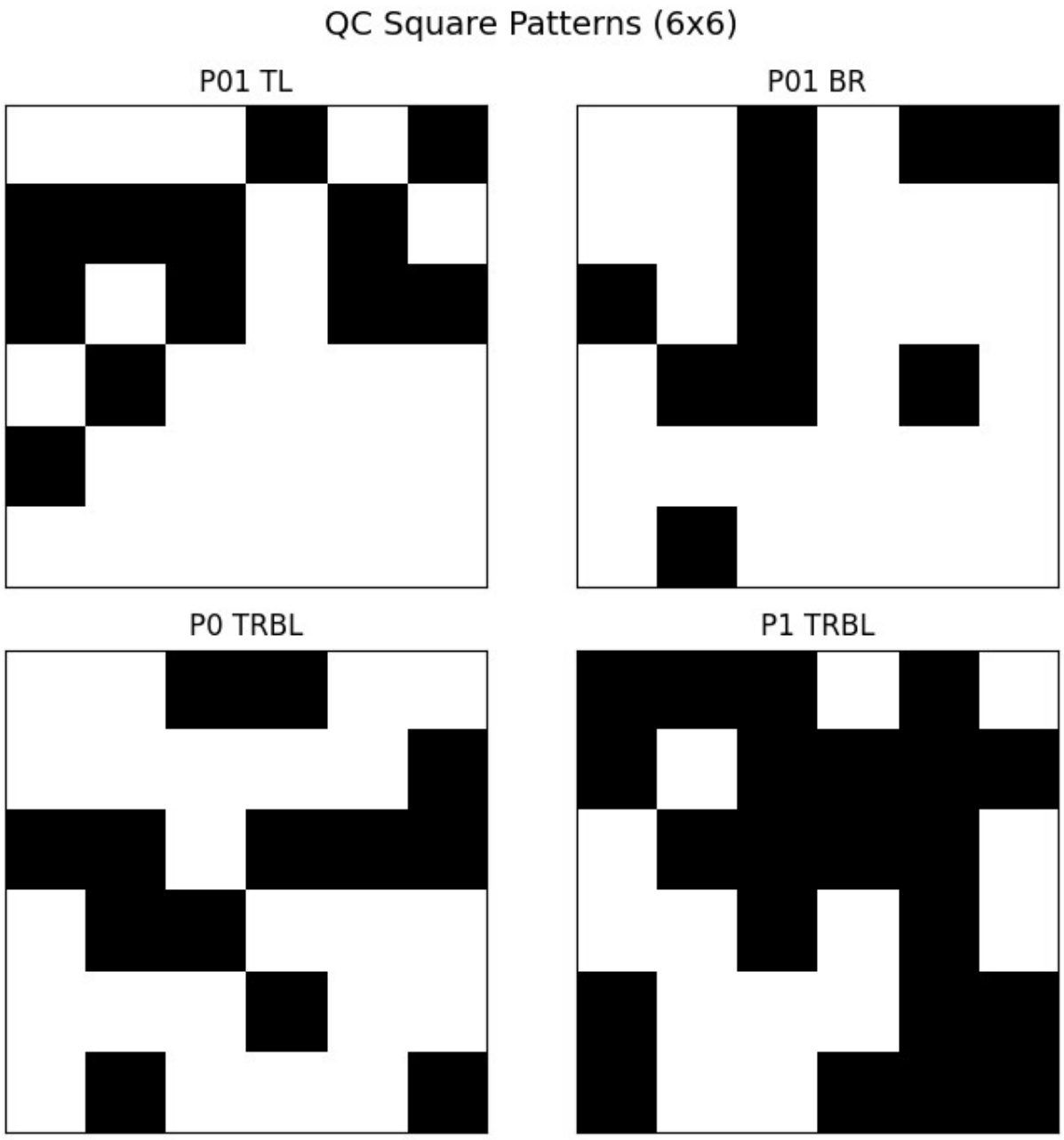
Matrix Representations of QC Squares

In [5]:

```
# 0: black
# 1: white

n = [ # Empty Matrix
    [0, 0, 0, 0, 0, 0], # Row 0
    [0, 0, 0, 0, 0, 0], # Row 1
    [0, 0, 0, 0, 0, 0], # Row 2
    [0, 0, 0, 0, 0, 0], # Row 3
    [0, 0, 0, 0, 0, 0], # Row 4
    [0, 0, 0, 0, 0, 0]  # Row 5
]


# Page 0 and 1, Top Left
p01_tl = [
    [1, 1, 1, 0, 1, 0], # Row 0
    [0, 0, 0, 1, 0, 1], # Row 1
    [0, 1, 0, 1, 0, 0], # Row 2
    [1, 0, 1, 1, 1, 1], # Row 3
    [0, 1, 1, 1, 1, 1], # Row 4
    [1, 1, 1, 1, 1, 1] # Row 5
]

# Page 0 and 1, Bottom Right
p01_br = [
    [1, 1, 0, 1, 0, 0], # Row 0
    [1, 1, 0, 1, 1, 1], # Row 1
    [0, 1, 0, 1, 1, 1], # Row 2
    [1, 0, 0, 1, 0, 1], # Row 3
    [1, 1, 1, 1, 1, 1], # Row 4
    [1, 0, 1, 1, 1, 1]  # Row 5
]

# Page 0, Top Right and Bottom Left
p0_trbl = [
    [1, 1, 0, 0, 1, 1], # Row 0
    [1, 1, 1, 1, 1, 0], # Row 1
    [0, 0, 1, 0, 0, 0], # Row 2
    [1, 0, 0, 1, 1, 1], # Row 3
    [1, 1, 1, 0, 1, 1], # Row 4
    [1, 0, 1, 1, 1, 0]  # Row 5
]

# Page 1, Top Right and Bottom Left

p1_trbl = [
    [0, 0, 0, 1, 0, 1], # Row 0
    [0, 1, 0, 0, 0, 0], # Row 1
    [1, 0, 0, 0, 0, 1], # Row 2
    [1, 1, 0, 1, 0, 1], # Row 3
    [0, 1, 1, 1, 0, 0], # Row 4
    [0, 1, 1, 0, 0, 0] # Row 5
]
```

Visual Representation

In [6]:
```
import matplotlib.pyplot as plt

# Show graph of what the QC squares look like

qc_patterns = {
    'p01_tl': np.array(p01_tl),
    'p01_br': np.array(p01_br),
    'p0_trbl': np.array(p0_trbl),
    'p1_trbl': np.array(p1_trbl)
}

fig, axes = plt.subplots(2, 2, figsize=(7, 7))
for ax, (name, pattern) in zip(axes.flatten(), qc_patterns.items()):
    ax.imshow(pattern, cmap='gray', vmin=0, vmax=1, interpolation='nearest')
    ax.set_title(name.replace('_', ' ').upper())
    ax.set_xticks([])
    ax.set_yticks([])
    ax.grid(False)

plt.suptitle('QC Square Patterns (6x6)', fontsize=14)
plt.tight_layout()
plt.show()
```



Thought

I do not approve of the imperial system, but then I didn't make the paper. In future work, I would prefer a metric version of the medium. This is something that I think would take no longer than 30 minutes to design.

Normalization Script

Test Image

Degenerate points

points that do not form a valid geometric shape; they are collinear, overlapping, or not distinct enough to define a proper quadrilateral for transformations.

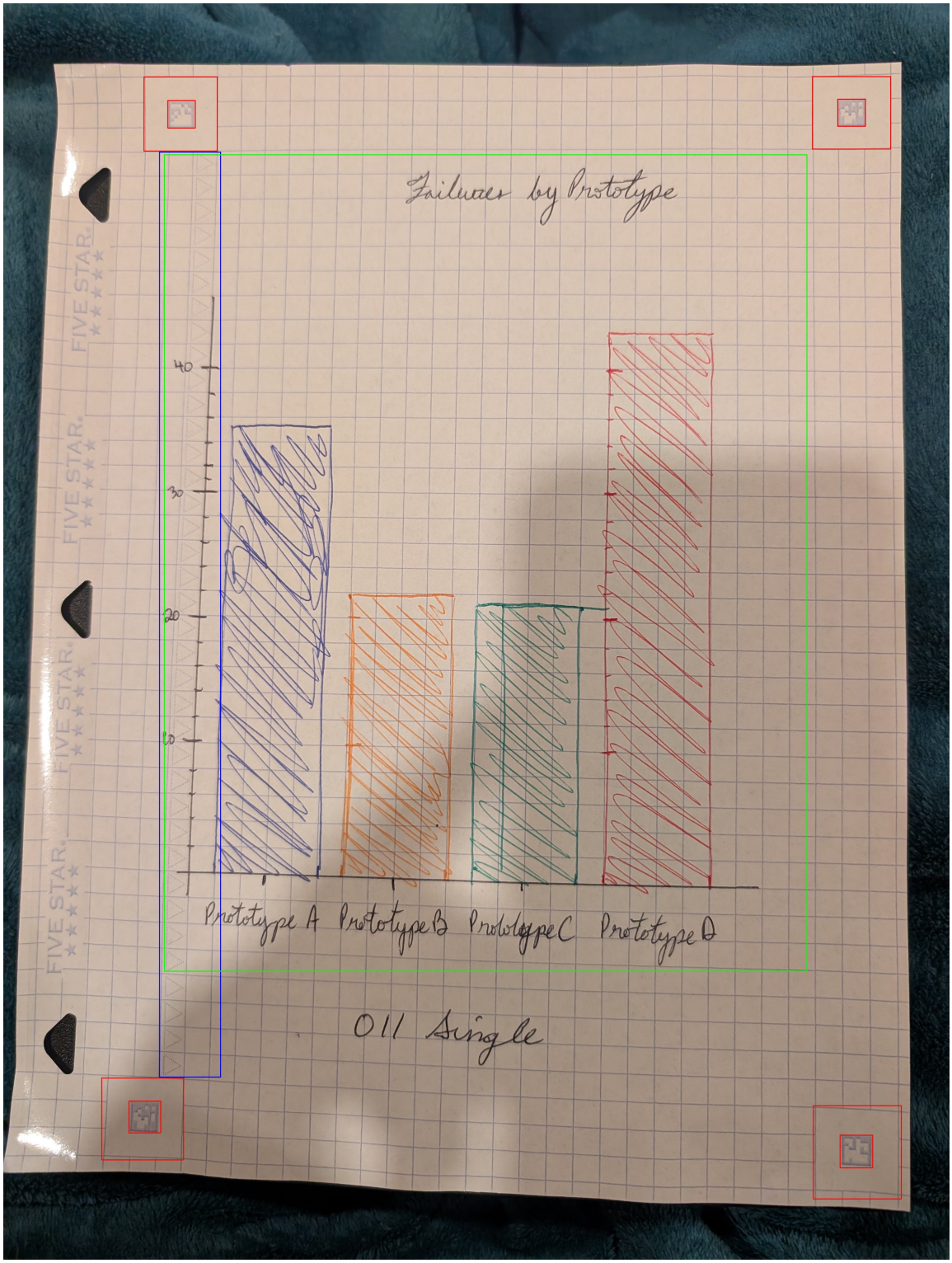
011 Single

Identifying Page Features

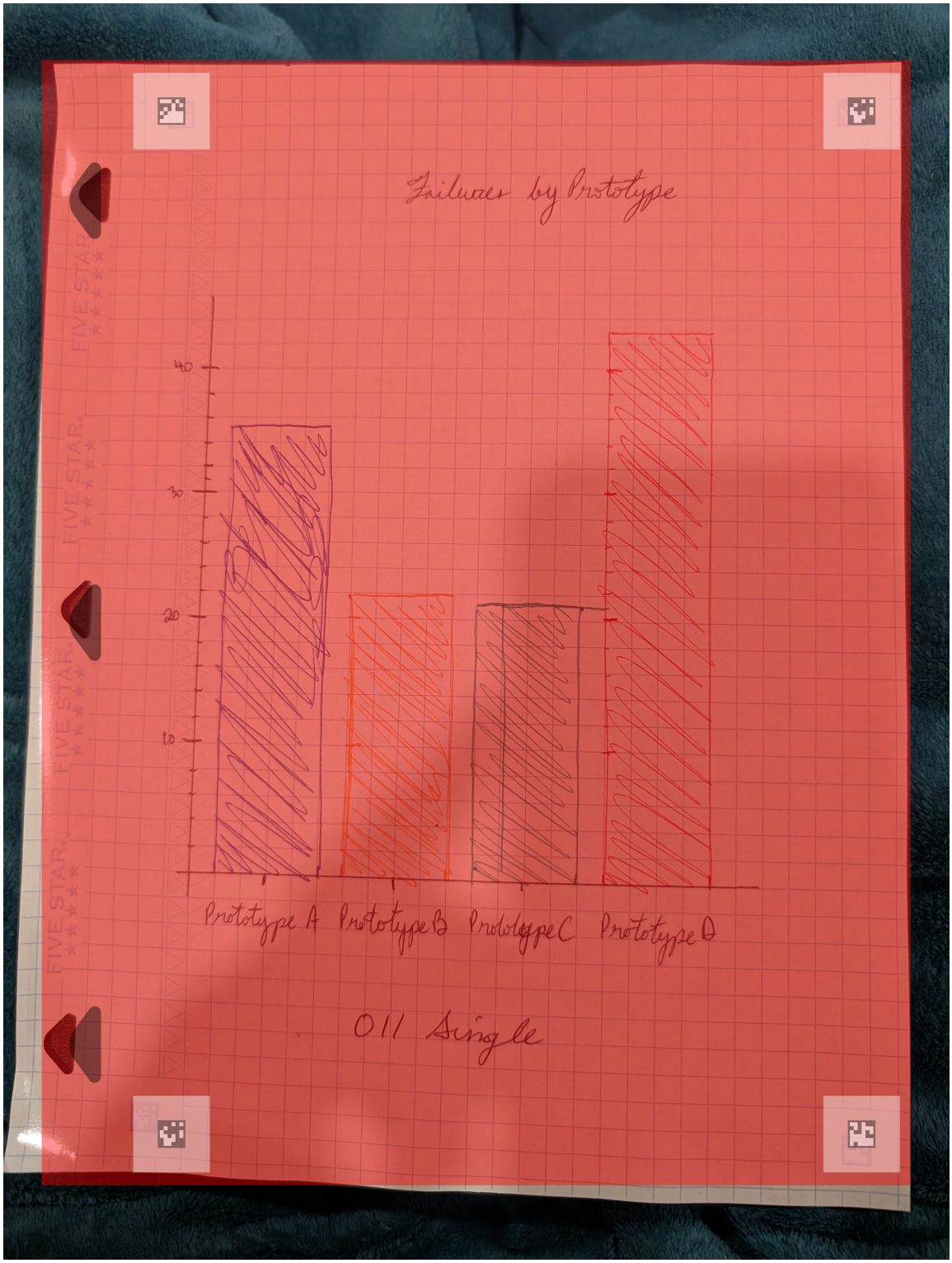
Red: QC squares

Blue: Triangles

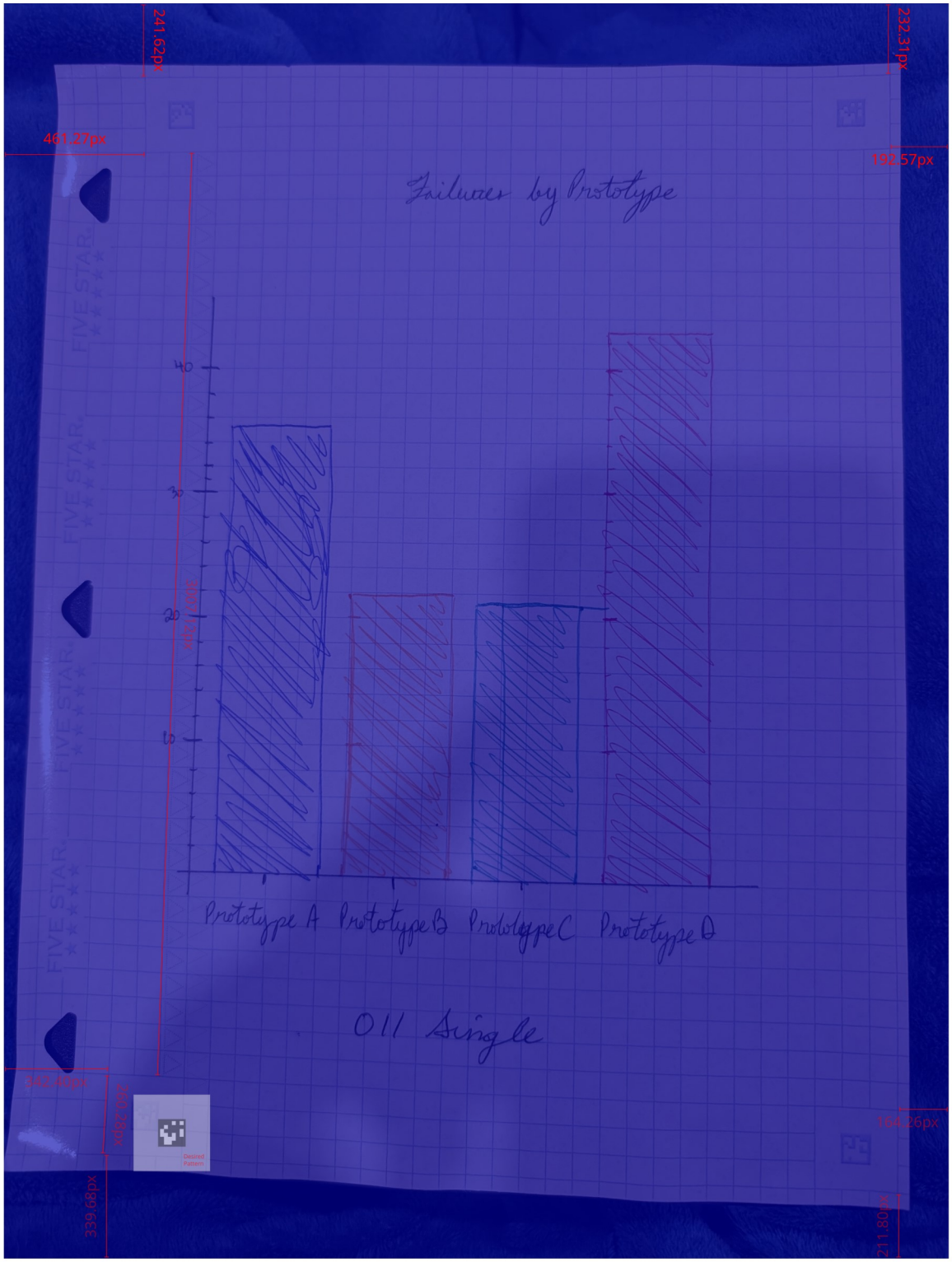
Green: Chart Area



Desired Features



Page Dimensions



In [7]:
```
## Test Image
import PIL
import os
import cv2
import numpy as np
```

In [8]:
```
def detect_qc_square(image, corner='auto', search_region_size=0.2, min_square_area=
    """
    Detect QC square in the image.
```

```
    Parameters:
    -----------
    image : numpy.ndarray
        RGB image (H, W, 3) or BGR if from cv2.imread
    corner : str, optional
        Which corner to search: 'tl', 'tr', 'bl', 'br', or 'auto' (search all)
        Default: 'auto'
    search_region_size : float
        Fraction of image dimensions to search in corner (0.0-1.0)
        Default: 0.2 (top/bottom/left/right 20%)
    min_square_area : int
        Minimum area for detected square (in pixels)
        Default: 100

    Returns:
    --------
    dict or list of dicts
        If corner='auto': list of dicts, one per detected QC square
        Otherwise: single dict with keys:
            - 'bbox': (x, y, width, height) bounding box
            - 'corner': detected corner position ('tl', 'tr', 'bl', 'br')
            - 'pattern': extracted 6x6 pattern (numpy array)
            - 'pattern_match': matched pattern name (if successful)
            - 'confidence': match confidence (0-1)
    """
    # Convert BGR to RGB if needed (cv2 loads as BGR)
    if len(image.shape) == 3:
        # Check if it's likely BGR by comparing first/last channels
        # Or just convert if from cv2.imread
        if image.dtype == np.uint8:
            rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        else:
            rgb_image = image.copy()
    else:
        raise ValueError("Image must be RGB/BGR (3 channels)")

    h, w = rgb_image.shape[:2]

    # Define search regions for each corner
    search_regions = {
        'tl': (0, 0, int(w * search_region_size), int(h * search_region_size)),
        'tr': (int(w * (1 - search_region_size)), 0, w, int(h * search_region_size)
        'bl': (0, int(h * (1 - search_region_size)), int(w * search_region_size), h
        'br': (int(w * (1 - search_region_size)), int(h * (1 - search_region_size))
    }

    # Convert to grayscale for detection
    gray = cv2.cvtColor(rgb_image, cv2.COLOR_RGB2GRAY)

    def detect_in_region(region_bbox, corner_name):
        """Detect QC square in a specific region."""
        rx, ry, rw, rh = region_bbox
        region_gray = gray[ry:rh, rx:rw]

        if region_gray.size == 0:
```

```
            return None

        # Enhance contrast for better detection
        clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
        region_enhanced = clahe.apply(region_gray)

        # Adaptive threshold to find edges/boundaries
        binary = cv2.adaptiveThreshold(region_enhanced, 255,
                                       cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                                       cv2.THRESH_BINARY_INV, 11, 2)

        # Find contours
        contours, _ = cv2.findContours(binary, cv2.RETR_EXTERNAL,
                                       cv2.CHAIN_APPROX_SIMPLE)

        # Filter for square-like contours
        square_candidates = []
        region_area = region_gray.shape[0] * region_gray.shape[1]
        min_area_pixels = max(min_square_area, region_area * 0.01)  # At least 1% o

        for contour in contours:
            area = cv2.contourArea(contour)
            if area < min_area_pixels:
                continue

            # Approximate polygon
            peri = cv2.arcLength(contour, True)
            approx = cv2.approxPolyDP(contour, 0.02 * peri, True)

            # Check if roughly square (4 corners)
            if len(approx) >= 4:
                # Get bounding rect
                x_rect, y_rect, w_rect, h_rect = cv2.boundingRect(contour)

                # Check aspect ratio (should be roughly square)
                aspect_ratio = float(w_rect) / h_rect if h_rect > 0 else 0
                if 0.7 < aspect_ratio < 1.3:  # Allow some tolerance
                    # Check solidity (filled vs outline)
                    solidity = area / (w_rect * h_rect) if (w_rect * h_rect) > 0 el

                    square_candidates.append({
                        'contour': contour,
                        'area': area,
                        'bbox_local': (x_rect, y_rect, w_rect, h_rect),
                        'solidity': solidity,
                        'aspect_ratio': aspect_ratio
                    })

        if not square_candidates:
            return None

        # Sort by area (largest first) and prefer more square-like shapes
        square_candidates.sort(key=lambda x: x['area'] * x['solidity'], reverse=Tru
        best_candidate = square_candidates[0]

        # Convert local coordinates to global image coordinates
```

```
        x_local, y_local, w_local, h_local = best_candidate['bbox_local']
        bbox_global = (rx + x_local, ry + y_local, w_local, h_local)

        # Extract pattern from this region
        try:
            pattern = extract_pattern_from_rgb(rgb_image, bbox_global, envelope_mar

            # Match pattern against known patterns
            pattern_match, confidence = match_pattern(pattern, {
                'p01_tl': np.array(p01_tl),
                'p01_br': np.array(p01_br),
                'p0_trbl': np.array(p0_trbl),
                'p1_trbl': np.array(p1_trbl)
            })

            return {
                'bbox': bbox_global,
                'corner': corner_name,
                'pattern': pattern,
                'pattern_match': pattern_match,
                'confidence': confidence,
                'area': best_candidate['area'],
                'solidity': best_candidate['solidity']
            }
        except Exception as e:
            print(f"Error extracting pattern for {corner_name}: {e}")
            return {
                'bbox': bbox_global,
                'corner': corner_name,
                'pattern': None,
                'pattern_match': None,
                'confidence': 0.0,
                'error': str(e)
            }

    # Search in specified corner(s)
    if corner == 'auto':
        results = []
        for corner_name, region_bbox in search_regions.items():
            result = detect_in_region(region_bbox, corner_name)
            if result:
                results.append(result)
        return results
    else:
        if corner not in search_regions:
            raise ValueError(f"Corner must be one of: {list(search_regions.keys())}
        return detect_in_region(search_regions[corner], corner)

def match_pattern(extracted_pattern, known_patterns):
    """
    Match extracted 6x6 pattern against known patterns.

    Parameters:
    -----------
    extracted_pattern : numpy.ndarray
        6x6 binary pattern
```

```
    known_patterns : dict
        Dictionary of pattern_name -> pattern_array

    Returns:
    --------
    tuple: (best_match_name, confidence_score)
    """
    if extracted_pattern is None:
        return None, 0.0

    best_match = None
    best_score = 0.0

    for name, ref_pattern in known_patterns.items():
        # Calculate similarity (simple matching)
        matches = np.sum(extracted_pattern == ref_pattern)
        similarity = matches / 36.0  # 36 cells total (6x6)

        if similarity > best_score:
            best_score = similarity
            best_match = name

    return best_match, best_score
```

In [9]:

```
def extract_pattern_from_rgb(rgb_image, bbox, envelope_margin=0.1):
    """
    Extract 6x6 binary pattern from RGB image.

    envelope_margin: percentage of bbox to use as margin for envelope
    """
    x, y, w, h = bbox

    # Crop region
    qc_region_rgb = rgb_image[y:y+h, x:x+w]

    # Account for the 0.75in envelope - extract inner square
    margin = int(min(w, h) * envelope_margin)
    inner_region_rgb = qc_region_rgb[margin:h-margin, margin:w-margin]

    # Convert to grayscale
    inner_gray = cv2.cvtColor(inner_region_rgb, cv2.COLOR_RGB2GRAY)

    # Get dimensions
    grid_h, grid_w = inner_gray.shape

    # Divide into 6x6 cells
    cell_h = grid_h // 6
    cell_w = grid_w // 6

    # Extract binary pattern
    pattern_6x6 = np.zeros((6, 6), dtype=int)

    for i in range(6):
        for j in range(6):
            # Sample cell center region (avoid edges between cells)
            y_start = i * cell_h + cell_h // 4
```

```
            y_end = (i + 1) * cell_h - cell_h // 4
            x_start = j * cell_w + cell_w // 4
            x_end = (j + 1) * cell_w - cell_w // 4

            if y_end > y_start and x_end > x_start:
                cell_region = inner_gray[y_start:y_end, x_start:x_end]

                # Threshold: mean value determines if cell is black or white
                # Black cells (filled) have low pixel values
                # White cells (empty) have high pixel values
                mean_value = np.mean(cell_region)

                # Adaptive threshold based on local image statistics
                # If mean is below 128 (or use Otsu's method), it's black (filled)
                if mean_value < 128:
                    pattern_6x6[i, j] = 0  # black/filled
                else:
                    pattern_6x6[i, j] = 1  # white/empty

    return pattern_6x6
```

Test

In [10]:
```
# Load image
image_path = 'hand_drawn_notes/bc_011_single-000.jpg'
image = cv2.imread(image_path)  # Returns BGR

# Detect all QC squares automatically
results = detect_qc_square(image, corner='auto', min_square_area=25)

# Process results
for result in results:
    print(f"Found QC square in {result['corner']} corner")
    print(f"  Bounding box: {result['bbox']}")
    if result['pattern_match']:
        print(f"  Matched pattern: {result['pattern_match']}")
        print(f"  Confidence: {result['confidence']:.2%}")
```

```
Found QC square in tr corner
  Bounding box: (2457, 266, 615, 550)
  Matched pattern: p01_br
  Confidence: 72.22%
Found QC square in bl corner
  Bounding box: (20, 3264, 594, 639)
  Matched pattern: p01_tl
  Confidence: 72.22%
Found QC square in br corner
  Bounding box: (2457, 3264, 615, 685)
  Matched pattern: p01_br
  Confidence: 66.67%
```

Visualisation

In [11]:
```
import matplotlib.pyplot as plt
import matplotlib.patches as patches
```

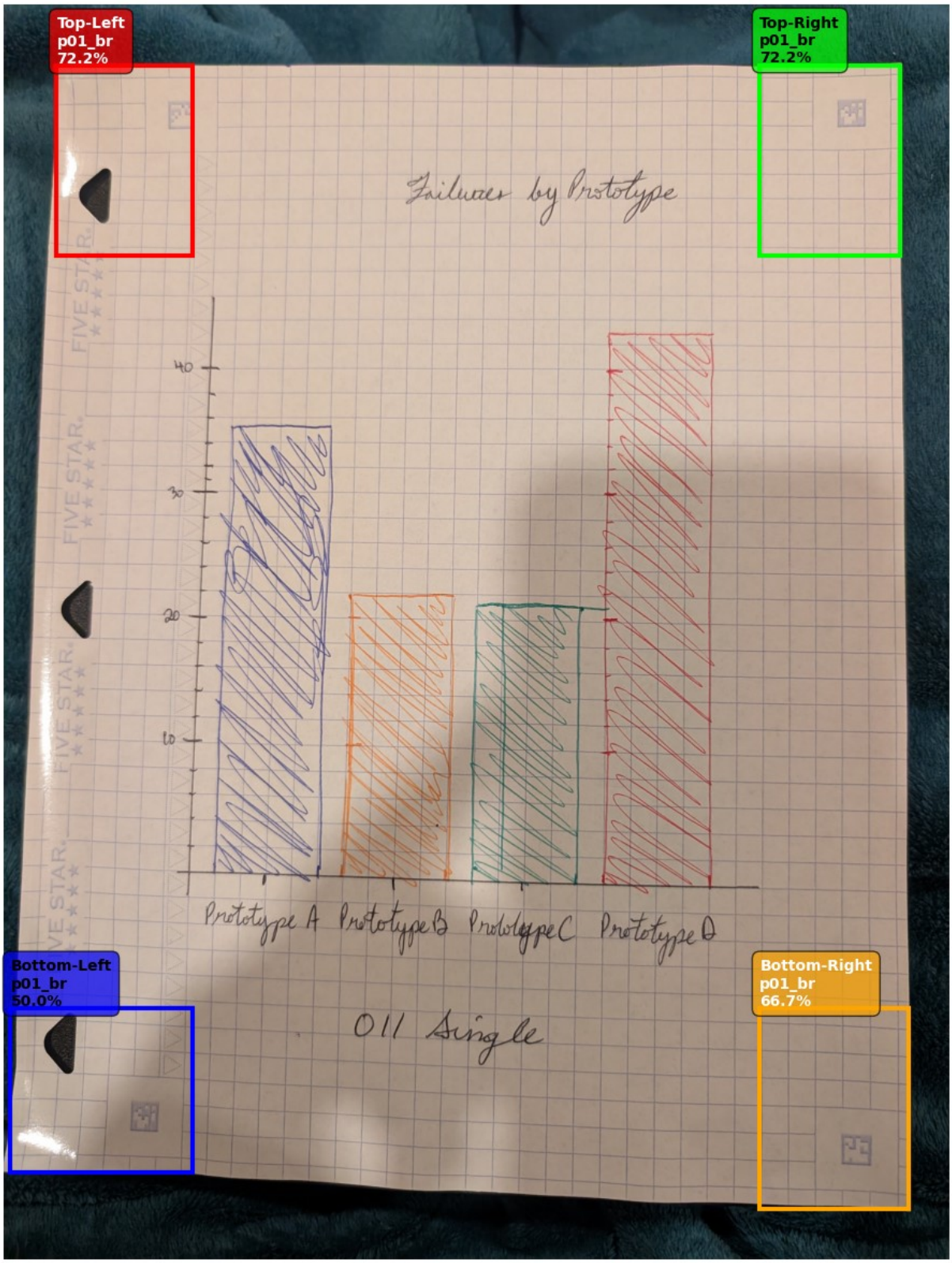
```python
def visualize_qc_squares(image, results, show_labels=True, figsize=(15, 12)):
    """
    Visualize detected QC squares on the original image.

    Parameters:
    -----------
    image : numpy.ndarray
        Original RGB image
    results : list of dicts
        List of detection results from detect_qc_square()
    show_labels : bool
        Whether to show labels with corner and pattern info
    figsize : tuple
        Figure size for matplotlib

    Returns:
    --------
    matplotlib.figure.Figure
        The figure object (can be saved or displayed)
    """
    # Create a copy of the image to draw on
    if len(image.shape) == 3:
        display_image = image.copy()
    else:
        display_image = cv2.cvtColor(image, cv2.COLOR_GRAY2RGB)

    # Define colors for each corner
    corner_colors = {
        'tl': (255, 0, 0),      # Red for top-left
        'tr': (0, 255, 0),      # Green for top-right
        'bl': (0, 0, 255),      # Blue for bottom-left
        'br': (255, 165, 0),    # Orange for bottom-right
    }

    corner_names = {
        'tl': 'Top-Left',
        'tr': 'Top-Right',
        'bl': 'Bottom-Left',
        'br': 'Bottom-Right'
    }

    # Create figure
    fig, ax = plt.subplots(1, 1, figsize=figsize)
    ax.imshow(display_image)
    ax.axis('off')

    # Draw bounding boxes for each detected QC square
    for result in results:
        if result is None:
            continue

        corner = result.get('corner', 'unknown')
        bbox = result.get('bbox', None)

        if bbox is None:
```

```
            continue

        x, y, w, h = bbox
        color = corner_colors.get(corner, (255, 255, 255))  # Default to white

        # Draw bounding box rectangle
        rect = patches.Rectangle(
            (x, y), w, h,
            linewidth=3,
            edgecolor=[c/255.0 for c in color],
            facecolor='none'
        )
        ax.add_patch(rect)

        # Add label if requested
        if show_labels:
            label_text = corner_names.get(corner, corner.upper())

            # Add pattern match info if available
            if result.get('pattern_match'):
                pattern_name = result['pattern_match']
                confidence = result.get('confidence', 0)
                label_text += f'\n{pattern_name}\n{confidence:.1%}'

            # Position label at top-left of bounding box
            # Adjust if too close to image edge
            label_x = x
            label_y = y - 10 if y > 30 else y + h + 10

            ax.text(
                label_x, label_y,
                label_text,
                fontsize=10,
                bbox=dict(
                    boxstyle='round,pad=0.5',
                    facecolor=[c/255.0 for c in color],
                    edgecolor='black',
                    alpha=0.7
                ),
                color='white' if corner in ['tl', 'br'] else 'black',
                weight='bold'
            )

    plt.tight_layout()
    return fig
```

In [12]:

```
# Load image and detect QC squares
image_path = 'hand_drawn_notes/bc_011_single-004.jpg'
image = cv2.imread(image_path)
rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Detect all QC squares
results = detect_qc_square(rgb_image, corner='auto')

# Visualize using matplotlib
fig = visualize_qc_squares(rgb_image, results, show_labels=True)
```

```
plt.show()

# Or save the figure
# fig.savefig('output_files/qc_detections.png', dpi=150, bbox_inches='tight')
```



Cropping

In [13]:

```python
def get_page_corners_from_qc(results, allow_diagonal_fallback=True):
    """
    Extract page corner coordinates from detected QC squares.

    Parameters:
    -----------
    results : list of dicts
        List of detection results from detect_qc_square()
        Should contain 4 results (one for each corner)
    allow_diagonal_fallback : bool
        When True (default) the function will synthesize the missing corners
        if only a TL/BR or TR/BL diagonal pair is detected.

    Returns:
    --------
    tuple
        (corner dictionary, fallback reason string or None). The dictionary has
        keys 'tl', 'tr', 'bl', 'br'. Returns (None, None) if the required
        corners cannot be computed.
    """
    corners = {}

    for result in results:
        if result is None:
            continue
        corner = result.get('corner')
        bbox = result.get('bbox')

        if corner and bbox:
            x, y, w, h = bbox

            # Use the outer corner of the bounding box (closest to image edge)
            # This represents the corner of the QC square envelope
            if corner == 'tl':
                corners['tl'] = (x, y)
            elif corner == 'tr':
                corners['tr'] = (x + w, y)
            elif corner == 'bl':
                corners['bl'] = (x, y + h)
            elif corner == 'br':
                corners['br'] = (x + w, y + h)

    required_corners = ['tl', 'tr', 'bl', 'br']
    if all(corner in corners for corner in required_corners):
        return corners, None

    def _synthesize_from_tl_br(tl, br):
        return {
            'tl': tl,
            'tr': (br[0], tl[1]),
            'br': br,
            'bl': (tl[0], br[1])
        }

    def _synthesize_from_tr_bl(tr, bl):
        return {
```

```
            'tl': (bl[0], tr[1]),
            'tr': tr,
            'br': (tr[0], bl[1]),
            'bl': bl
        }

    if allow_diagonal_fallback:
        if 'tl' in corners and 'br' in corners:
            synthesized = _synthesize_from_tl_br(corners['tl'], corners['br'])
            print("Partial corners detected: using TL/BR diagonal to estimate the m
            return synthesized, 'diagonal_tl_br'
        if 'tr' in corners and 'bl' in corners:
            synthesized = _synthesize_from_tr_bl(corners['tr'], corners['bl'])
            print("Partial corners detected: using TR/BL diagonal to estimate the m
            return synthesized, 'diagonal_tr_bl'

    missing = [c for c in required_corners if c not in corners]
    print(f"Warning: Missing corners: {missing}")
    return None, None


def crop_image_using_qc_corners(image, results, output_size=None, margin=0, allow_d
    """
    Crop and rectify the image using detected QC square corners.

    Parameters:
    -----------
    image : numpy.ndarray
        Original RGB image
    results : list of dicts
        List of detection results from detect_qc_square()
    output_size : tuple, optional
        Desired output size (width, height). If None, calculates from corners.
    margin : int or float
        Margin to add around the page (in pixels or as fraction of page size)
        Default: 0
    allow_diagonal_fallback : bool
        When True (default) synthesizes the missing corners when a TL/BR or
        TR/BL diagonal pair is available so the page can still be rectified.

    Returns:
    --------
    numpy.ndarray
        Cropped and rectified image
    dict
        Metadata including transformation matrix and corner coordinates
    """
    # Get corner coordinates
    corners, fallback_reason = get_page_corners_from_qc(results, allow_diagonal_fal
    if corners is None:
        raise ValueError("Could not extract all 4 corner points from QC squares")

    # Source points
    src_points = np.array([
        corners['tl'],  # Top-left
        corners['tr'],  # Top-right
```

```
        corners['br'],  # Bottom-right
        corners['bl']   # Bottom-left
    ], dtype=np.float32)

    # Calculate destination points
    if output_size is None:
        # Calculate output size based on the width and height of the page
        width_top = np.linalg.norm(src_points[1] - src_points[0])
        width_bottom = np.linalg.norm(src_points[2] - src_points[3])
        height_left = np.linalg.norm(src_points[3] - src_points[0])
        height_right = np.linalg.norm(src_points[2] - src_points[1])

        # Use average dimensions
        output_width = int(max(width_top, width_bottom))
        output_height = int(max(height_left, height_right))
    else:
        output_width, output_height = output_size

    # Apply margin
    if isinstance(margin, float):
        margin_x = int(output_width * margin)
        margin_y = int(output_height * margin)
    else:
        margin_x = margin_y = margin

    output_width += 2 * margin_x
    output_height += 2 * margin_y

    # Destination points
    dst_points = np.array([
        [margin_x, margin_y],                           # Top-left
        [output_width - margin_x, margin_y],            # Top-right
        [output_width - margin_x, output_height - margin_y],  # Bottom-right
        [margin_x, output_height - margin_y]            # Bottom-left
    ], dtype=np.float32)

    # Get perspective transformation matrix
    M = cv2.getPerspectiveTransform(src_points, dst_points)

    # Apply perspective transformation
    cropped = cv2.warpPerspective(
        image, M,
        (output_width, output_height),
        flags=cv2.INTER_LINEAR,
        borderMode=cv2.BORDER_CONSTANT,
        borderValue=(255, 255, 255)  # White background for areas outside page
    )

    metadata = {
        'transformation_matrix': M,
        'source_corners': corners,
        'output_size': (output_width, output_height),
        'margin': (margin_x, margin_y),
        'corner_source': fallback_reason or 'detected'
    }
```

```
    return cropped, metadata

# Alternative: Use center of QC squares instead of corners
def get_page_corners_from_qc_centers(results):
    """
    Extract page corners using the center of each QC square.
    This is useful if you want the page boundary to be at the center of the QC squa
    """
    corners = {}

    for result in results:
        if result is None:
            continue
        corner = result.get('corner')
        bbox = result.get('bbox')

        if corner and bbox:
            x, y, w, h = bbox
            # Use center of bounding box
            center_x = x + w / 2
            center_y = y + h / 2
            corners[corner] = (center_x, center_y)

    required_corners = ['tl', 'tr', 'bl', 'br']
    if all(corner in corners for corner in required_corners):
        return corners
    else:
        missing = [c for c in required_corners if c not in corners]
        print(f"Warning: Missing corners: {missing}")
        return None

# Visualization function to show the crop region
def visualize_crop_region(image, results, show_corners=True):
    """
    Visualize the crop region defined by QC squares.
    """
    corners, _ = get_page_corners_from_qc(results)
    if corners is None:
        print("Cannot visualize: missing corners")
        return None

    # Create a copy to draw on
    display_image = image.copy()

    # Draw lines connecting corners
    corner_order = ['tl', 'tr', 'br', 'bl', 'tl']  # Close the polygon
    points = [corners[corner] for corner in corner_order]
    points = np.array(points, dtype=np.int32)

    # Draw polygon outline
    cv2.polylines(display_image, [points], isClosed=True,
                  color=(0, 255, 0), thickness=3)

    # Draw corner points
    if show_corners:
        for corner_name, (x, y) in corners.items():
```

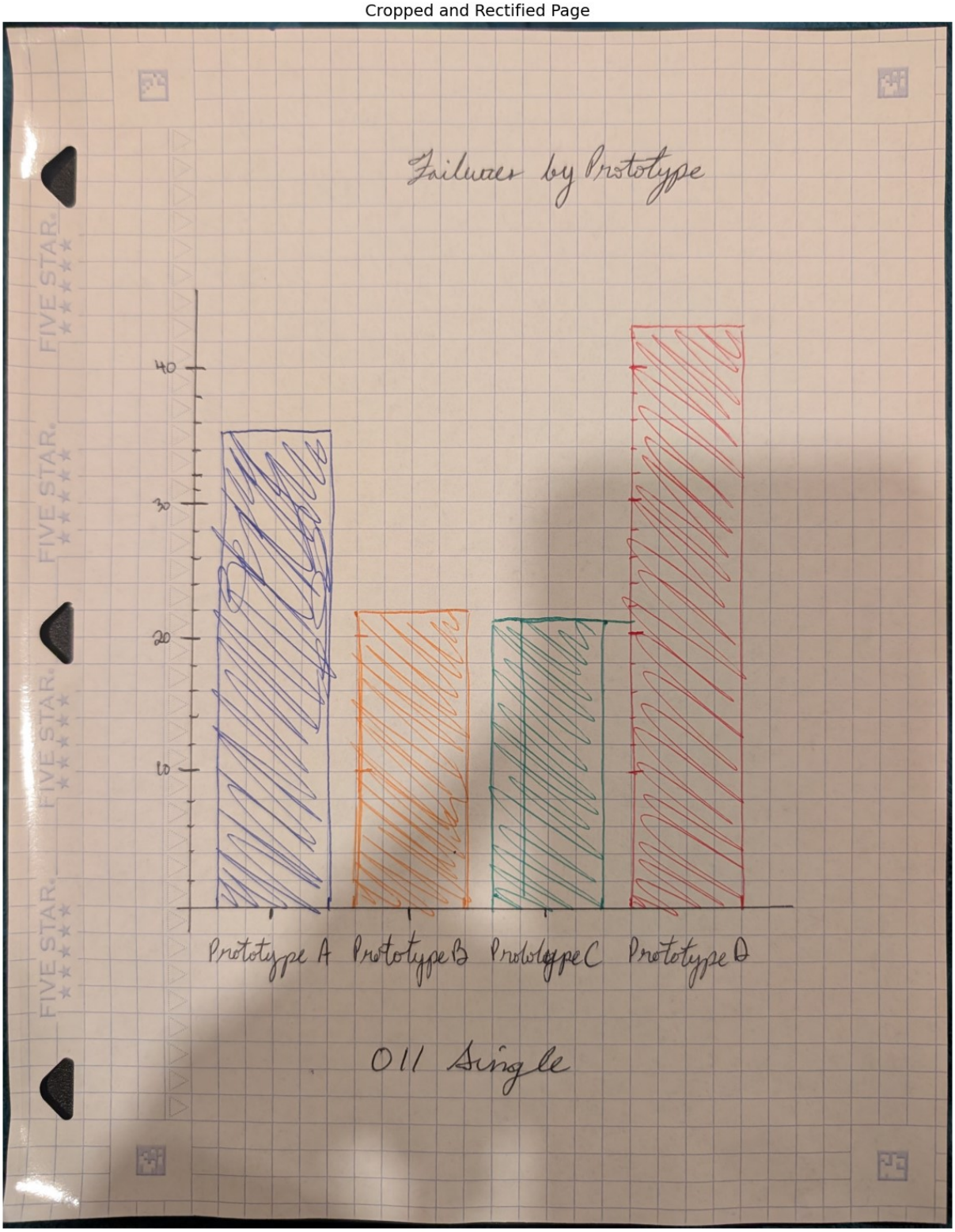
```
            cv2.circle(display_image, (int(x), int(y)), 10, (255, 0, 0), -1)
            cv2.putText(display_image, corner_name.upper(),
                        (int(x) + 15, int(y)),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255, 0, 0), 2)

    return display_image
```

Test

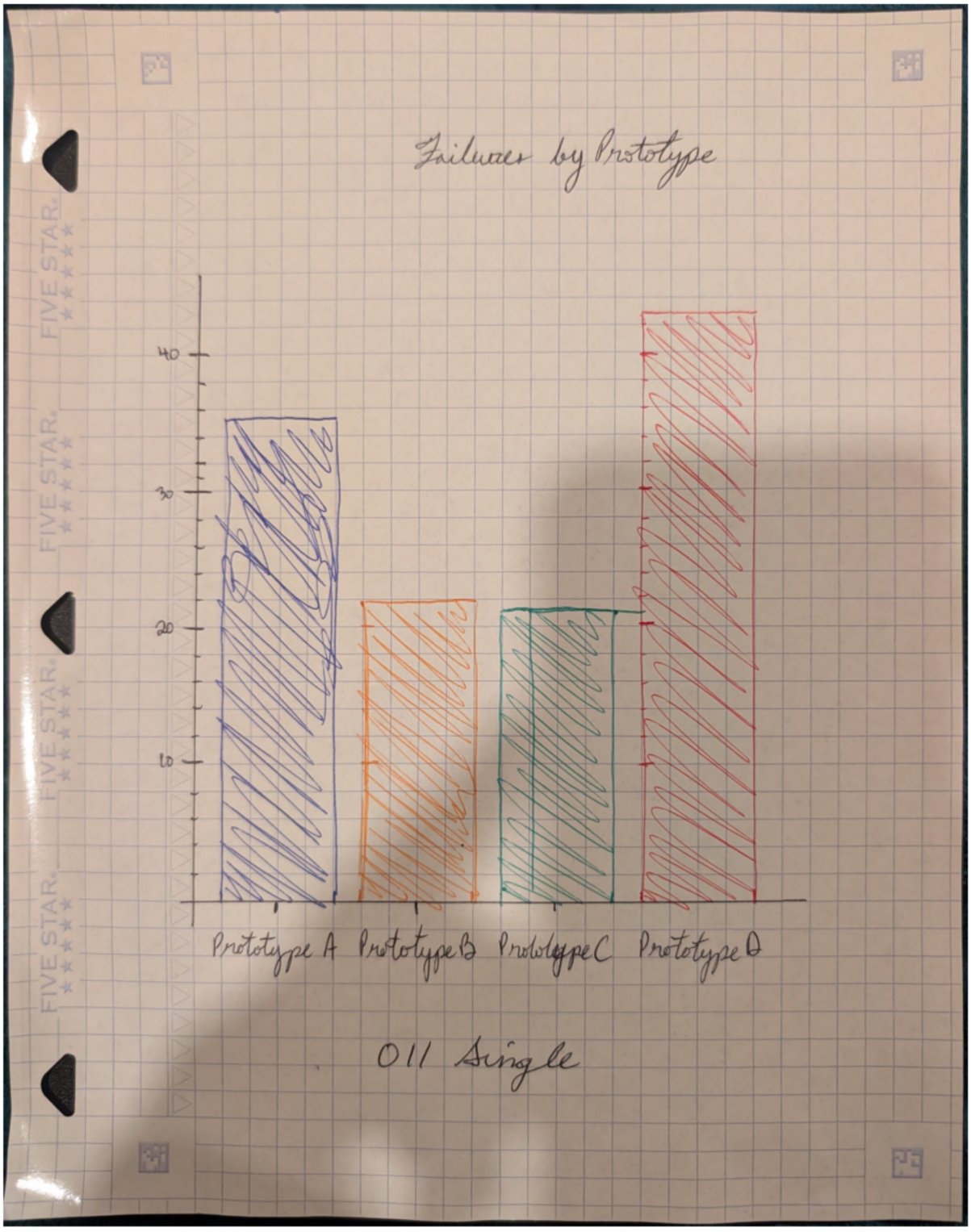
In [14]:
```
def visualize_cropped_result(cropped_image, title="Cropped and Rectified Page", fig
    """
    Visualize the cropped/rectified image result.

    Parameters:
    -----------
    cropped_image : numpy.ndarray
        The cropped image result
    title : str
        Title for the plot
    figsize : tuple
        Figure size for matplotlib
    """
    fig, ax = plt.subplots(1, 1, figsize=figsize)
    ax.imshow(cropped_image)
    ax.set_title(title, fontsize=14)
    ax.axis('off')
    plt.tight_layout()
    return fig
```

In [15]:
```
# Load and detect
image_path = 'hand_drawn_notes/bc_011_single-004.jpg'
image = cv2.imread(image_path)
rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
results = detect_qc_square(rgb_image, corner='auto')

# Crop the image
cropped_image, metadata = crop_image_using_qc_corners(
    rgb_image, results, margin=20
)

# Show the cropped result
fig = visualize_cropped_result(cropped_image)
plt.figure(figsize=(12, 15))
plt.imshow(cropped_image)
plt.title('Cropped Result')
plt.axis('off')
plt.show()

fig.savefig('output_files/bc_011_single_cropped.png', dpi=150, bbox_inches='tight')
```

Cropped and Rectified Page



Cropped Result



Rotated Photo

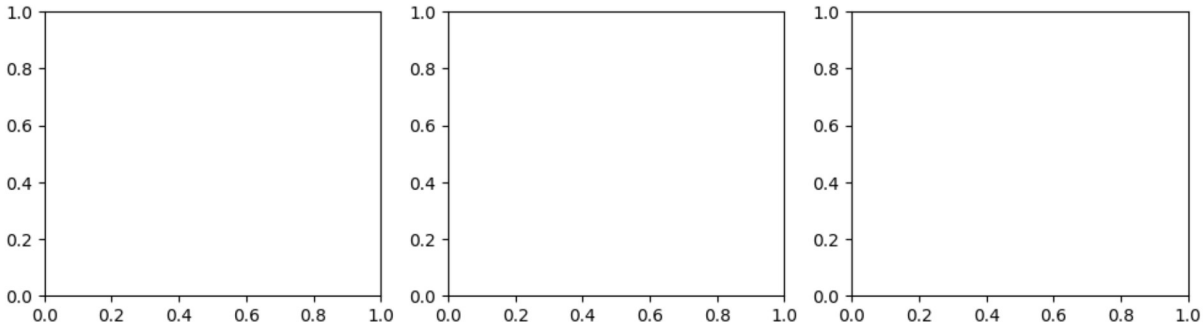
In [16]:
```
NotImplementedError
```

Out[16]:
```
NotImplementedError
```

Gallery of Processed Images

In [21]:
```
# Gallery of Processed Images

import matplotlib.pyplot as plt
import matplotlib.patches as patches
# from normalize_image import detect_qc_square  # adjust import path
import cv2
import math

image_paths = [

...

]  # Selection
ncols = 3
nrows = math.ceil(len(image_paths) / ncols)
fig, axes = plt.subplots(nrows, ncols, figsize=(ncols * 4, nrows * 3))

for ax, path in zip(axes.flat, image_paths):
  image = cv2.cvtColor(cv2.imread(path), cv2.COLOR_BGR2RGB)
  results = detect_qc_square(image, corner='auto')
  ax.imshow(image)
  ax.axis('off')
  for result in results or []:
      x, y, w, h = result['bbox']
      color = {'tl':'red','tr':'green','bl':'blue','br':'orange'}.get(result['corne
      rect = patches.Rectangle((x, y), w, h, edgecolor=color, facecolor='none', lw=
      ax.add_patch(rect)
      ax.text(x, y - 5, f"{result['corner']} {result.get('confidence',0):.0%}", col
              bbox=dict(facecolor=color, alpha=0.6, pad=1), fontsize=8)

for ax in axes.flat[len(image_paths):]:
  ax.remove()
plt.tight_layout()
plt.savefig("output_files/qc_corner_gallery.png", dpi=150)
```

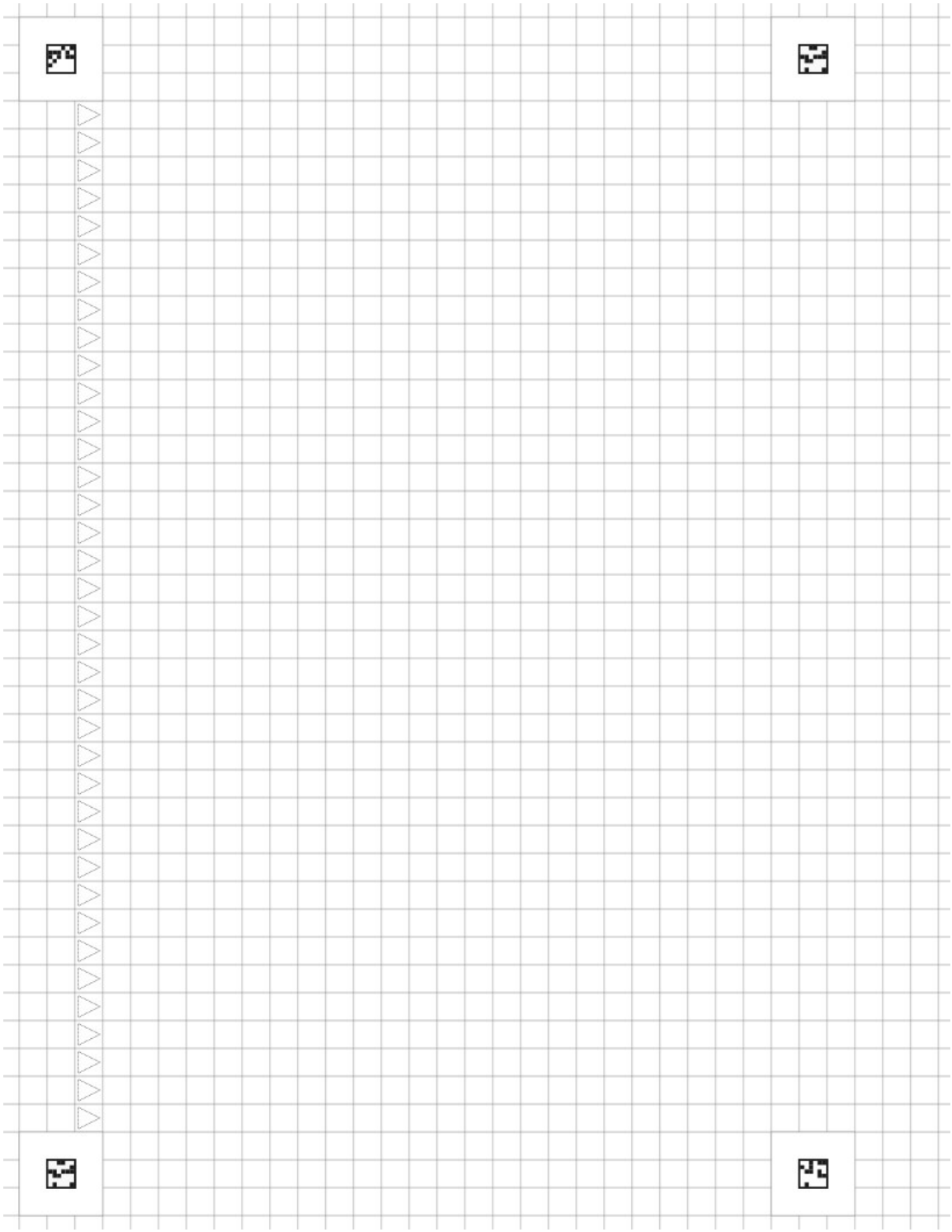
```
---------------------------------------------------------------------------
error                                     Traceback (most recent call last)
Cell In[21], line 19
     16 fig, axes = plt.subplots(nrows, ncols, figsize=(ncols * 4, nrows * 3))
     18 for ax, path in zip(axes.flat, image_paths):
---> 19   image = cv2.cvtColor(cv2.imread(path), cv2.COLOR_BGR2RGB)
     20   results = detect_qc_square(image, corner='auto')
     21   ax.imshow(image)

error: OpenCV(4.12.0) :-1: error: (-5:Bad argument) in function 'imread'
> Overload resolution failed:
>  - Expected 'filename' to be a str or path-like object
>  - Expected 'filename' to be a str or path-like object
>  - Expected 'filename' to be a str or path-like object
```



In [ ]:

A3 Page Types and Features

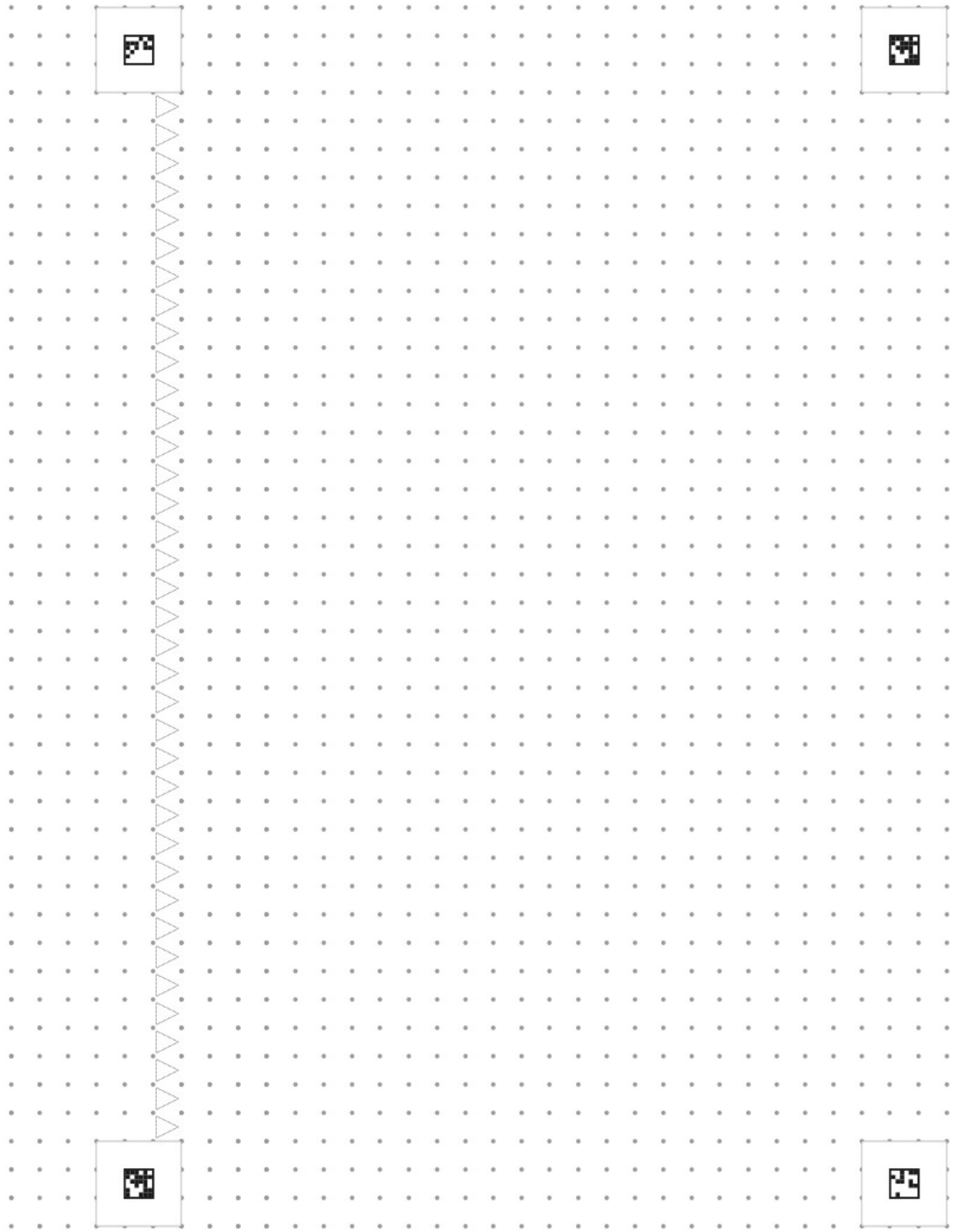
I am working on different kinds of pages for my personal applications. I am also working on different features to help with administrative work.



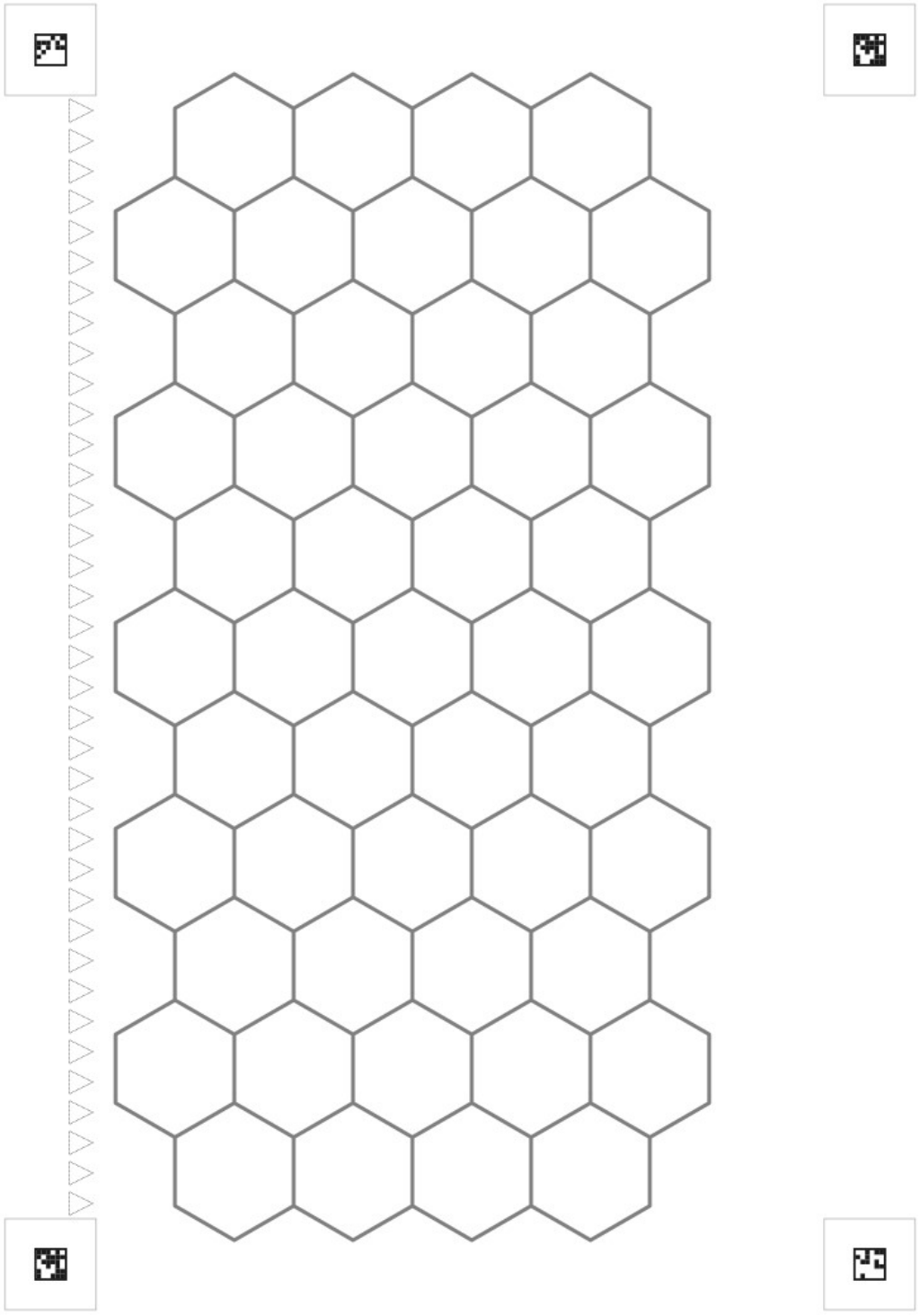
Regular Page

Lined Page

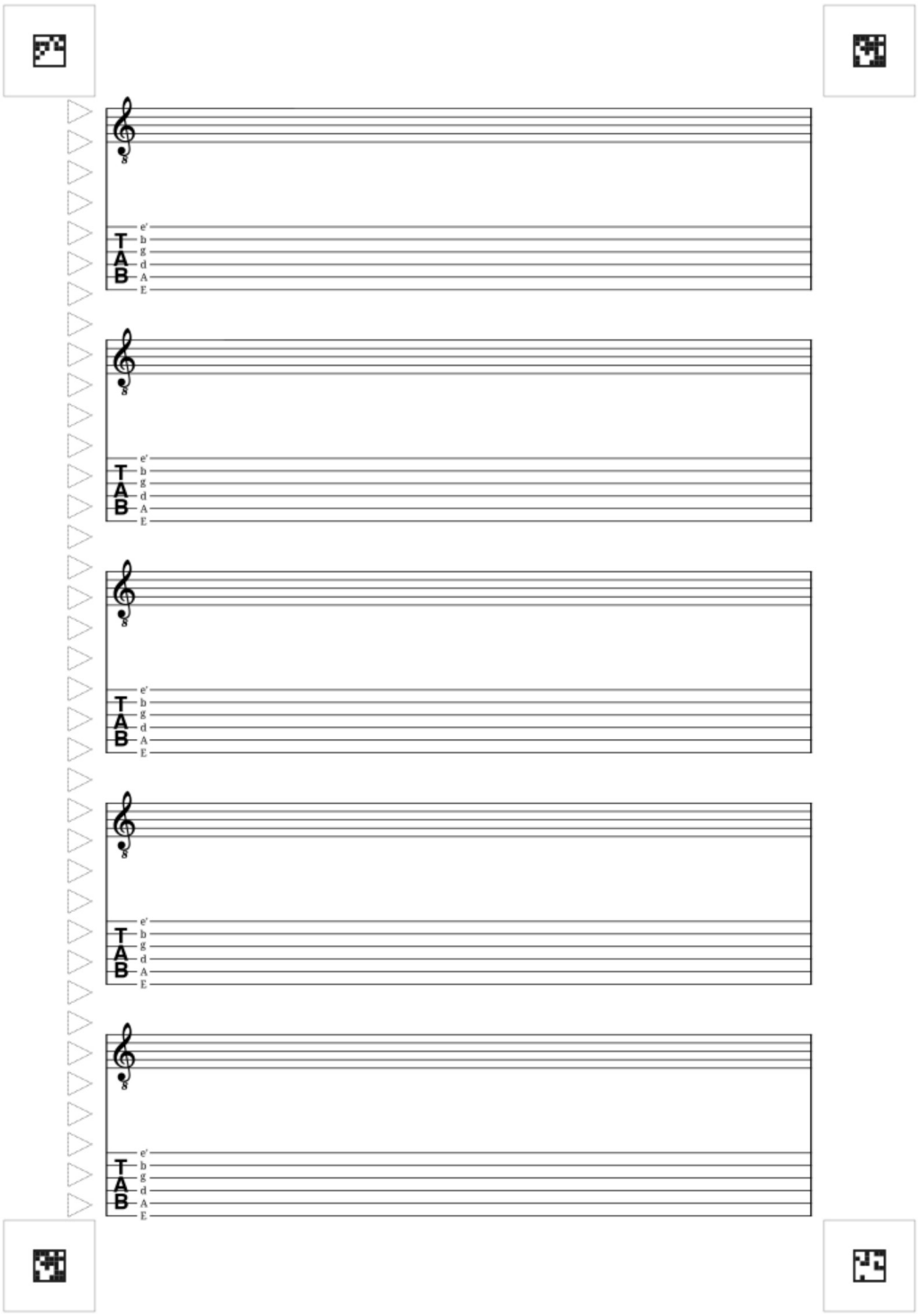
Dot Matrix



Hex



Musical Staff and Guitar Tab



Anti-Cheat Page Signer Code

Here is my `homework_signer.py` code that is meant to sign a `.pdf` to look like the example in **A4**

In [ ]:
```
try:
    import svgwrite
    HAS_SVGWRITE = True
except ImportError:
```

```
    HAS_SVGWRITE = False

import xml.etree.ElementTree as ET
from xml.dom import minidom
import csv
import os
import argparse
import sys
import base64

# PDF processing imports
try:
    import fitz  # PyMuPDF
    HAS_PYMUPDF = True
except ImportError:
    HAS_PYMUPDF = False

# 4-state barcode generator for Royal Mail (RM4SCC-like)
# Following basic RM4SCC (often used for postal barcodes): F, A, D, T (tracker, asc

# Map each character to barcode bars (12 characters max, can be extended)
CHAR_MAP = {
    'A': 'ATDA', 'B': 'ADTA', 'C': 'AATD', 'D': 'ADAT', 'E': 'TADA', 'F': 'TDAA',
    'G': 'TDDA', 'H': 'TADA', 'I': 'DTAA', 'J': 'DATD', 'K': 'DAAT', 'L': 'DAAD',
    'M': 'ATAD', 'N': 'TDAA', 'O': 'TDAD', 'P': 'TDDA', 'Q': 'ATAA', 'R': 'AADT',
    'S': 'AADT', 'T': 'AATA', 'U': 'ATDD', 'V': 'TADD', 'W': 'TDDA', 'X': 'TADA',
    'Y': 'DTAA', 'Z': 'DATD',
    '0': 'ADDA', '1': 'DADA', '2': 'DAAD', '3': 'ADAD', '4': 'DADA', '5': 'DAAD',
    '6': 'DDAA', '7': 'DADA', '8': 'DAAD', '9': 'DDAA',
    # Add mappings as needed
}

def char_to_bars(c):
    c = c.upper()
    return CHAR_MAP.get(c, 'ATDA')  # default fallback

def encode_4state_barcode(data):
    bars = []
    for c in data:
        bars.extend(char_to_bars(c))
    return bars

def generate_4state_barcode_svg(data, filename='barcode.svg', bar_width=4, space=2,
    if not HAS_SVGWRITE:
        raise ImportError("svgwrite is required for generate_4state_barcode_svg. In
    bars = encode_4state_barcode(data)
    dwg = svgwrite.Drawing(filename, size=(len(bars) * (bar_width + space), height_
    x = 0
    y_full = 0
    y_tracker = (height_full - tracker_height) // 2
    for bar in bars:
        if bar == 'F':  # Full
            dwg.add(dwg.rect((x, y_full), (bar_width, height_full), fill='black'))
        elif bar == 'A':  # Ascender
            dwg.add(dwg.rect((x, y_full), (bar_width, height_asc), fill='black'))
        elif bar == 'D':  # Descender
```

```
            dwg.add(dwg.rect((x, height_full - height_desc), (bar_width, height_des
        elif bar == 'T':  # Tracker
            dwg.add(dwg.rect((x, y_tracker), (bar_width, tracker_height), fill='bla
        x += bar_width + space
    dwg.save()

def insert_barcode_into_svg(input_svg_path, barcode_data, output_svg_path=None, bar
    """
    Insert a barcode into an existing SVG document at the rect element with ID 'aus
    If no auspost rect is found, adds barcode at specified position or default bott

    Args:
        input_svg_path: Path to the input SVG file
        barcode_data: String data to encode as barcode
        output_svg_path: Path to save the output SVG (defaults to input_svg_path)
        bar_width: Width of each bar (auto-calculated if None)
        space: Space between bars (auto-calculated if None)
        label_text: Text to display in the label (defaults to barcode_data if None)
        position: Tuple (x, y) for barcode position if no auspost rect found (defau
    """
    if output_svg_path is None:
        output_svg_path = input_svg_path

    # Parse the existing SVG
    tree = ET.parse(input_svg_path)
    root = tree.getroot()

    # Register namespaces to handle SVG properly
    namespaces = {'svg': 'http://www.w3.org/2000/svg'}
    svg_namespace = None
    if root.tag.startswith('{'):
        # Extract namespace from root tag
        ns = root.tag.split('}')[0].strip('{')
        namespaces[''] = ns
        svg_namespace = ns
    else:
        # Check for default namespace in xmlns attribute
        if 'xmlns' in root.attrib:
            svg_namespace = root.attrib['xmlns']
        else:
            svg_namespace = 'http://www.w3.org/2000/svg'

    # Find ALL rects with ID "auspost" or inkscape:label="auspost" (one per page)
    auspost_rects = []
    auspost_label_texts = []

    def find_auspost_recursive(elem, parent=None):
        nonlocal auspost_rects, auspost_label_texts
        # Handle both namespaced and non-namespaced elements
        tag = elem.tag.split('}')[-1] if '}' in elem.tag else elem.tag
        if tag == 'rect':
            # Check for id="auspost" or inkscape:label="auspost"
            elem_id = elem.get('id', '')
            # Check for inkscape:label attribute (may be namespaced)
            inkscape_label = None
            for attr_name, attr_value in elem.attrib.items():
```

```
                if attr_name.endswith('label') and attr_value == 'auspost':
                    inkscape_label = attr_value
                    break
            if elem_id == 'auspost' or inkscape_label == 'auspost':
                auspost_rects.append((elem, parent))
        elif tag == 'text':
            # Check for inkscape:label="auspost_label"
            inkscape_label = None
            for attr_name, attr_value in elem.attrib.items():
                if attr_name.endswith('label') and attr_value == 'auspost_label':
                    inkscape_label = attr_value
                    break
            if inkscape_label == 'auspost_label':
                auspost_label_texts.append(elem)
        # Recursively search children
        for child in elem:
            find_auspost_recursive(child, elem)

    find_auspost_recursive(root)

    # Get SVG dimensions for fallback positioning
    # Try viewBox first (most reliable)
    if 'viewBox' in root.attrib:
        viewbox_parts = root.attrib['viewBox'].split()
        if len(viewbox_parts) >= 4:
            svg_width = float(viewbox_parts[2])
            svg_height = float(viewbox_parts[3])
        else:
            # Fallback to width/height attributes
            width_attr = root.get('width', '612')
            height_attr = root.get('height', '792')
            # Handle units like '8.5in'
            if 'in' in str(width_attr):
                svg_width = float(str(width_attr).replace('in', '').strip()) * 96
            else:
                svg_width = float(width_attr)
            if 'in' in str(height_attr):
                svg_height = float(str(height_attr).replace('in', '').strip()) * 96
            else:
                svg_height = float(height_attr)
    else:
        # Parse width/height attributes with unit handling
        width_attr = root.get('width', '612')
        height_attr = root.get('height', '792')
        # Handle units like '8.5in'
        if 'in' in str(width_attr):
            svg_width = float(str(width_attr).replace('in', '').strip()) * 96
        else:
            svg_width = float(width_attr)
        if 'in' in str(height_attr):
            svg_height = float(str(height_attr).replace('in', '').strip()) * 96
        else:
            svg_height = float(height_attr)

    # Encode the barcode
    bars = encode_4state_barcode(barcode_data)
```

```
    num_bars = len(bars)

    # If no auspost rects found, use default position
    if not auspost_rects:
        # Default to bottom-right corner with padding
        if position is None:
            # Calculate default size for barcode area
            default_width = min(200, svg_width * 0.3)
            default_height = 50
            default_x = svg_width - default_width - 20  # 20px padding from right
            default_y = svg_height - default_height - 20  # 20px padding from botto
            position = (default_x, default_y)
            rect_width = default_width
            rect_height = default_height
        else:
            default_width = min(200, svg_width * 0.3)
            default_height = 50
            rect_width = default_width
            rect_height = default_height
    else:
        # Use the first rect for dimensions (they should all be the same size)
        first_rect, _ = auspost_rects[0]
        rect_width = float(first_rect.get('width', 100))
        rect_height = float(first_rect.get('height', 50))

    # Calculate bar dimensions to fit within the rect
    if bar_width is None or space is None:
        # Auto-calculate to fit the width
        # Total width needed: num_bars * bar_width + (num_bars - 1) * space
        # We want this to fit in rect_width with some padding
        available_width = rect_width * 0.95  # 95% of width for padding
        if num_bars > 1:
            # bar_width = available_width / (num_bars + (num_bars - 1) * space_rati
            # Assuming space = bar_width / 2 for good appearance
            space_ratio = 0.5
            bar_width = available_width / (num_bars * (1 + space_ratio) - space_rat
            space = bar_width * space_ratio
        else:
            bar_width = available_width
            space = 0

    # Calculate heights to fit within rect_height
    height_full = rect_height * 0.9  # 90% of height
    height_asc = height_full * 0.8
    height_desc = height_full * 0.8
    tracker_height = height_full * 0.2

    # Calculate total barcode width for centering
    total_barcode_width = num_bars * bar_width + (num_bars - 1) * space

    # Remove the original rects or make them invisible (for all pages)
    if auspost_rects:
        for auspost_rect, _ in auspost_rects:
            auspost_rect.set('fill', 'none')
            auspost_rect.set('stroke', 'none')
```

```
    # Find all page layers (page 0 and page 1)
    page_layers = []
    for elem in root.iter():
        tag = elem.tag.split('}')[-1] if '}' in elem.tag else elem.tag
        if tag == 'g':
            # Check for inkscape:label starting with "page" or id="layer2"/"layer3"
            elem_id = elem.get('id', '')
            inkscape_label = None
            for attr_name, attr_value in elem.attrib.items():
                if attr_name.endswith('label') and attr_value and attr_value.starts
                    inkscape_label = attr_value
                    break
            if elem_id in ['layer2', 'layer3'] or (inkscape_label and inkscape_labe
                page_layers.append(elem)

    # Determine where to insert barcode
    if auspost_rects:
        # Match auspost rects with page layers
        # Insert barcode rects into each page using the corresponding auspost rect
        for i, (auspost_rect, auspost_parent) in enumerate(auspost_rects):
            # Get the rect position for this specific page
            rect_x = float(auspost_rect.get('x', 0))
            rect_y = float(auspost_rect.get('y', 0))

            # Calculate starting position (centered horizontally) for this page
            start_x = rect_x + (rect_width - total_barcode_width) / 2
            start_y = rect_y + (rect_height - height_full) / 2
            y_tracker = start_y + (height_full - tracker_height) / 2

            # Find the corresponding page layer parent
            if page_layers and i < len(page_layers):
                parent = page_layers[i]
            else:
                parent = auspost_parent if auspost_parent is not None else root

            # Insert barcode rects for this page
            x_pos = start_x
            for bar in bars:
                # Create rect element with proper namespace
                if svg_namespace:
                    rect_elem = ET.Element('{' + svg_namespace + '}rect')
                else:
                    rect_elem = ET.Element('rect')
                # Set fill color as requested - using style attribute for better co
                rect_elem.set('style', 'fill:#CCCCCCFF;fill-opacity:1')
                # Also set as attribute for maximum compatibility
                rect_elem.set('fill', '#CCCCCCFF')

                if bar == 'F':  # Full
                    rect_elem.set('x', str(x_pos))
                    rect_elem.set('y', str(start_y))
                    rect_elem.set('width', str(bar_width))
                    rect_elem.set('height', str(height_full))
                elif bar == 'A':  # Ascender
                    rect_elem.set('x', str(x_pos))
                    rect_elem.set('y', str(start_y))
```

```
                    rect_elem.set('width', str(bar_width))
                    rect_elem.set('height', str(height_asc))
                elif bar == 'D':  # Descender
                    rect_elem.set('x', str(x_pos))
                    rect_elem.set('y', str(start_y + height_full - height_desc))
                    rect_elem.set('width', str(bar_width))
                    rect_elem.set('height', str(height_desc))
                elif bar == 'T':  # Tracker
                    rect_elem.set('x', str(x_pos))
                    rect_elem.set('y', str(y_tracker))
                    rect_elem.set('width', str(bar_width))
                    rect_elem.set('height', str(tracker_height))

                parent.append(rect_elem)
                x_pos += bar_width + space
    else:
        # No auspost rects found - add barcode at specified or default position
        if position is None:
            position = (svg_width - rect_width - 20, svg_height - rect_height - 20)

        start_x = position[0] + (rect_width - total_barcode_width) / 2
        start_y = position[1] + (rect_height - height_full) / 2
        y_tracker = start_y + (height_full - tracker_height) / 2

        # Insert barcode rects directly into root
        x_pos = start_x
        for bar in bars:
            # Create rect element with proper namespace
            if svg_namespace:
                rect_elem = ET.Element('{' + svg_namespace + '}rect')
            else:
                rect_elem = ET.Element('rect')
            # Set fill color as requested - using style attribute for better compat
            rect_elem.set('style', 'fill:#CCCCCCFF;fill-opacity:1')
            # Also set as attribute for maximum compatibility
            rect_elem.set('fill', '#CCCCCCFF')

            if bar == 'F':  # Full
                rect_elem.set('x', str(x_pos))
                rect_elem.set('y', str(start_y))
                rect_elem.set('width', str(bar_width))
                rect_elem.set('height', str(height_full))
            elif bar == 'A':  # Ascender
                rect_elem.set('x', str(x_pos))
                rect_elem.set('y', str(start_y))
                rect_elem.set('width', str(bar_width))
                rect_elem.set('height', str(height_asc))
            elif bar == 'D':  # Descender
                rect_elem.set('x', str(x_pos))
                rect_elem.set('y', str(start_y + height_full - height_desc))
                rect_elem.set('width', str(bar_width))
                rect_elem.set('height', str(height_desc))
            elif bar == 'T':  # Tracker
                rect_elem.set('x', str(x_pos))
                rect_elem.set('y', str(y_tracker))
                rect_elem.set('width', str(bar_width))
```

```
                rect_elem.set('height', str(tracker_height))

            root.append(rect_elem)
            x_pos += bar_width + space

        # Add label text below barcode if label_text is provided
        if label_text:
            if svg_namespace:
                text_elem = ET.Element('{' + svg_namespace + '}text')
            else:
                text_elem = ET.Element('text')
            text_elem.set('x', str(start_x))
            text_elem.set('y', str(start_y + height_full + 15))
            text_elem.set('font-family', 'Arial, sans-serif')
            text_elem.set('font-size', '12')
            text_elem.set('fill', '#000000')
            text_elem.text = label_text
            root.append(text_elem)

    # Update ALL existing text elements with label "auspost_label" (one per page)
    for auspost_label_text in auspost_label_texts:
        # Use label_text if provided, otherwise use barcode_data
        display_text = label_text if label_text is not None else barcode_data

        # Clear any existing text content from the text element itself
        auspost_label_text.text = None
        # Find the tspan element inside the text element and update it
        tspan_found = False
        for tspan in auspost_label_text.iter():
            tspan_tag = tspan.tag.split('}')[-1] if '}' in tspan.tag else tspan.tag
            if tspan_tag == 'tspan':
                # Update the text content in the tspan
                tspan.text = display_text
                tspan_found = True
                break
        # If no tspan found, create one or set text directly
        if not tspan_found:
            # Create a tspan element if it doesn't exist
            if svg_namespace:
                tspan_elem = ET.Element('{' + svg_namespace + '}tspan')
            else:
                tspan_elem = ET.Element('tspan')
            tspan_elem.text = display_text
            auspost_label_text.append(tspan_elem)

    # Save the modified SVG
    # Pretty print the XML
    xml_str = ET.tostring(root, encoding='unicode')
    dom = minidom.parseString(xml_str)
    pretty_xml = dom.toprettyxml(indent="  ")

    # Remove the XML declaration line added by minidom if the original didn't have
    with open(input_svg_path, 'r', encoding='utf-8') as f:
        original_content = f.read()
        has_xml_declaration = original_content.strip().startswith('<?xml')
```

```
    if not has_xml_declaration:
        # Remove the XML declaration
        lines = pretty_xml.split('\n')
        if lines[0].startswith('<?xml'):
            pretty_xml = '\n'.join(lines[1:])

    with open(output_svg_path, 'w', encoding='utf-8') as f:
        f.write(pretty_xml)

# Example usage:
# generate_4state_barcode_svg("HELLO123", "hello_barcode.svg")
# insert_barcode_into_svg("template.svg", "HELLO123", "output.svg")


def sign_pdf_with_barcodes(pdf_path, csv_path='v4_uuids.csv', output_path=None, out
    """
    Process a PDF document: overlay each page onto template SVG, add unique barcode

    Args:
        pdf_path: Path to the input PDF file
        csv_path: Path to the v4_uuids.csv file
        output_path: Path for the final merged PDF (defaults to input name with '_s
        output_dir: Directory for temporary files (defaults to same as PDF)
        template_svg_path: Path to the calibration template SVG (default: calibrati

    Returns:
        tuple: (output_path, list of (page_num, uuid, temp_pdf_path) tuples)
    """
    if not HAS_PYMUPDF:
        raise ImportError("PyMuPDF (fitz) is required. Install with: pip install Py

    # Check that at least one of the page-specific templates exists
    if not os.path.exists('calibration_page-coloured-0.svg') and not os.path.exists
        # Fallback: check if default template exists
        if not os.path.exists(template_svg_path):
            raise FileNotFoundError(f"Template SVGs not found: calibration_page-col

    if output_dir is None:
        output_dir = os.path.dirname(pdf_path) or '.'

    if output_path is None:
        base_name = os.path.splitext(os.path.basename(pdf_path))[0]
        output_path = os.path.join(output_dir, f"{base_name}_signed.pdf")

    # Read the CSV file
    rows = []
    with open(csv_path, 'r', encoding='utf-8') as f:
        reader = csv.DictReader(f)
        rows = list(reader)

    # Open the PDF
    doc = fitz.open(pdf_path)
    num_pages = len(doc)

    signed_pages = []
    temp_files = []
```

```
try:
    # Process each page by overlaying onto template
    for page_num in range(num_pages):
        # Find next available UUID
        selected_uuid = None
        selected_index = None
        for i, row in enumerate(rows):
            if not row.get('entity', '').strip() and not row.get('state', '').s
                selected_uuid = row['uuid']
                selected_index = i
                break

        if selected_uuid is None:
            print(f"Warning: No more UUIDs available for page {page_num + 1}")
            break

        # Get the PDF page (preserve vector content)
        page = doc[page_num]
        page_rect = page.rect

        # Select the correct template file based on even/odd page numbers
        # Odd pages (1, 3, 5...) → calibration_page-coloured-1.svg (which has "
        # Even pages (0, 2, 4...) → calibration_page-coloured-0.svg (which has
        if page_num % 2 == 1:  # Odd page (1-indexed: 1, 3, 5...)
            page_template_path = 'calibration_page-coloured-1.svg'
            template_page_label = "page 1"
        else:  # Even page (0-indexed: 0, 2, 4...)
            page_template_path = 'calibration_page-coloured-0.svg'
            template_page_label = "page 0"

        # Check if the page-specific template exists, fallback to default templ
        if not os.path.exists(page_template_path):
            if os.path.exists(template_svg_path):
                page_template_path = template_svg_path
            else:
                raise FileNotFoundError(f"Template SVG not found: {page_templat

        # Read template SVG to get auspost rect position and render as backgrou
        tree = ET.parse(page_template_path)
        root = tree.getroot()

        # Get SVG dimensions - try viewBox first (most reliable)
        if 'viewBox' in root.attrib:
            viewbox_parts = root.attrib['viewBox'].split()
            if len(viewbox_parts) >= 4:
                svg_width = float(viewbox_parts[2])
                svg_height = float(viewbox_parts[3])
            else:
                # Fallback to width/height attributes
                width_attr = root.get('width', '816')
                height_attr = root.get('height', '1056')
                # Handle units like '8.5in'
                if 'in' in str(width_attr):
                    svg_width = float(str(width_attr).replace('in', '').strip()
                else:
```

```
                    svg_width = float(width_attr)
                if 'in' in str(height_attr):
                    svg_height = float(str(height_attr).replace('in', '').strip
                else:
                    svg_height = float(height_attr)
            else:
                # Parse width/height attributes with unit handling
                width_attr = root.get('width', '816')
                height_attr = root.get('height', '1056')
                # Handle units like '8.5in'
                if 'in' in str(width_attr):
                    svg_width = float(str(width_attr).replace('in', '').strip()) *
                else:
                    svg_width = float(width_attr)
                if 'in' in str(height_attr):
                    svg_height = float(str(height_attr).replace('in', '').strip())
                else:
                    svg_height = float(height_attr)

            # Find auspost rect and auspost_label text in the appropriate template
            auspost_rect = None
            auspost_label_text = None
            target_page_layer = None

            # First, find the target page layer
            for elem in root.iter():
                tag = elem.tag.split('}')[-1] if '}' in elem.tag else elem.tag
                if tag == 'g':
                    elem_id = elem.get('id', '')
                    inkscape_label = None
                    for attr_name, attr_value in elem.attrib.items():
                        if attr_name.endswith('label') and attr_value == template_p
                            inkscape_label = attr_value
                            break
                    if elem_id in ['layer2', 'layer3'] or inkscape_label == templat
                        target_page_layer = elem
                        break

            # Search for auspost elements - check both the page layer and meta laye
            def find_auspost_elements(elem):
                nonlocal auspost_rect, auspost_label_text
                tag = elem.tag.split('}')[-1] if '}' in elem.tag else elem.tag
                if tag == 'rect':
                    elem_id = elem.get('id', '')
                    inkscape_label = None
                    for attr_name, attr_value in elem.attrib.items():
                        if attr_name.endswith('label') and attr_value == 'auspost':
                            inkscape_label = attr_value
                            break
                    if elem_id == 'auspost' or inkscape_label == 'auspost':
                        auspost_rect = elem
                elif tag == 'text':
                    inkscape_label = None
                    for attr_name, attr_value in elem.attrib.items():
                        if attr_name.endswith('label') and attr_value == 'auspost_l
                            inkscape_label = attr_value
```

```
                    break
            if inkscape_label == 'auspost_label':
                auspost_label_text = elem
        for child in elem:
            find_auspost_elements(child)

    # Search in the target page layer if found
    if target_page_layer is not None:
        find_auspost_elements(target_page_layer)

    # Also search in meta layers (layer1, layer4) which contain auspost ele
    for elem in root.iter():
        tag = elem.tag.split('}')[-1] if '}' in elem.tag else elem.tag
        if tag == 'g':
            elem_id = elem.get('id', '')
            inkscape_label = None
            for attr_name, attr_value in elem.attrib.items():
                if attr_name.endswith('label'):
                    inkscape_label = attr_value
                    break
            # Check for meta layers (page 0 meta, page 1 meta)
            if elem_id in ['layer1', 'layer4'] or (inkscape_label and 'meta
                # Check if this meta layer matches our page
                if (page_num % 2 == 0 and ('page 0' in str(inkscape_label)
                   (page_num % 2 == 1 and ('page 1' in str(inkscape_label)
                    find_auspost_elements(elem)

    # Fallback: if still not found, search entire document
    if auspost_rect is None:
        find_auspost_elements(root)

    if auspost_rect is None:
        raise ValueError("Could not find 'auspost' rect in template SVG")

    # Get auspost rect position and dimensions
    auspost_x = float(auspost_rect.get('x', 0))
    auspost_y = float(auspost_rect.get('y', 0))
    auspost_width = float(auspost_rect.get('width', 200))
    auspost_height = float(auspost_rect.get('height', 50))

    # Check if the parent layer has a transform that affects coordinates
    # For calibration_page-coloured-1.svg, layer3 and layer4 have transform
    # We need to find the transform by checking parent layers
    import re
    # Find the parent layer (g element) that contains the auspost_rect
    parent_layer = None
    for elem in root.iter():
        if elem == auspost_rect:
            continue
        # Check if auspost_rect is a child of this element
        for child in elem:
            if child == auspost_rect:
                tag = elem.tag.split('}')[-1] if '}' in elem.tag else elem.
                if tag == 'g':
                    parent_layer = elem
                    break
```

```
            if parent_layer is not None:
                break

        # Check for transform in parent layer and apply it
        label_transform_x = 0
        label_transform_y = 0
        if parent_layer is not None:
            transform_attr = parent_layer.get('transform', '')
            if transform_attr and 'translate' in transform_attr:
                # Extract translate values (e.g., "translate(-880)" or "transla
                match = re.search(r'translate\(([^)]+)\)', transform_attr)
                if match:
                    translate_values = match.group(1).split(',')
                    translate_x = float(translate_values[0].strip())
                    translate_y = float(translate_values[1].strip()) if len(tra
                    auspost_x += translate_x
                    auspost_y += translate_y
                    # Store transform for label text adjustment
                    label_transform_x = translate_x
                    label_transform_y = translate_y

        # Get auspost_label text position if it exists
        label_x = None
        label_y = None
        if auspost_label_text is not None:
            label_x = float(auspost_label_text.get('x', auspost_x))
            label_y = float(auspost_label_text.get('y', auspost_y + auspost_hei
            # Check for tspan inside text element
            for tspan in auspost_label_text.iter():
                tspan_tag = tspan.tag.split('}')[-1] if '}' in tspan.tag else t
                if tspan_tag == 'tspan':
                    tspan_x = tspan.get('x')
                    tspan_y = tspan.get('y')
                    if tspan_x is not None:
                        label_x = float(tspan_x)
                    if tspan_y is not None:
                        label_y = float(tspan_y)
                    break
            # Apply the same transform to label coordinates
            label_x += label_transform_x
            label_y += label_transform_y

        # Calculate scaling factor from SVG to PDF page
        scale_x = page_rect.width / svg_width
        scale_y = page_rect.height / svg_height

        # Scale auspost position to PDF coordinates
        pdf_auspost_x = auspost_x * scale_x
        pdf_auspost_y = auspost_y * scale_y
        pdf_auspost_width = auspost_width * scale_x
        pdf_auspost_height = auspost_height * scale_y

        # Scale label position to PDF coordinates
        pdf_label_x = label_x * scale_x if label_x is not None else None
        pdf_label_y = label_y * scale_y if label_y is not None else None
```

```
            # Hardcoded default font style as backup (used for SVG background)
            # Change DEFAULT_FONT_FAMILY to your desired font
            DEFAULT_FONT_FAMILY = 'Space Mono'  # Font name for SVG
            DEFAULT_FONT_SIZE = '13.3333px'
            DEFAULT_FILL_COLOR = '#cccccc'

            # Update the label text in the SVG template (so it's part of the backgr
            if auspost_label_text is not None:
                import re
                # Set font on the parent text element
                text_style = auspost_label_text.get('style', '')
                if text_style:
                    # Ensure font-family is in text element style
                    if 'font-family' not in text_style:
                        if text_style and not text_style.endswith(';'):
                            text_style += ';'
                        text_style += f'font-family:{DEFAULT_FONT_FAMILY}'
                    else:
                        # Override font-family in text element
                        text_style = re.sub(r'font-family:[^;]+', f'font-family:{DE
                    auspost_label_text.set('style', text_style)
                else:
                    # Create style for text element
                    text_style_parts = [
                        f'font-size:{DEFAULT_FONT_SIZE}',
                        f'font-family:{DEFAULT_FONT_FAMILY}',
                        f'fill:{DEFAULT_FILL_COLOR}',
                        f'fill-opacity:1'
                    ]
                    auspost_label_text.set('style', ';'.join(text_style_parts))

                # Also set font-family as a direct attribute
                auspost_label_text.set('font-family', DEFAULT_FONT_FAMILY)

                # Find existing tspan and update its text content
                tspan_found = False
                for tspan in auspost_label_text.iter():
                    tag_name = tspan.tag.split('}')[-1] if '}' in tspan.tag else ts
                    if tag_name == 'tspan':
                        # Update text content
                        tspan.text = selected_uuid
                        tspan.tail = None

                        # Ensure style attribute has font-family
                        original_style = tspan.get('style', '')
                        if original_style:
                            original_style = re.sub(r'font-family:[^;]+', f'font-fa
                            if 'font-family' not in original_style:
                                if original_style and not original_style.endswith('
                                    original_style += ';'
                                original_style += f'font-family:{DEFAULT_FONT_FAMIL
                            tspan.set('style', original_style)
                        else:
                            # Create style with defaults
                            style_parts = [
                                f'font-size:{DEFAULT_FONT_SIZE}',
```

```
                        f'font-family:{DEFAULT_FONT_FAMILY}',
                        f'fill:{DEFAULT_FILL_COLOR}',
                        f'fill-opacity:1'
                    ]
                    tspan.set('style', ';'.join(style_parts))
                tspan_found = True
                break

        # If no tspan found, create one
        if not tspan_found:
            svg_namespace = None
            if root.tag.startswith('{'):
                svg_namespace = root.tag.split('}')[0].strip('{')
            else:
                svg_namespace = root.attrib.get('xmlns', 'http://www.w3.org

            if svg_namespace:
                tspan_elem = ET.Element('{' + svg_namespace + '}tspan')
            else:
                tspan_elem = ET.Element('tspan')

            tspan_x = auspost_label_text.get('x')
            tspan_y = auspost_label_text.get('y')
            if tspan_x:
                tspan_elem.set('x', tspan_x)
            if tspan_y:
                tspan_elem.set('y', tspan_y)

            tspan_elem.text = selected_uuid
            auspost_label_text.append(tspan_elem)

    # Render template as background image (underlay)
    # Save template to temp file for rendering
    temp_template_svg = os.path.join(output_dir, f"temp_template_{page_num}
    xml_str = ET.tostring(root, encoding='unicode')
    dom = minidom.parseString(xml_str)
    pretty_xml = dom.toprettyxml(indent="  ")
    with open(page_template_path, 'r', encoding='utf-8') as f:
        original_content = f.read()
        has_xml_declaration = original_content.strip().startswith('<?xml')
    if not has_xml_declaration:
        lines = pretty_xml.split('\n')
        if lines[0].startswith('<?xml'):
            pretty_xml = '\n'.join(lines[1:])
    with open(temp_template_svg, 'w', encoding='utf-8') as f:
        f.write(pretty_xml)
    temp_files.append(temp_template_svg)

    # Render template SVG to pixmap for background
    zoom = 2.0
    mat = fitz.Matrix(zoom, zoom)
    template_doc = fitz.open(temp_template_svg)
    template_page = template_doc[0]
    template_pix = template_page.get_pixmap(matrix=mat)
    template_doc.close()
```

```
            # Create new PDF page with template as background
            temp_pdf = os.path.join(output_dir, f"temp_page_{page_num}.pdf")
            new_doc = fitz.open()
            new_page = new_doc.new_page(width=page_rect.width, height=page_rect.hei

            # Insert template as background image (underlay) - this includes the la
            new_page.insert_image(fitz.Rect(0, 0, page_rect.width, page_rect.height

            # Encode the barcode BEFORE overlaying PDF (so we have the dimensions)
            uuid_for_barcode = selected_uuid.replace('-', '')
            bars = encode_4state_barcode(uuid_for_barcode)
            num_bars = len(bars)

            # Calculate barcode dimensions to fit in auspost area
            # Make bars thinner with more spacing for better legibility
            available_width = pdf_auspost_width * 0.95
            # Make bars thinner by increasing the divisor
            bar_width = (available_width * 0.85) / (num_bars * 3) if num_bars > 0 e
            # Ensure minimum bar width for legibility
            if bar_width < 1.5:
                bar_width = 1.5
            space = bar_width * 1.0  # More space between bars
            total_barcode_width = num_bars * bar_width + (num_bars - 1) * space

            height_full = pdf_auspost_height * 0.9
            height_asc = height_full * 0.8
            height_desc = height_full * 0.8
            tracker_height = height_full * 0.2

            # Center barcode in auspost area
            barcode_start_x = pdf_auspost_x + (pdf_auspost_width - total_barcode_wi
            barcode_start_y = pdf_auspost_y + (pdf_auspost_height - height_full) /
            y_tracker = barcode_start_y + (height_full - tracker_height) / 2

            # Draw barcode bars BEFORE overlaying PDF (so it's in the background)
            # Use black color for maximum legibility
            barcode_color = (0.6, 0.6, 0.6)  # Dark gray (RGB values must be 0.0-1.
            x_pos = barcode_start_x
            for bar in bars:
                if bar == 'F':  # Full
                    rect = fitz.Rect(x_pos, barcode_start_y, x_pos + bar_width, bar
                    new_page.draw_rect(rect, color=barcode_color, fill=barcode_colo
                elif bar == 'A':  # Ascender
                    rect = fitz.Rect(x_pos, barcode_start_y, x_pos + bar_width, bar
                    new_page.draw_rect(rect, color=barcode_color, fill=barcode_colo
                elif bar == 'D':  # Descender
                    rect = fitz.Rect(x_pos, barcode_start_y + height_full - height_
                    new_page.draw_rect(rect, color=barcode_color, fill=barcode_colo
                elif bar == 'T':  # Tracker
                    rect = fitz.Rect(x_pos, y_tracker, x_pos + bar_width, y_tracker
                    new_page.draw_rect(rect, color=barcode_color, fill=barcode_colo
                x_pos += bar_width + space

            # Overlay original PDF page content on top (preserving vector)
            # Use show_pdf_page to insert the original page on top of background
            new_page.show_pdf_page(fitz.Rect(0, 0, page_rect.width, page_rect.heigh
```

```
            # Save the new PDF page
            new_doc.save(temp_pdf)
            new_doc.close()
            temp_files.append(temp_pdf)

            # Update CSV
            rows[selected_index]['entity'] = f"{os.path.basename(output_path)}_page
            rows[selected_index]['state'] = 'active'

            signed_pages.append((page_num + 1, selected_uuid, temp_pdf))

        # Merge all PDF pages back together
        merged_doc = fitz.open()
        for page_num, uuid, temp_pdf_path in signed_pages:
            page_doc = fitz.open(temp_pdf_path)
            merged_doc.insert_pdf(page_doc)
            page_doc.close()

        # Save merged PDF
        merged_doc.save(output_path)
        merged_doc.close()

        # Write back to CSV
        with open(csv_path, 'w', encoding='utf-8', newline='') as f:
            fieldnames = ['uuid', 'entity', 'state']
            writer = csv.DictWriter(f, fieldnames=fieldnames)
            writer.writeheader()
            writer.writerows(rows)

    finally:
        # Clean up temporary files
        for temp_file in temp_files:
            try:
                if os.path.exists(temp_file):
                    os.remove(temp_file)
            except Exception as e:
                print(f"Warning: Could not remove temp file {temp_file}: {e}")

    doc.close()

    return (output_path, signed_pages)


def sign_with_next_uuid(csv_path, input_svg_path, output_filename=None, output_dir=
    """
    Sign an SVG with the next available UUID from the CSV file.

    Args:
        csv_path: Path to the v4_uuids.csv file
        input_svg_path: Path to the input SVG file
        output_filename: Custom output filename (without extension). If None, uses
        output_dir: Directory to save output (defaults to same as input_svg_path)

    Returns:
        tuple: (uuid, output_path) or None if no available UUID found
```

```
    """
    # Read the CSV file
    rows = []
    with open(csv_path, 'r', encoding='utf-8') as f:
        reader = csv.DictReader(f)
        rows = list(reader)

    # Find the next available UUID (empty entity and state)
    selected_uuid = None
    selected_index = None
    for i, row in enumerate(rows):
        if not row.get('entity', '').strip() and not row.get('state', '').strip():
            selected_uuid = row['uuid']
            selected_index = i
            break

    if selected_uuid is None:
        return None

    # Remove hyphens from UUID for barcode encoding
    uuid_for_barcode = selected_uuid.replace('-', '')

    # Determine output path
    if output_dir is None:
        output_dir = os.path.dirname(input_svg_path) or '.'

    if output_filename is None:
        output_filename = selected_uuid.replace('-', '_')

    output_path = os.path.join(output_dir, f"{output_filename}.svg")

    # Generate the barcode (use UUID with hyphens for label, without hyphens for en
    insert_barcode_into_svg(input_svg_path, uuid_for_barcode, output_path, label_te

    # Update the CSV (store just the filename, not the full path)
    output_filename_with_ext = f"{output_filename}.svg"
    rows[selected_index]['entity'] = output_filename_with_ext
    rows[selected_index]['state'] = 'active'

    # Write back to CSV
    with open(csv_path, 'w', encoding='utf-8', newline='') as f:
        fieldnames = ['uuid', 'entity', 'state']
        writer = csv.DictWriter(f, fieldnames=fieldnames)
        writer.writeheader()
        writer.writerows(rows)

    return (selected_uuid, output_path)


def sign_with_uuid(uuid, input_svg_path, output_filename=None, output_dir=None):
    """
    Sign an SVG with a specific UUID.

    Args:
        uuid: UUID string to use for signing
        input_svg_path: Path to the input SVG file
```

```
        output_filename: Custom output filename (without extension). If None, uses
        output_dir: Directory to save output (defaults to same as input_svg_path)

    Returns:
        tuple: (uuid, output_path)
    """
    # Remove hyphens from UUID for barcode encoding
    uuid_for_barcode = uuid.replace('-', '')

    # Determine output path
    if output_dir is None:
        output_dir = os.path.dirname(input_svg_path) or '.'

    if output_filename is None:
        output_filename = uuid.replace('-', '_')

    output_path = os.path.join(output_dir, f"{output_filename}.svg")

    # Generate the barcode (use UUID with hyphens for label, without hyphens for en
    insert_barcode_into_svg(input_svg_path, uuid_for_barcode, output_path, label_te

    return (uuid, output_path)


def main():
    """
    Command-line interface for signing documents with UUIDs.
    """
    parser = argparse.ArgumentParser(
        description='Sign an SVG document with a UUID barcode',
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog="""
Examples:
  # Use next available UUID from CSV
  python homework_signer.py

  # Use specific UUID
  python homework_signer.py --uuid "b09b61df-5c2e-442f-a6e8-0c8cb600642f"

  # Custom output filename
  python homework_signer.py --output "my_document"

  # Custom CSV file
  python homework_signer.py --csv "custom_uuids.csv"

  # Process PDF: extract pages, add barcode to each, merge back
  python homework_signer.py --pdf "document.pdf" --pdf-output "document_signed.pdf"
        """
    )

    parser.add_argument(
        '--uuid',
        type=str,
        help='Specific UUID to use for signing (if not provided, uses next availabl
    )
```

```
    parser.add_argument(
        '--csv',
        type=str,
        default='v4_uuids.csv',
        help='Path to CSV file with UUIDs (default: v4_uuids.csv)'
    )

    parser.add_argument(
        '--input',
        type=str,
        default='calibration_page-coloured.svg',
        help='Path to input SVG template (default: calibration_page-coloured.svg)'
    )

    parser.add_argument(
        '--output',
        type=str,
        help='Output filename without extension (default: uses UUID or next availab
    )

    parser.add_argument(
        '--output-dir',
        type=str,
        help='Directory to save output file (default: same as input file directory)
    )

    parser.add_argument(
        '--pdf',
        type=str,
        help='Process a PDF file: extract pages, add unique barcode to each, and me
    )

    parser.add_argument(
        '--pdf-output',
        type=str,
        help='Output path for signed PDF (default: input name with _signed suffix)'
    )

    args = parser.parse_args()

    try:
        if args.pdf:
            # Process PDF file
            if not os.path.exists(args.pdf):
                print(f"✗ Error: PDF file not found: {args.pdf}")
                sys.exit(1)

            if not os.path.exists(args.csv):
                print(f"✗ Error: CSV file not found: {args.csv}")
                sys.exit(1)

            output_path, signed_pages = sign_pdf_with_barcodes(
                pdf_path=args.pdf,
                csv_path=args.csv,
                output_path=args.pdf_output,
                output_dir=args.output_dir,
```

```
                template_svg_path=args.input
            )

            print(f"✓ Processed {len(signed_pages)} pages:")
            for page_num, uuid, _ in signed_pages:
                print(f"  Page {page_num}: UUID {uuid}")
            print(f"✓ Merged PDF saved to: {output_path}")
            print(f"✓ CSV updated with {len(signed_pages)} UUIDs")

        elif args.uuid:
            # Use specified UUID
            result = sign_with_uuid(
                uuid=args.uuid,
                input_svg_path=args.input,
                output_filename=args.output,
                output_dir=args.output_dir
            )
            uuid, output_path = result
            print(f"✓ Document signed with UUID: {uuid}")
            print(f"✓ Output saved to: {output_path}")
        else:
            # Use next available UUID from CSV
            if not os.path.exists(args.csv):
                print(f"✗ Error: CSV file not found: {args.csv}")
                sys.exit(1)

            result = sign_with_next_uuid(
                csv_path=args.csv,
                input_svg_path=args.input,
                output_filename=args.output,
                output_dir=args.output_dir
            )

            if result:
                uuid, output_path = result
                print(f"✓ Document signed with next available UUID: {uuid}")
                print(f"✓ Output saved to: {output_path}")
                print(f"✓ CSV updated: entity={os.path.basename(output_path)}, stat
            else:
                print("✗ Error: No available UUIDs found in CSV file")
                sys.exit(1)

    except Exception as e:
        import traceback
        print(f"✗ Error: {e}")
        if str(e) == "":
            traceback.print_exc()
        sys.exit(1)


if __name__ == '__main__':
    main()
```

In [ ]:
```
from homework_signer import sign_with_next_uuid

# Sign a document with the next available UUID
```

```python
# You can customize the output filename
result = sign_with_next_uuid(
    csv_path='v4_uuids.csv',
    input_svg_path='calibration_page-coloured.svg',
    output_filename='document_001'  # Customize this name
)

if result:
    uuid, output_path = result
    print(f"✓ Document signed successfully!")
    print(f"  UUID: {uuid}")
    print(f"  Output file: {output_path}")
    print(f"  CSV updated: entity={output_path}, state=active")
else:
    print("✗ No available UUID found in the CSV file")
```

A4 Example of Custom Anti-Cheat Paper

Music Theory Workbook

Level 0

0. Colour over the Whitespace then Put Stcker over these instructons

12/5/2025

Instruments: Piano, Bass, 6-String Guitar

480c34a3-b388-4685-a642-728a08bf24f8

1. Draw the popsicle stcks (**1 octave**) (/8)

2. Draw the popsicle stcks again (**2 octaves**) (/16)

3. *For a challenge:* Draw some popsicle stcks again (**with only 4 keys**) (/4)

4. Draw the popsicle stcks again (**with the "frst" key labelled**) (/1)

5. *For fun:* Draw the popsicle stcks again (**from 'C' to 'E', colour in the <u>half-tones only</u>**) (--/0)

---

6. What is the actual name of these popsicle stcks? *Hint Below:*

frst listed instrument

Answer: _____ (/1)

7. The guitar belongs to the _____ family, where sound comes from vibratng strings, making it a core member of the plucked string family alongside the banjo, lute, and harp. (/4)

8. Statement: “The guitar and the piano belong to the same family.”

Is that [True], or [False]? (/1)

9. Why are the ____ useful to use even though they are not on the _____ ? *Hints Below:*

[popsicles, Queston 7] (/2)

Answer: _____ (/2)

10. Statement: “I wanted to learn a song is not a good enough reason for learning the guitar.”

Is that [True], or [False]? (/1)

11. Rewrite this in order (**start from ‘C’**) (/12)

A B D F G E C

12. Rewrite this in order (**start from ‘E’**) (/12)

B A G F D C E

13. Why does a string make a noise?

Answer: _____ (/4)

14. What are two half-tones called?

Answer: _____ (/4)

15. Which keys do not black keys between them?

Answers: The pairs _____ and _____ (/2)

16. What are the 3 knobs are your electric guitar for?

Answers: _____, _____, and _____ (/4)

17. Statement: “You should make sure your amp is turned of before plugging in, or unplugging your instrument.”

Is that [True] or [False]? (/1)

18. What are the 4 common strings on the 6-string guitar and the 4-string bass?

Answer: _____, _____, _____, and _____ (/4)

19. Draw the neck of the guitar; with the following present: (/31)

- the frets up to the 13th (/10)

- the dots, as they appear on the neck (/10)

- the numbers of those frets (*you can choose all, or only the important ones*) (/10)

- *the strings are optonal* (--/0)

- at least 3 diferent pen colours (/1)

20. Draw the neck of the guitar again. (**using any combinaton of the previously mentoned**) (/1)

21. *Challenge:* Draw the neck of the guitar in a vertcal orientaton. (/1)