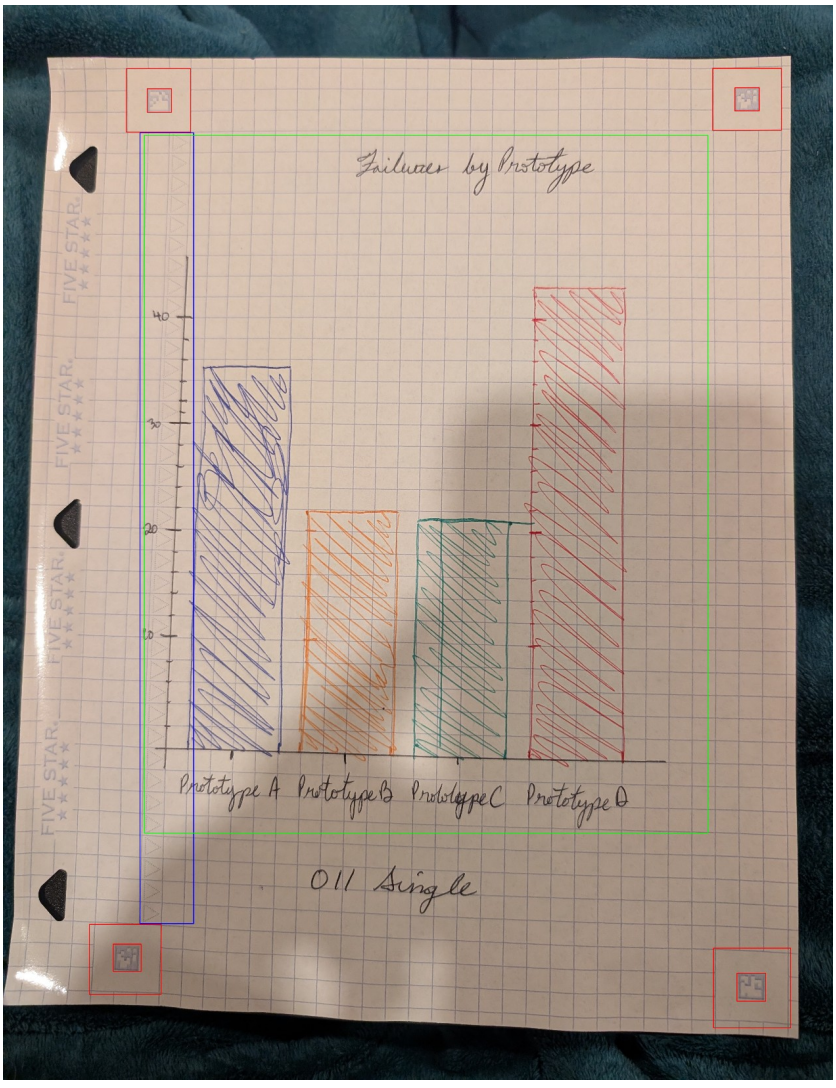
FieldChartOCR: Extraction of Handwritten Charts and Tables



Project Proposal by: Chris Brechin

Prepared for: DTSA-5506

Date: November 18th, 2025

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

Week	Milestone
1	Finalize annotation schema, label initial dataset.
2	**Implement normalization prototype, ntegrate 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, almost finished page normalization.

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 will improve the function to include a guess which will allow the chart area detection to occur. If not, a dataset of corner patterns will need to be annotated for modeling.

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

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_palette)
# 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
```

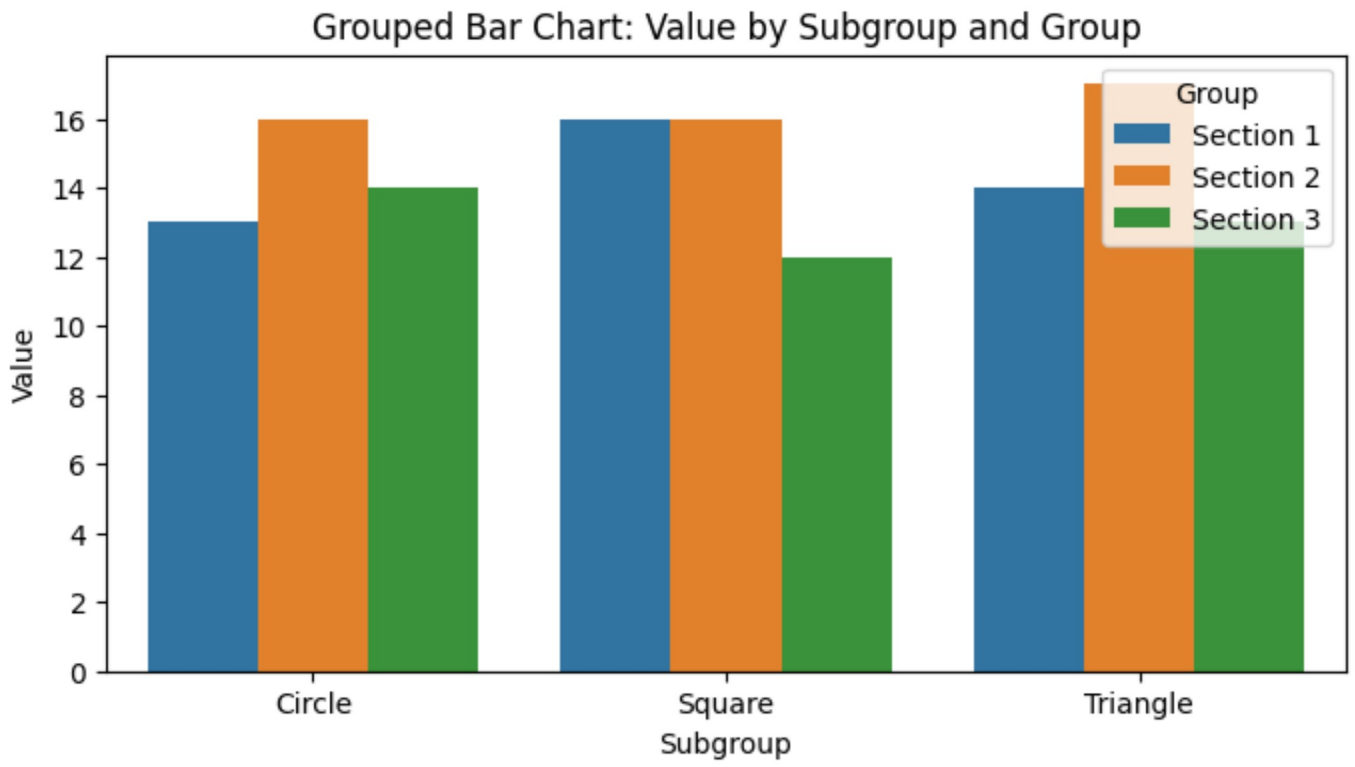
Out[3]:

	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 [4]:
```
plt.figure(figsize=(8, 4))
sns.barplot(
    data=bc_016_grouped,
    x='Subgroup',
    y='Value',
    hue='Group',
    palette=grouped_palette[:len(bc_016_grouped)]
)
```

```
plt.xlabel('Subgroup')
plt.ylabel('Value')
plt.title('Grouped Bar Chart: Value by Subgroup and Group')
plt.legend(title='Group')
plt.show()
```

```
C:\Users\lenovo\AppData\Local\Temp\ipykernel_14072\1973385117.py:2: UserWarning: The palette list has more values (8) than needed (3), which may not be intended.
  sns.barplot(
```



016 Grouped

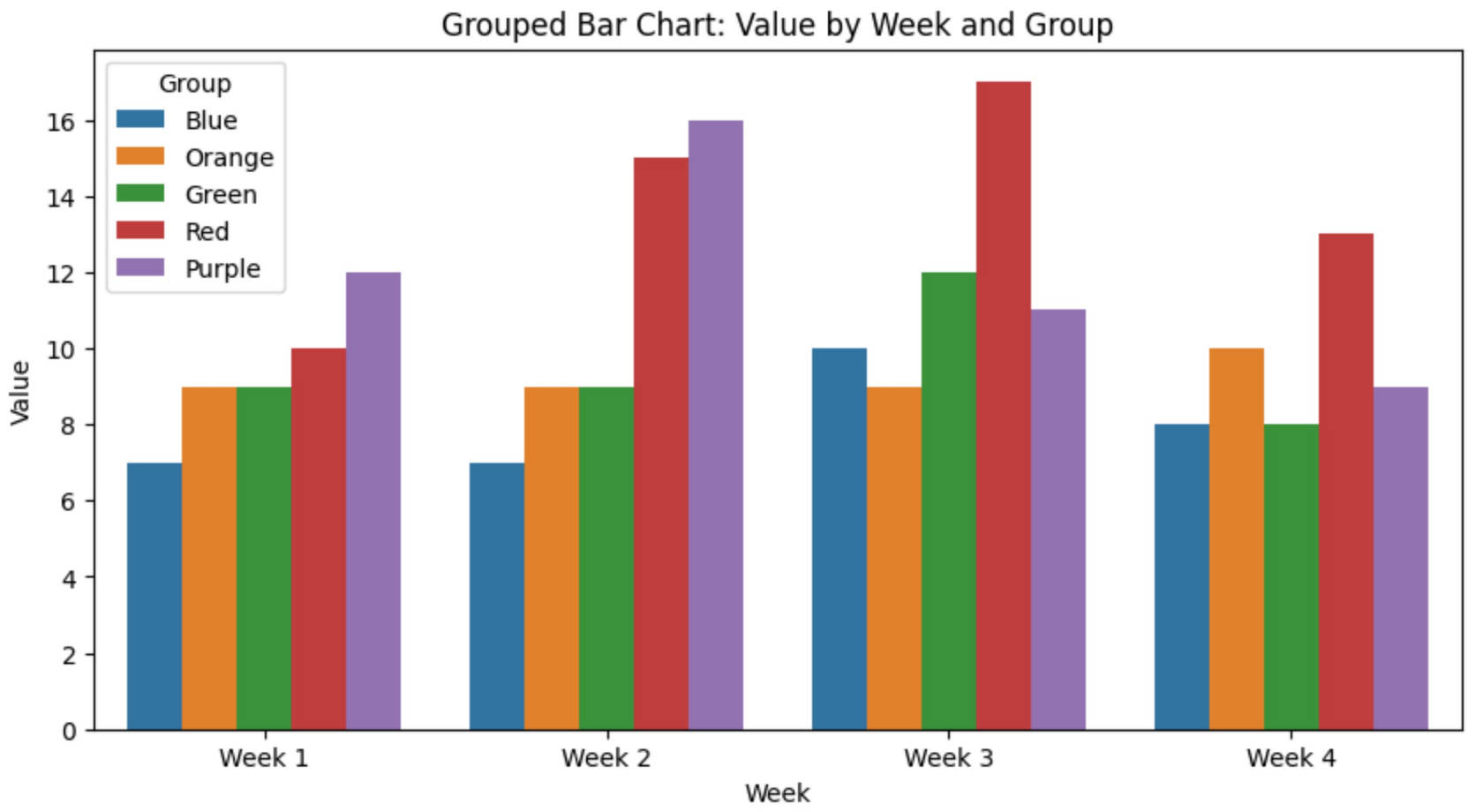
In [5]:
```
## 002

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

bc_017_grouped
```

Out[5]:

	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()
```



017 Grouped

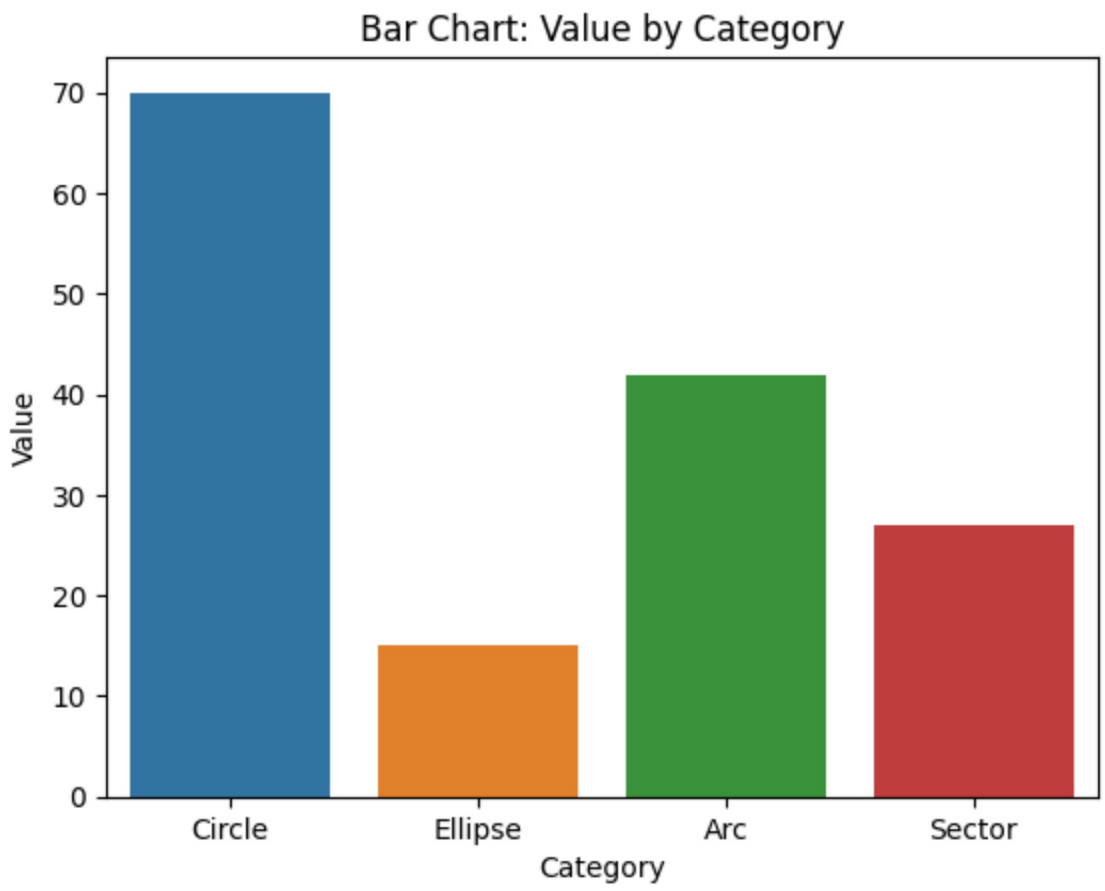
001 Single

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()
```



001 Single

002 Single

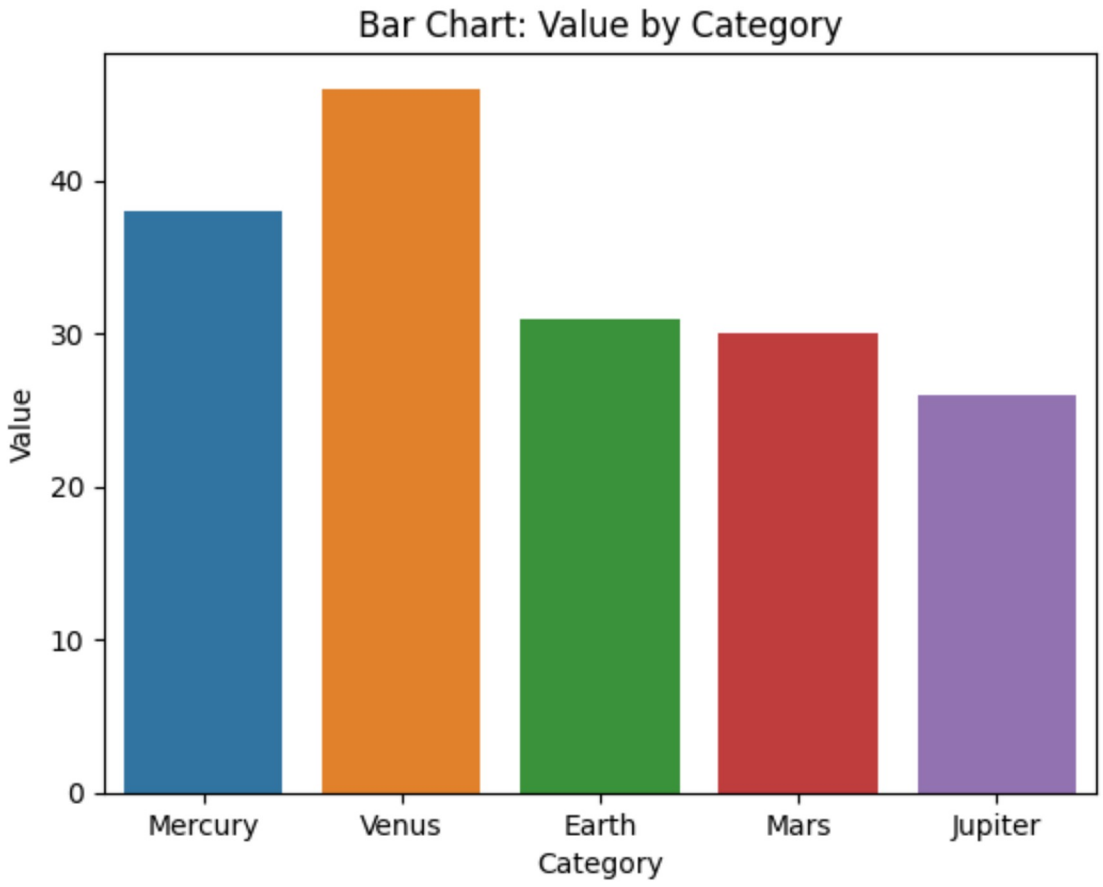
In [9]:
```
bc_002_single = pd.read_csv('./bar_charts/bar_single_002.csv')

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')



002 Single

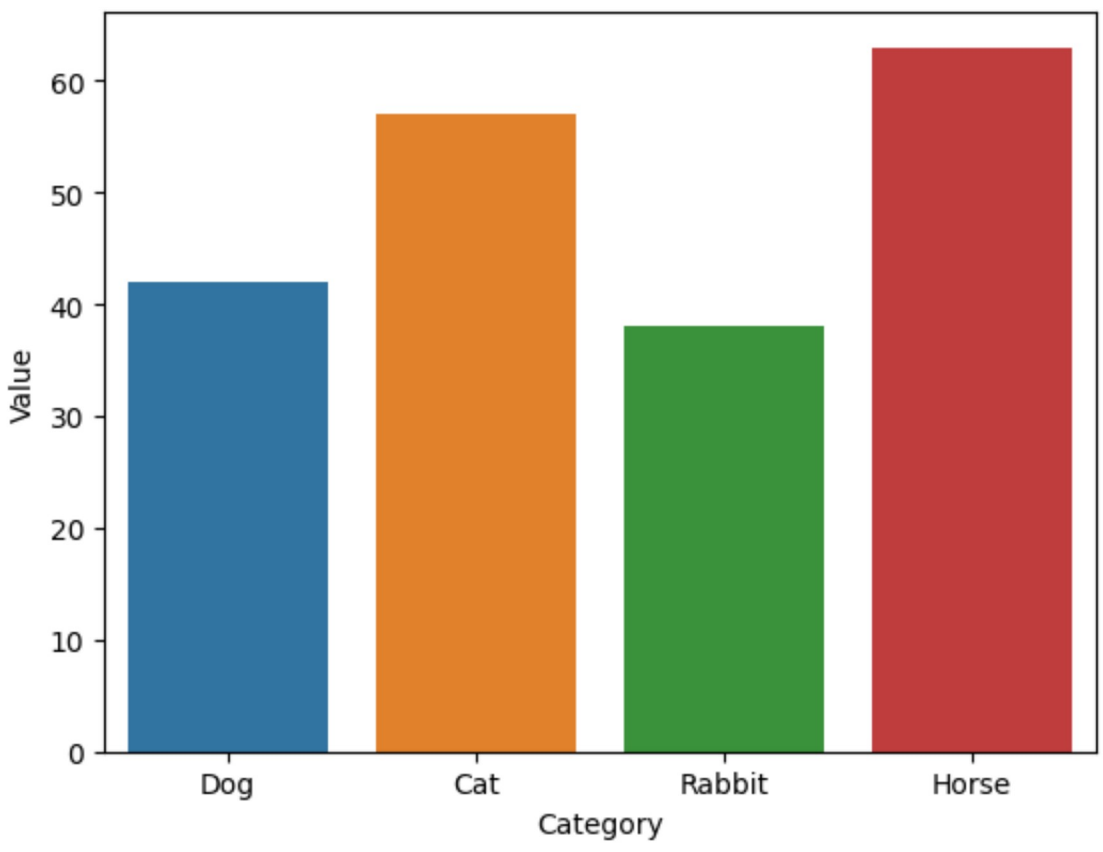
In [11]:
```
bc_003_single = pd.read_csv('./bar_charts/bar_single_003.csv')

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')
```



003 Single

004 Single

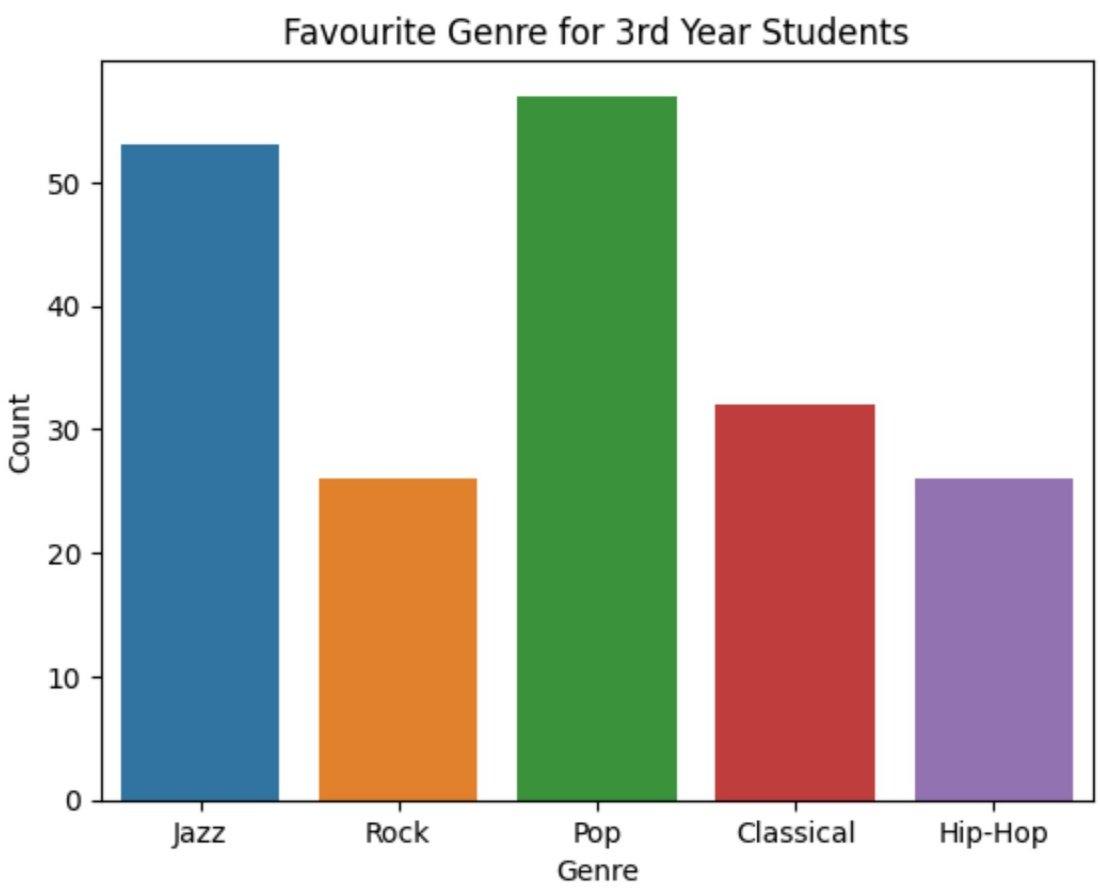
In [13]:
```
bc_004_single = pd.read_csv('./bar_charts/bar_single_004.csv')

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

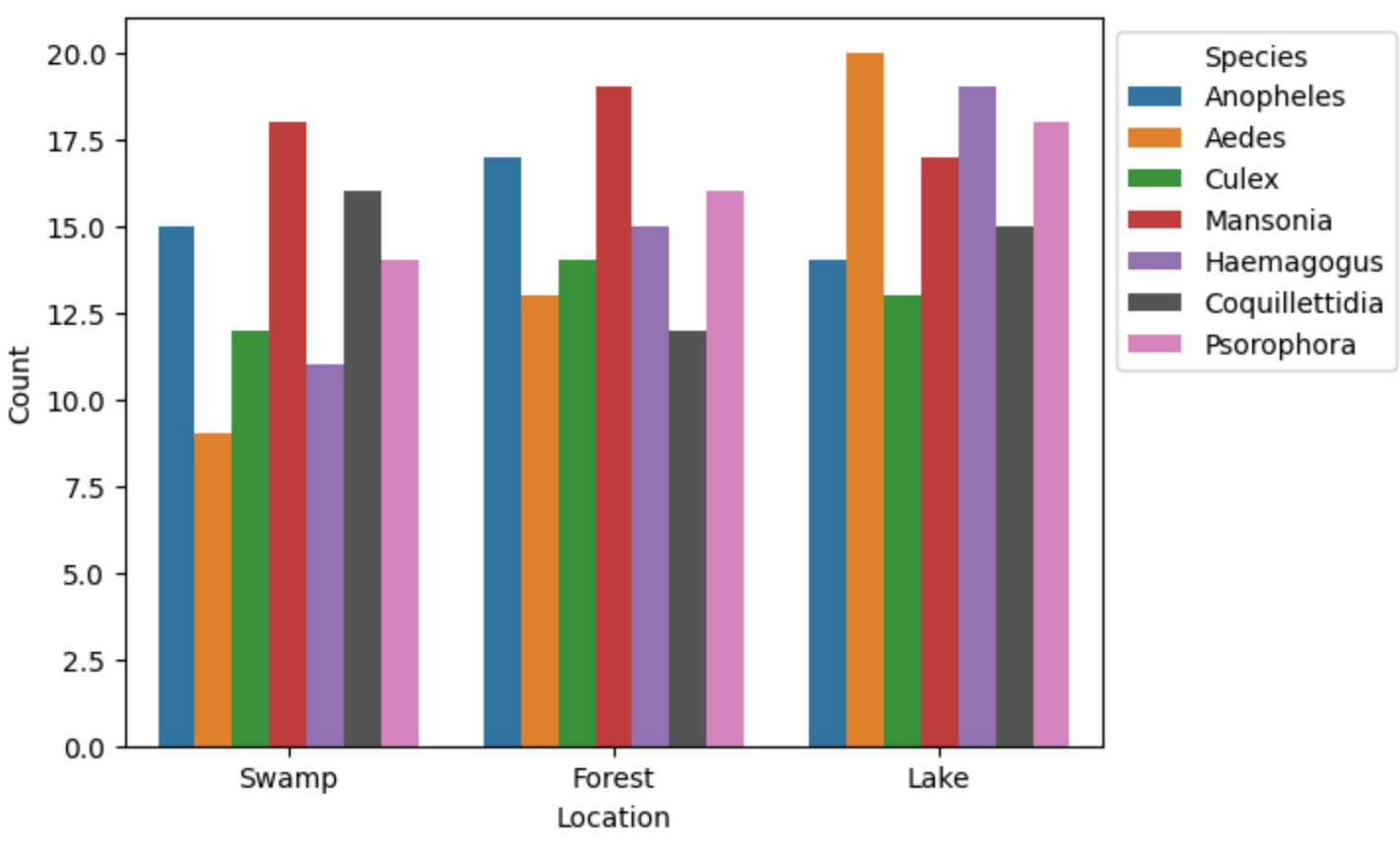
In [15]:
```
bc_028_grouped = pd.read_csv('./bar_charts/bar_grouped_028.csv')

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 size to number of hue groups
)
plt.xlabel('Location')
plt.legend(loc='upper left', bbox_to_anchor=(1, 1), title='Species') # Fix the legend position
```

Out[16]: <matplotlib.legend.Legend at 0x2abfff62990>



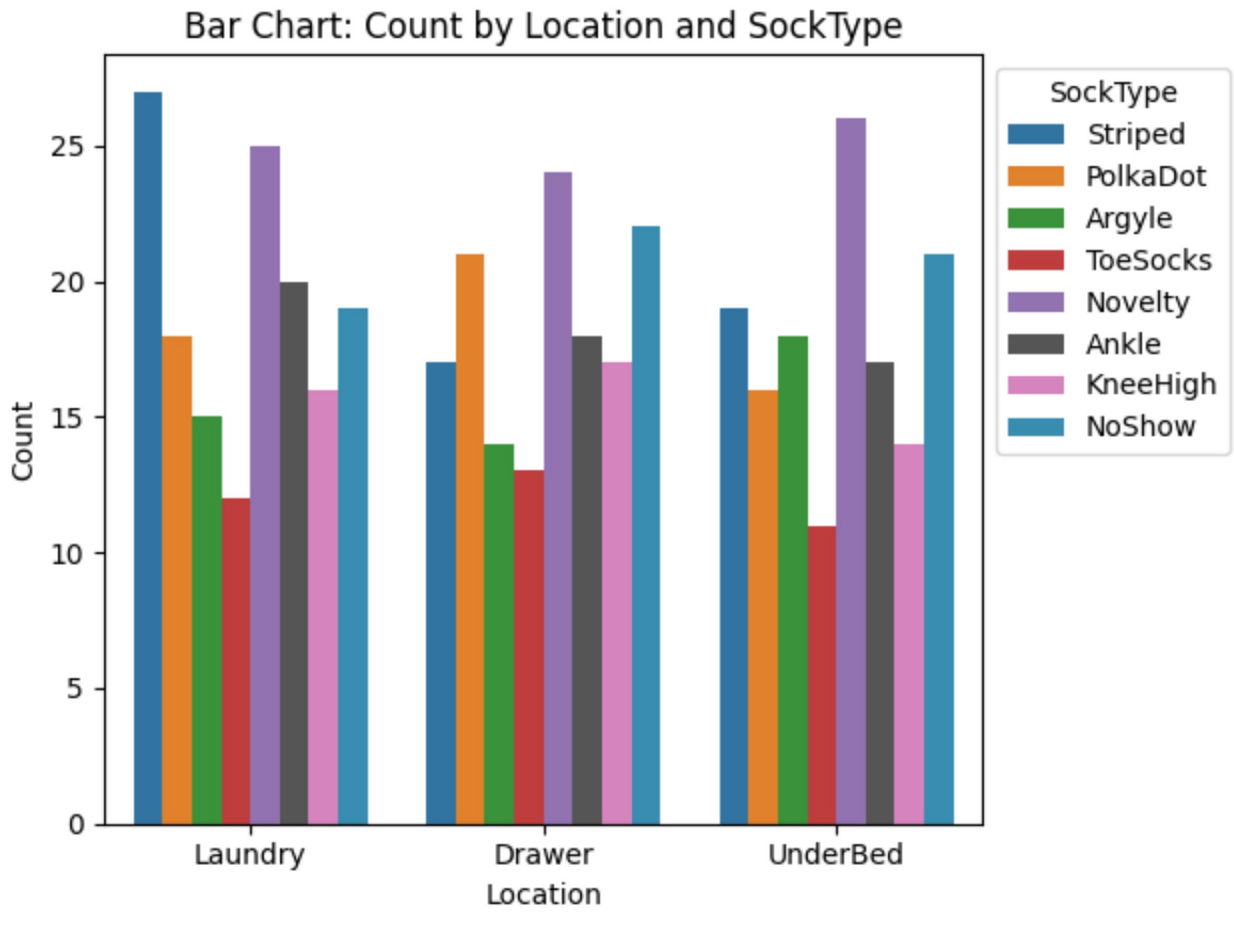
029 Grouped

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

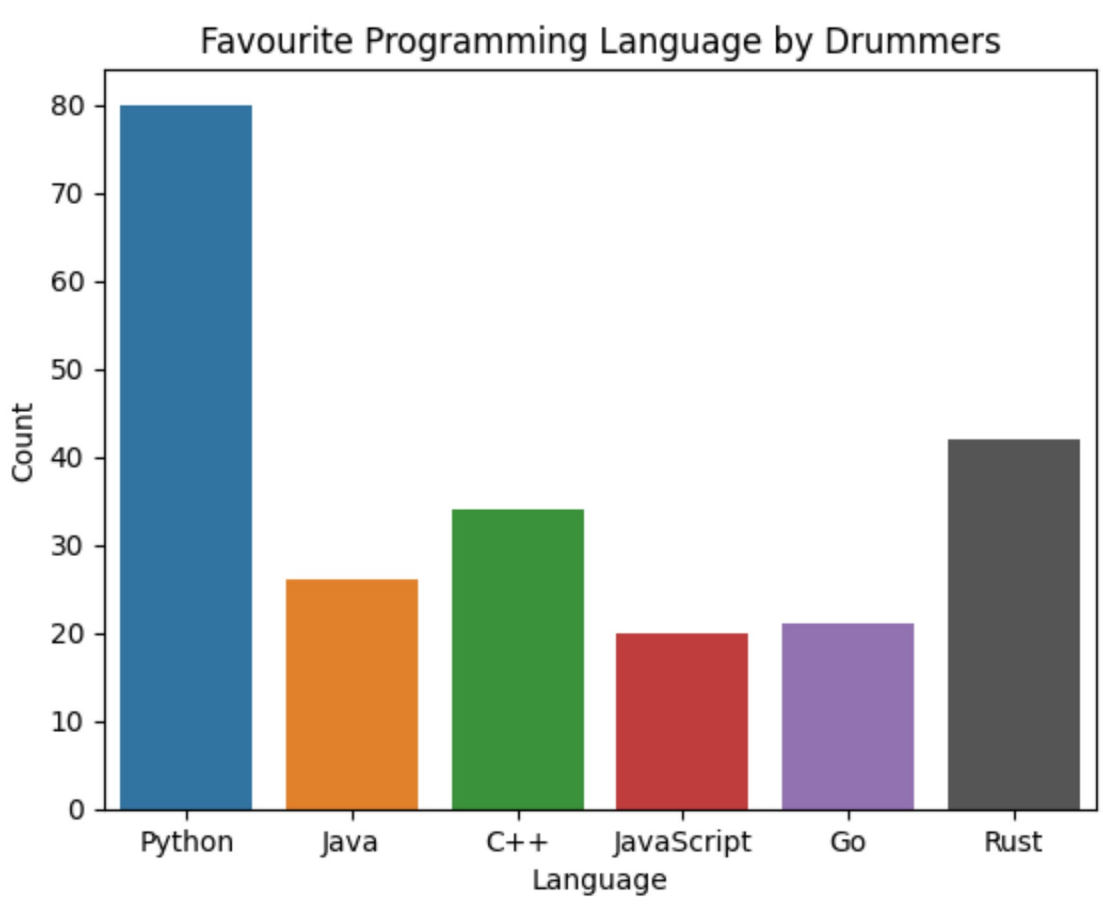
In [19]:
```
bc_005_single = pd.read_csv('./bar_charts/bar_single_005.csv')

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

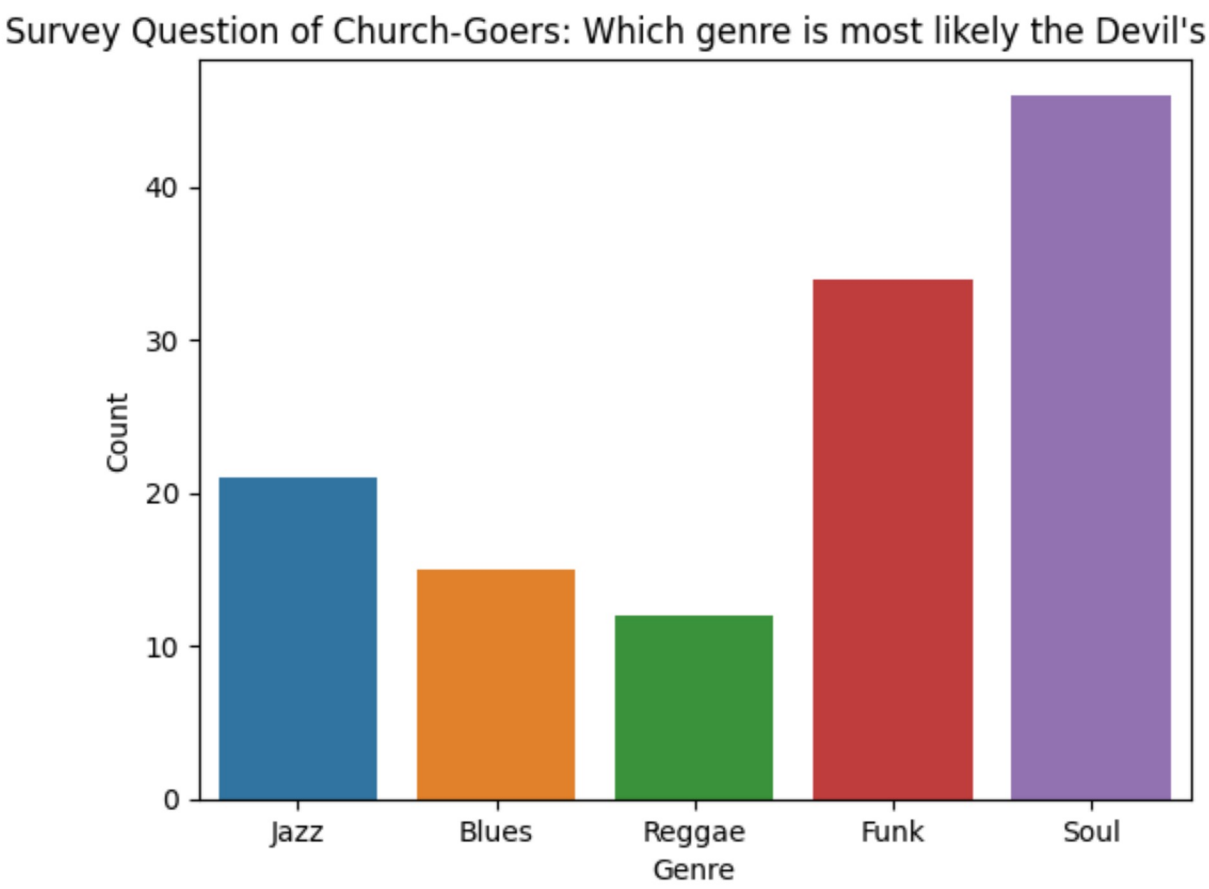
In [21]:
```
bc_006_single = pd.read_csv('./bar_charts/bar_single_006.csv')

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 favourite?')
```

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



007 Single

In [29]:
```
bc_007_single = pd.read_csv("bar_charts/bar_single_007.csv")

bc_007_single
```

Out[29]:

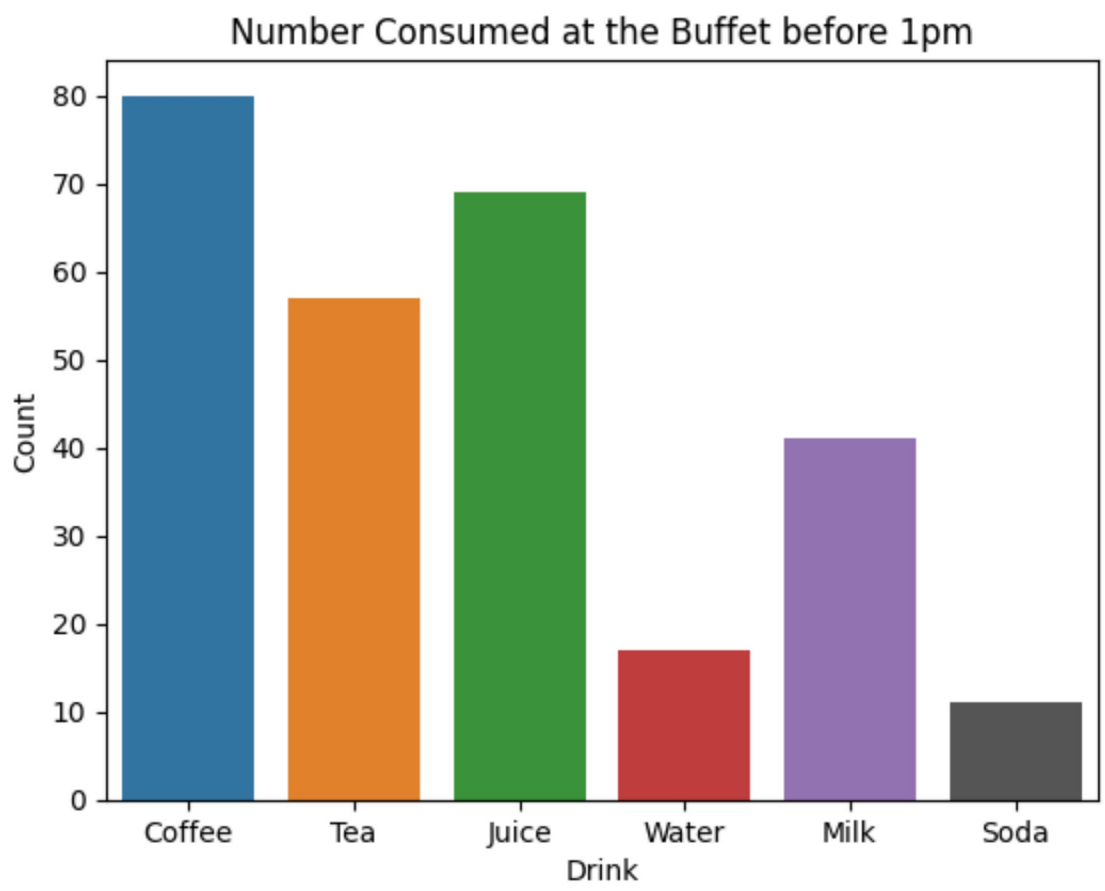
	Drink	Count
0	Coffee	80
1	Tea	57
2	Juice	69
3	Water	17
4	Milk	41
5	Soda	11

007 Single

In [31]:
```
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[31]: Text(0.5, 1.0, 'Number Consumed at the Buffet before 1pm')



008 Single

In [35]:
```
bc_008_single = pd.read_csv("bar_charts/bar_single_008.csv")

bc_008_single
```

Out[35]:

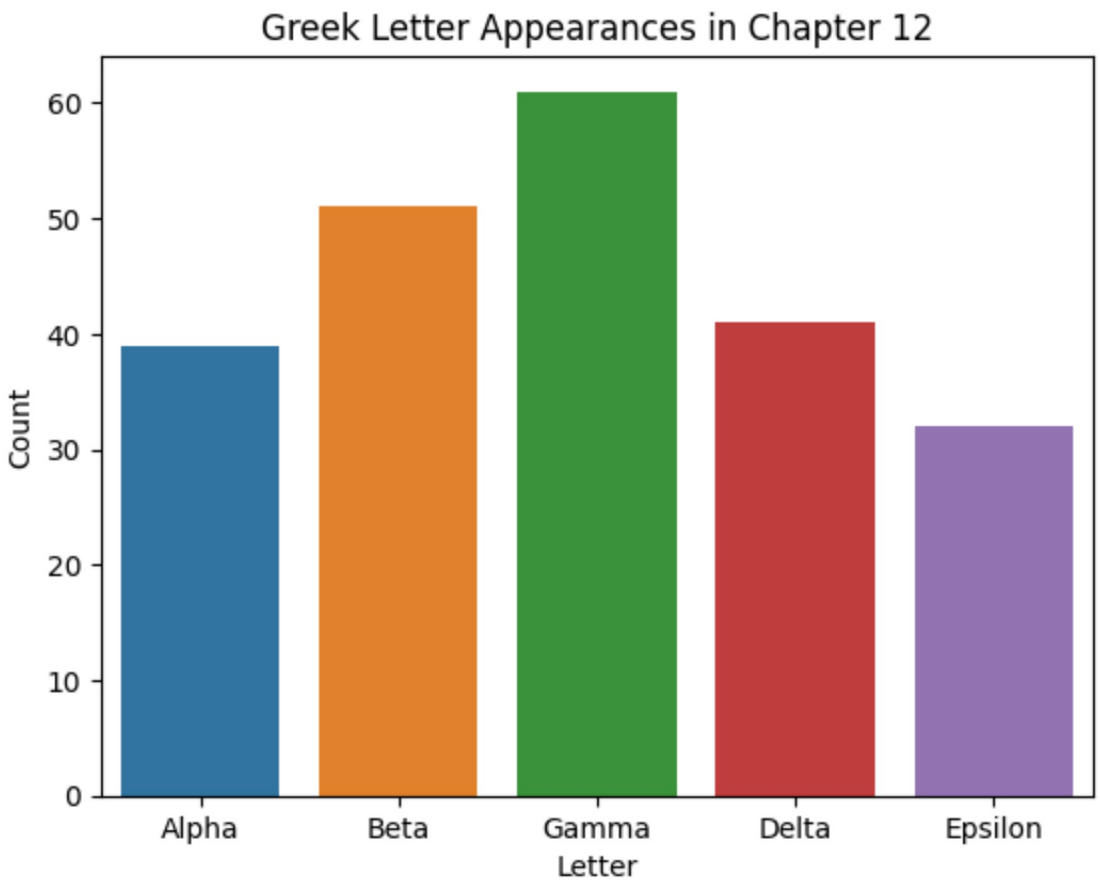
	Letter	Count
0	Alpha	39
1	Beta	51
2	Gamma	61
3	Delta	41
4	Epsilon	32

008 Single

In [38]:
```
sns.barplot(
    data=bc_008_single,
    x='Letter',
    hue='Letter',
    y='Count',
    palette=grouped_palette[:len(bc_007_single)]
)
plt.xlabel('Letter')
plt.title('Greek Letter Appearances in Chapter 12')
```

```
C:\Users\lenovo\AppData\Local\Temp\ipykernel_14072\1925220184.py:1: UserWarning: The palette list has more values (6) than needed (5), which may not be intended.
  sns.barplot(
```

Out[38]: Text(0.5, 1.0, 'Greek Letter Appearances in Chapter 12')



009 Single

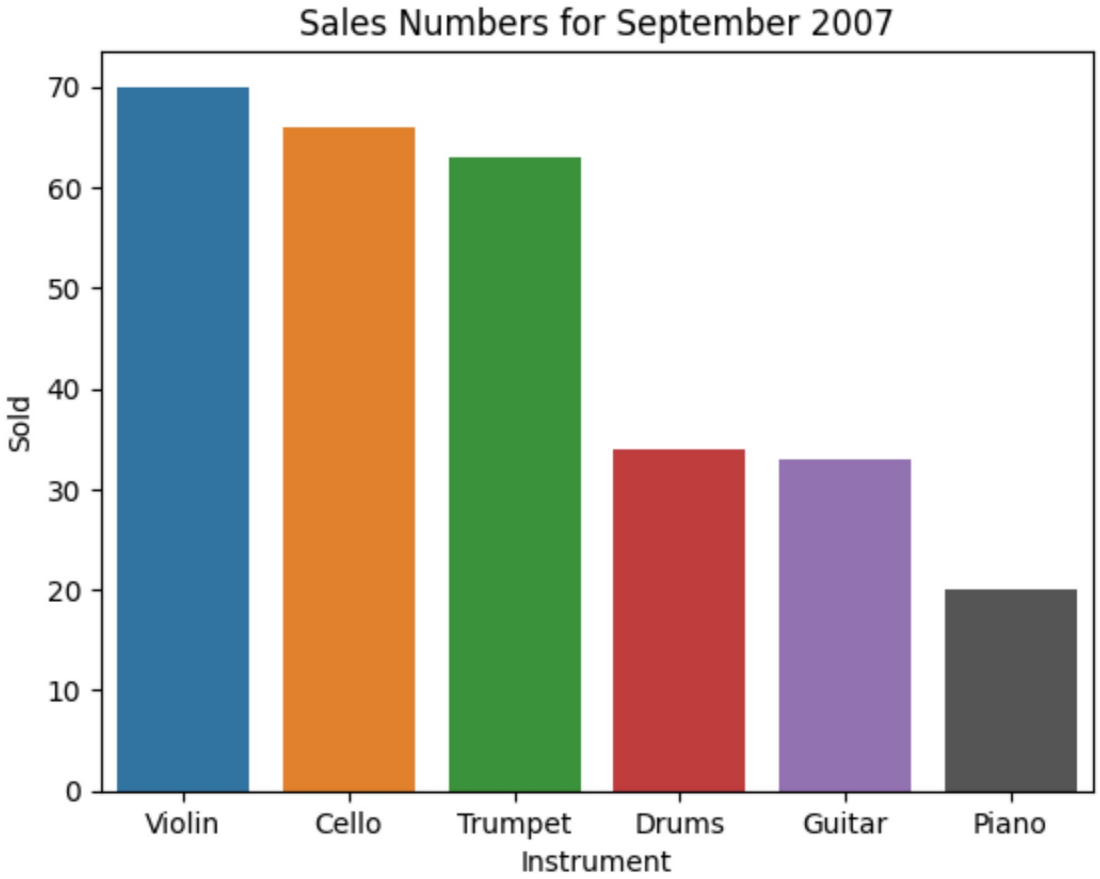
In [43]:
```
bc_009_single = pd.read_csv("bar_charts/bar_single_009.csv")

bc_009_single
```

Out[43]:

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

In [44]:
```
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[44]:
```
Text(0.5, 1.0, 'Sales Numbers for September 2007')
```



010 Single

In [51]:
```
bc_010_single = pd.read_csv("bar_charts/bar_single_010.csv")

bc_010_single
```

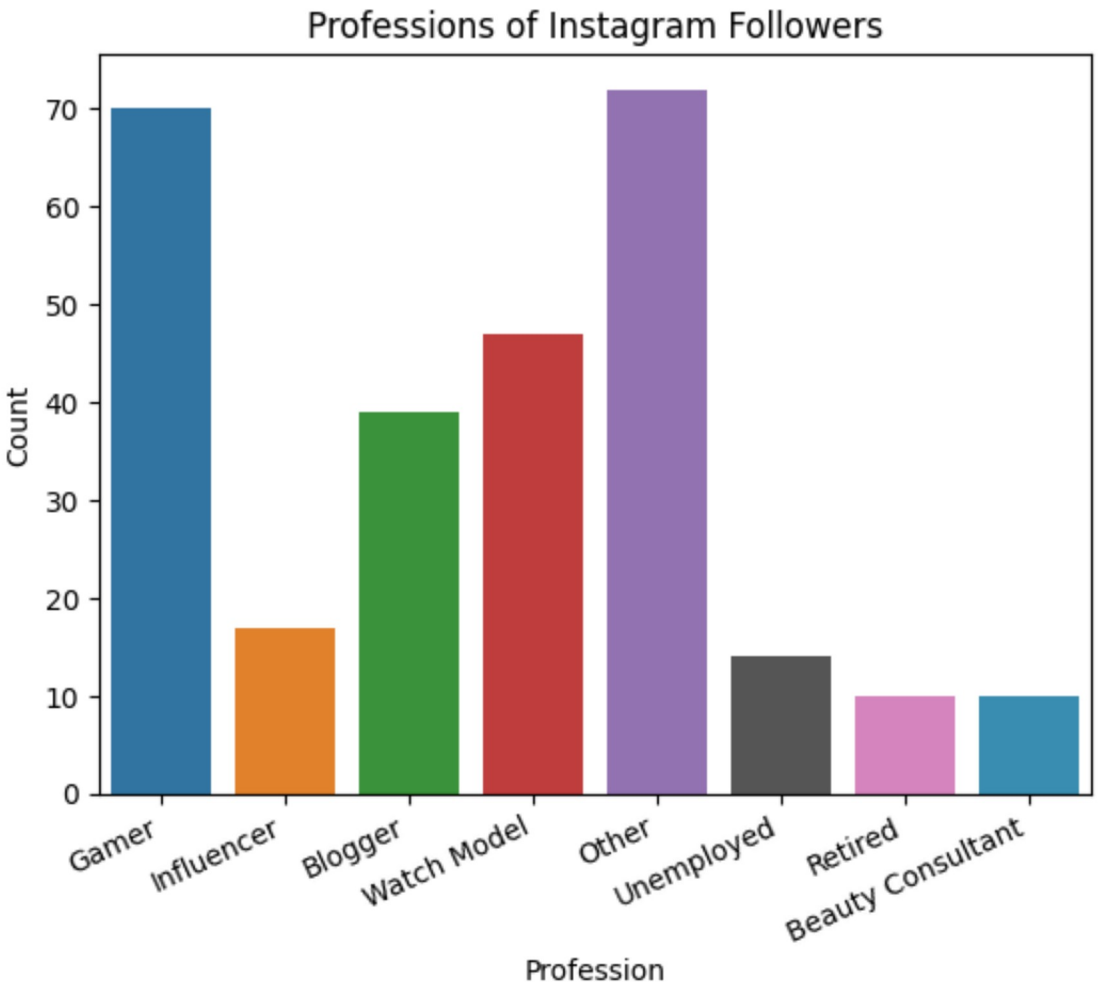
Out[51]:

	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

010 Single

In [52]:
```
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[52]:
```
([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 [54]:
```
bc_011_single = pd.read_csv("bar_charts/bar_single_011.csv")

bc_011_single
```

Out[54]:

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

011 Single

In [55]:
```
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[55]: Text(0.5, 1.0, 'Failures by Prototype')

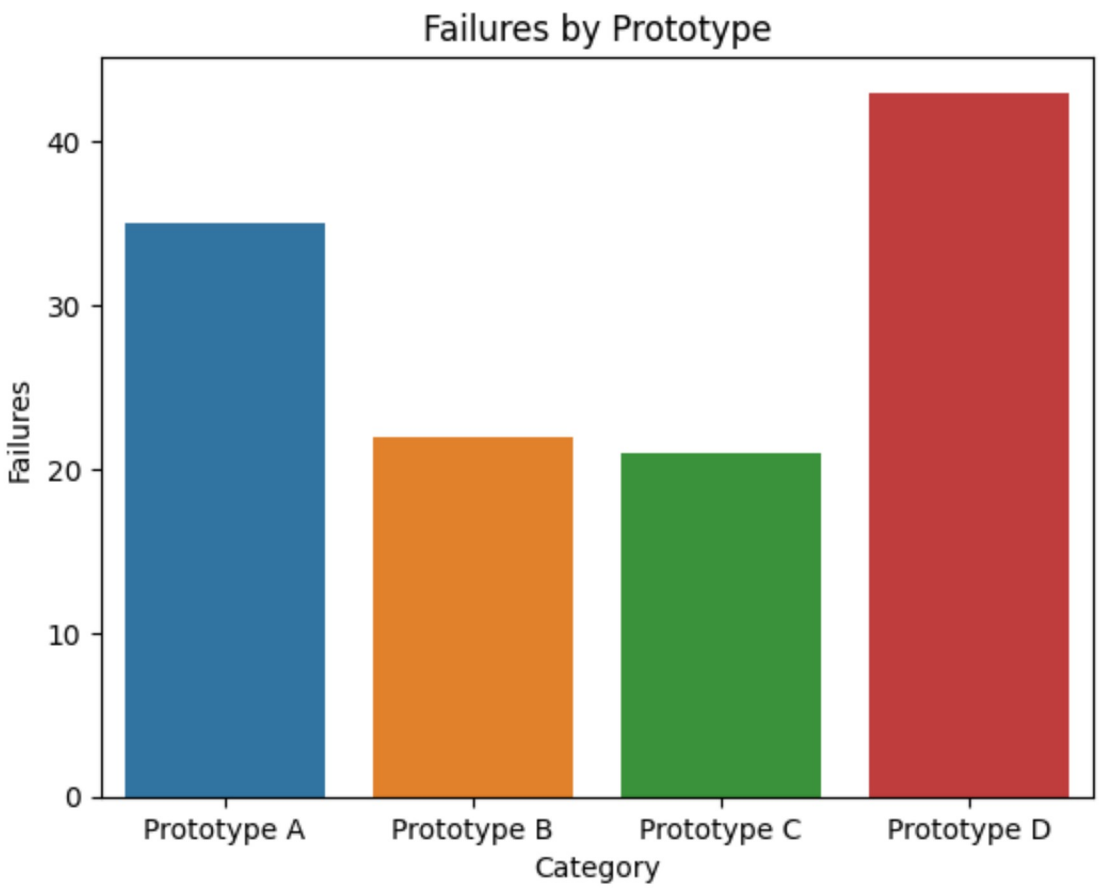
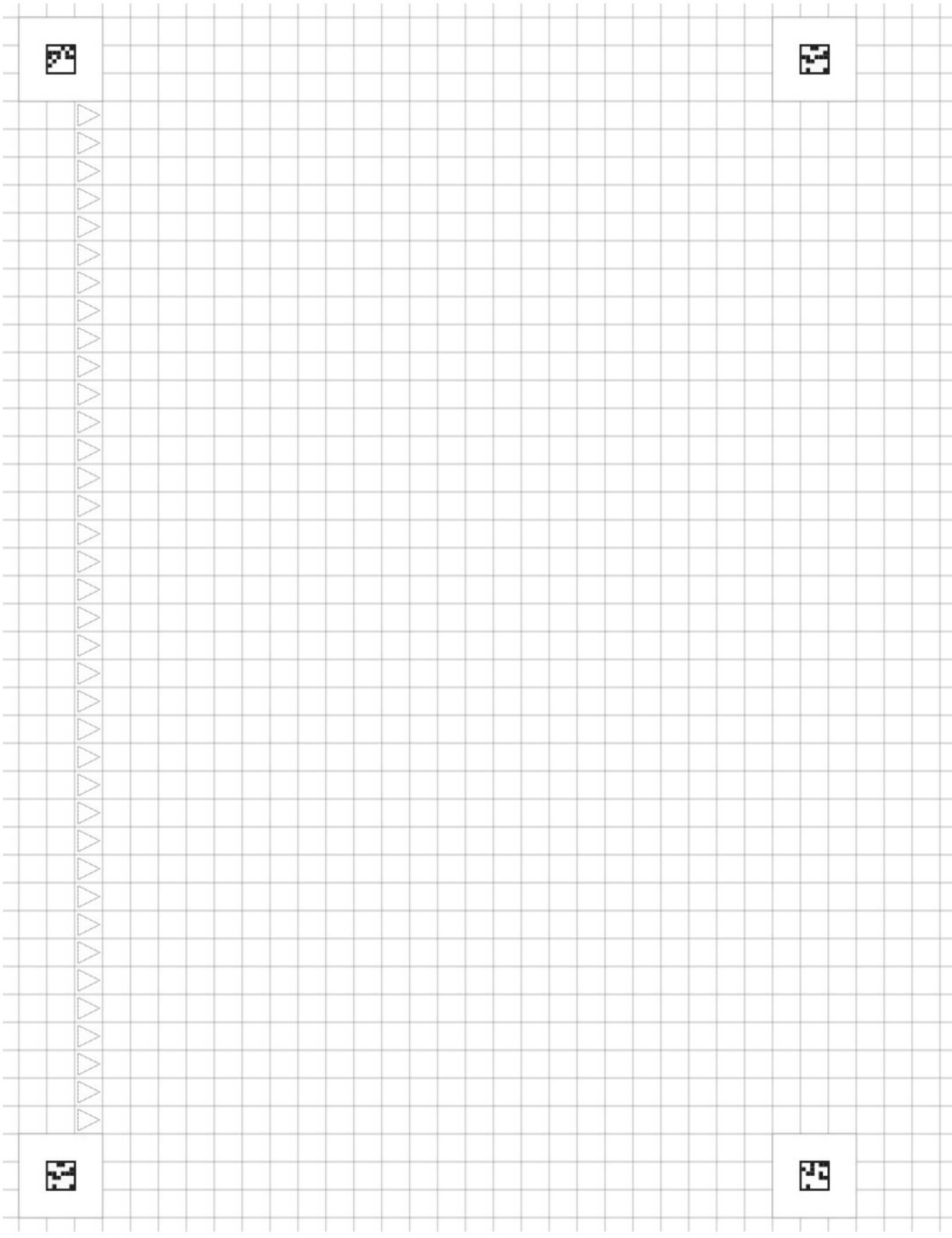


Image Normalization

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

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

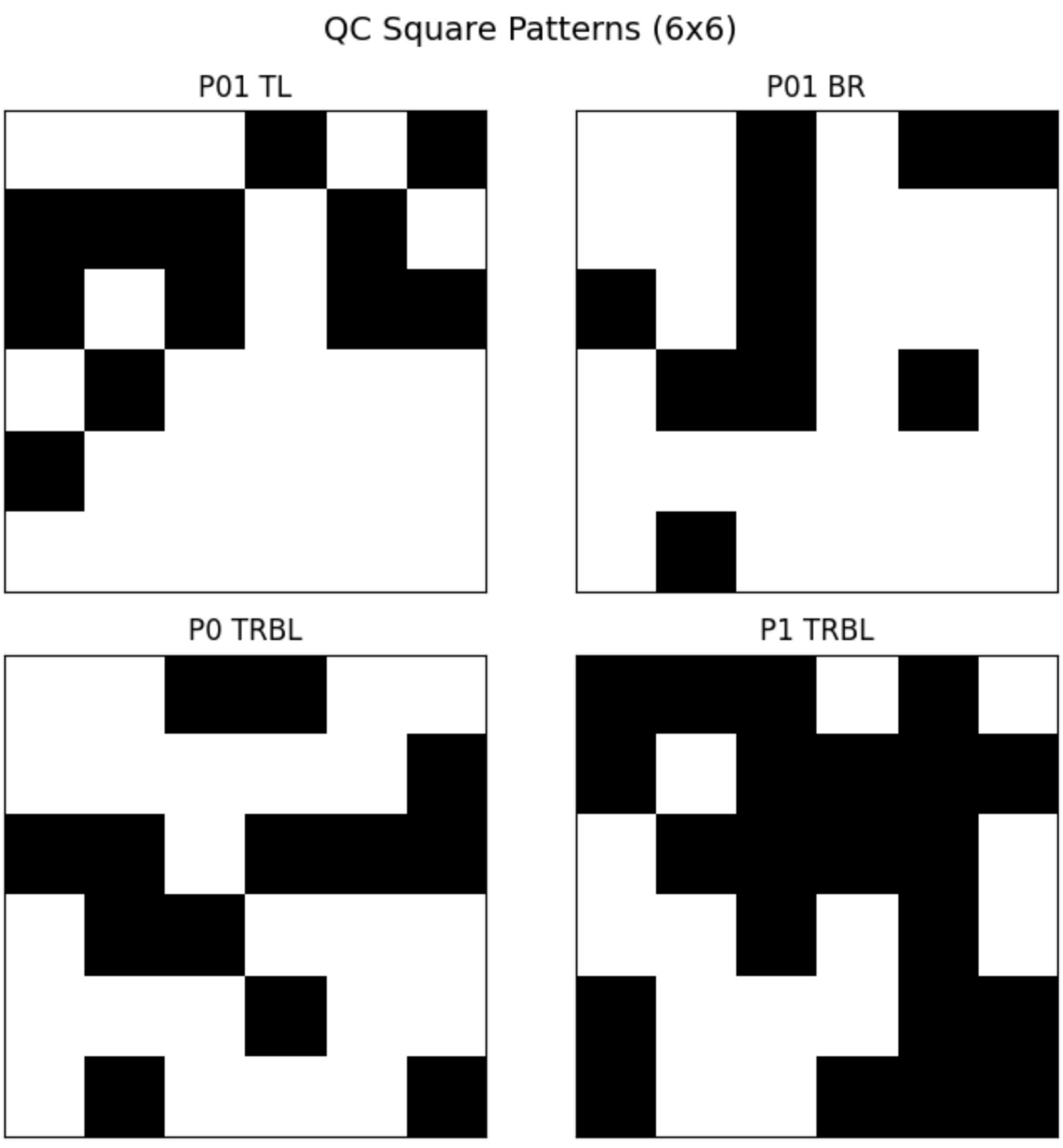
Matrix Representations of QC Squares

In [69]:
```
# 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 [70]:
```
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.

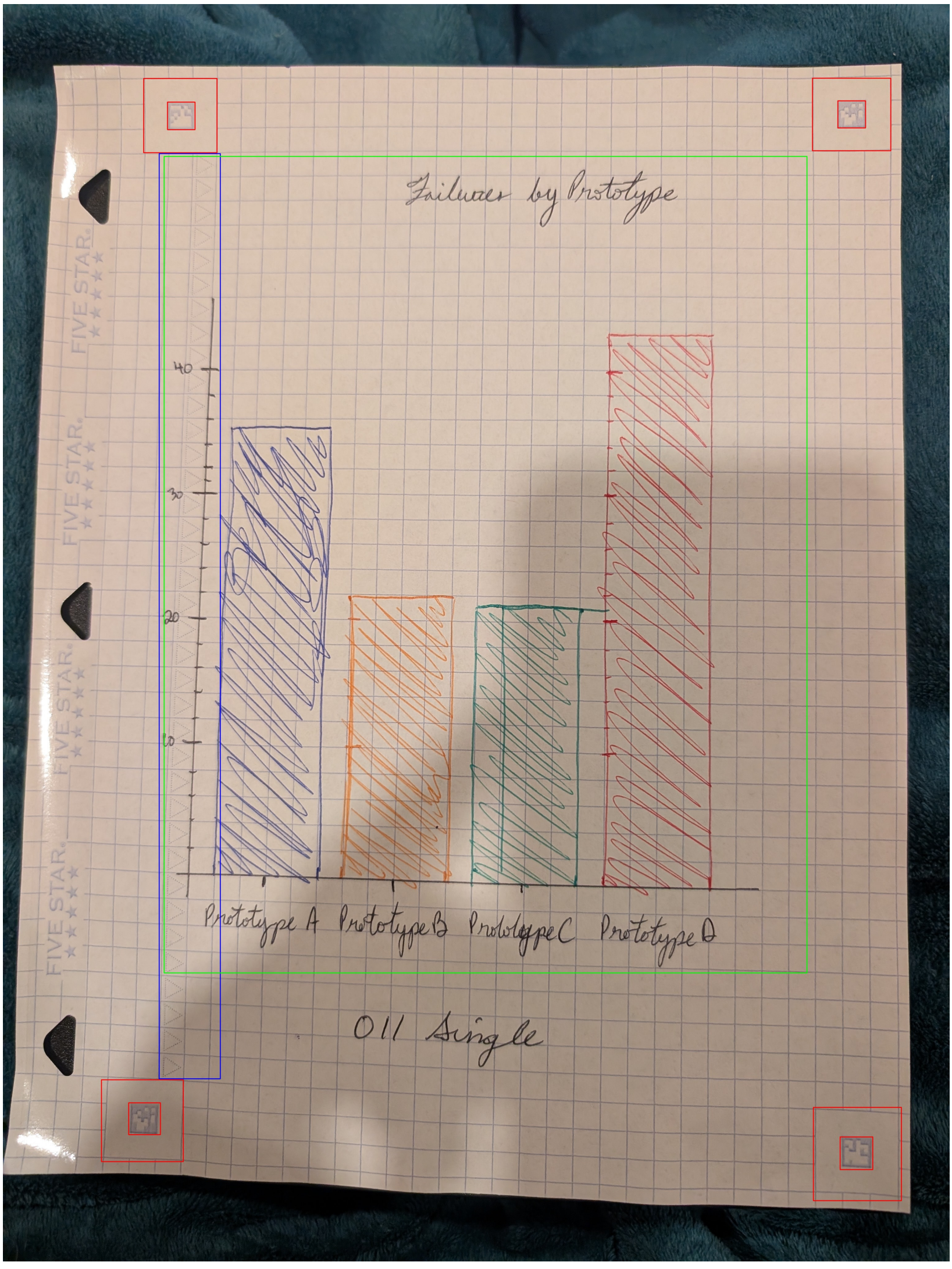
011 Single

Identifying Page Features

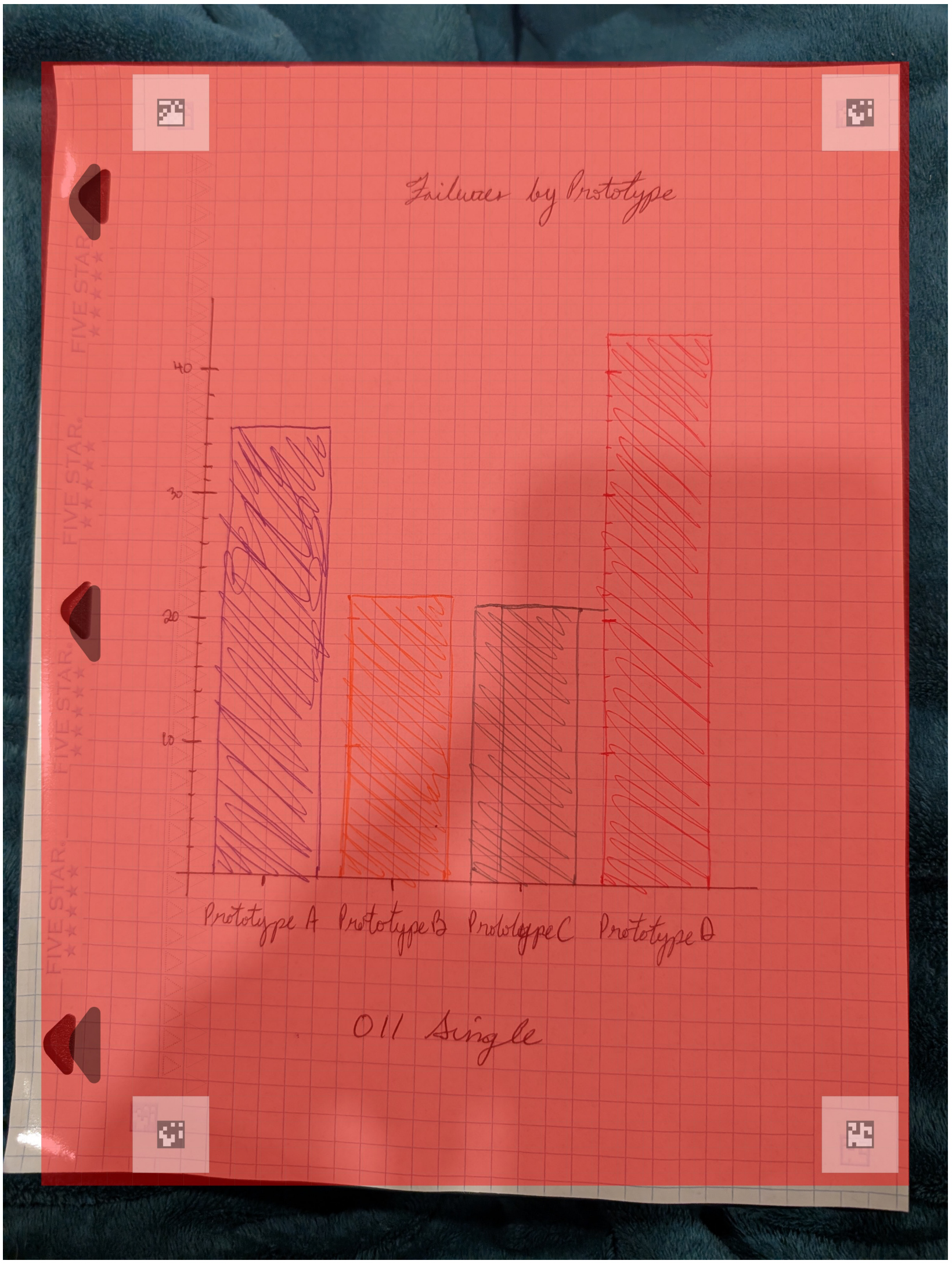
Red: QC squares

Blue: Triangles

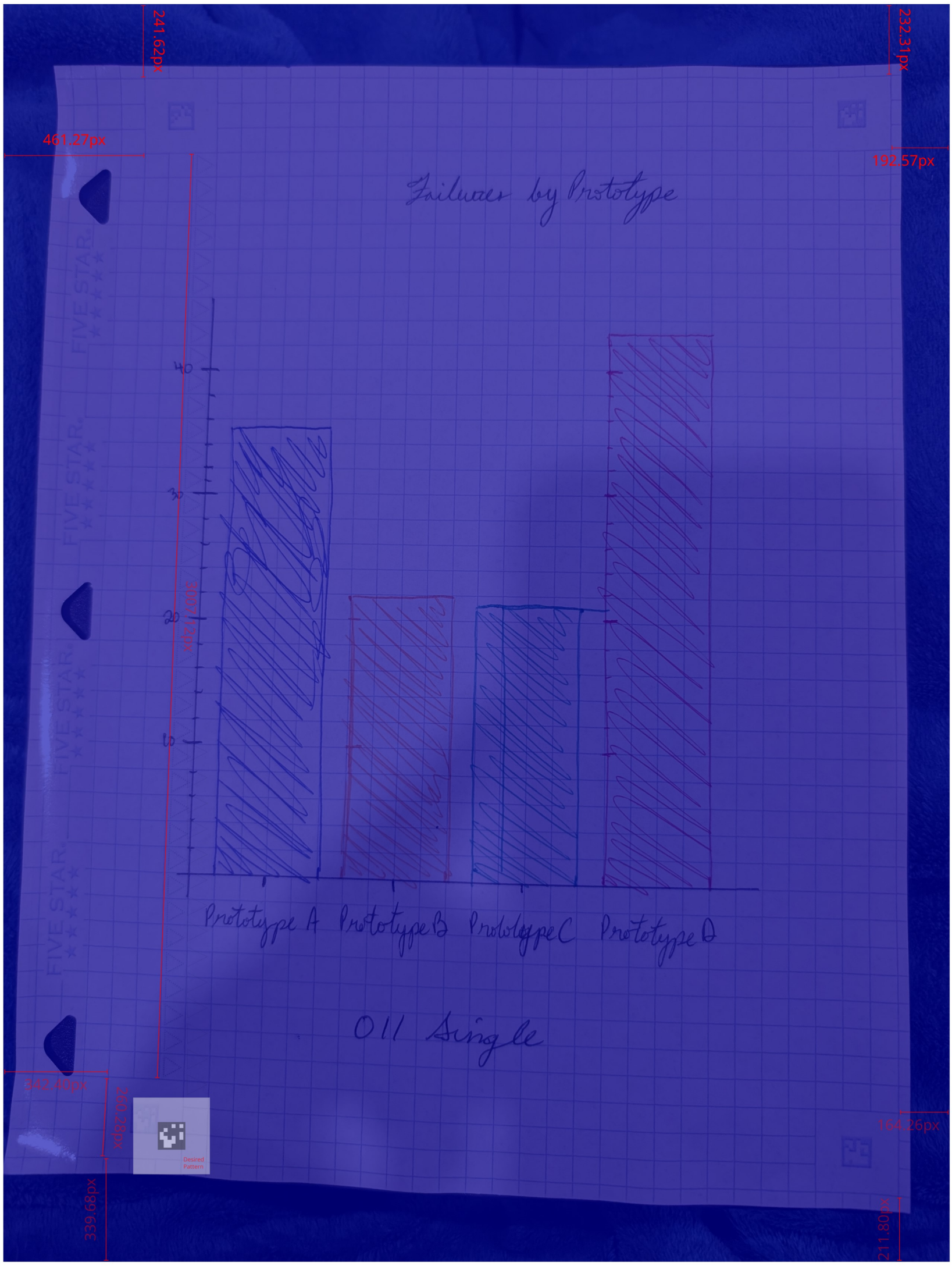
Green: Chart Area



Desired Features



Page Dimensions



In [71]:
```
## Test Image
import PIL
import os
import cv2
import numpy as np
```

In [81]:
```
def detect_qc_square(image, corner='auto', search_region_size=0.2, min_square_area=100):
    """
    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)), w, h)
    }

    # 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% of region

        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 else 0

                    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=True)
        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_margin=0.1)

            # 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())} or 'auto'")
        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 [82]:

```
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 [83]:
```
# 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%
```

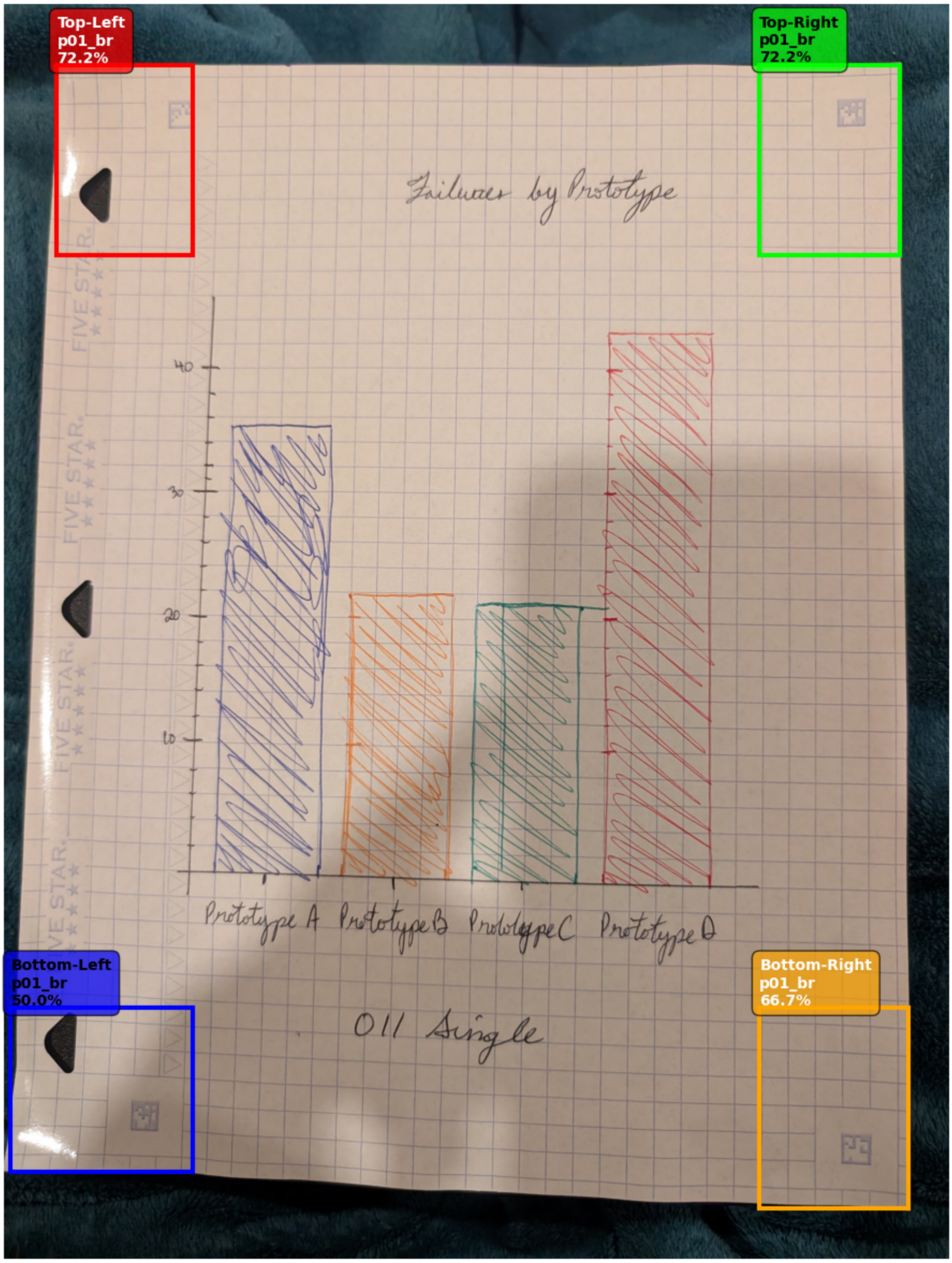
Visualisation

In [88]:
```
import matplotlib.pyplot as plt
import matplotlib.patches as patches

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 [90]:
```
# 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 (recommended for notebooks)
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 [45]:
```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 missing corners.")
            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 missing corners.")
            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_diagonal_fallback=True):
    """
    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_fallback=allow_diagonal_fallback)
    if corners is None:
        raise ValueError("Could not extract all 4 corner points from QC squares")

    # Source points (from detected corners)
    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 (rectified rectangle)
    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 (rectified rectangle)
    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 squares.
    """
    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 [91]:

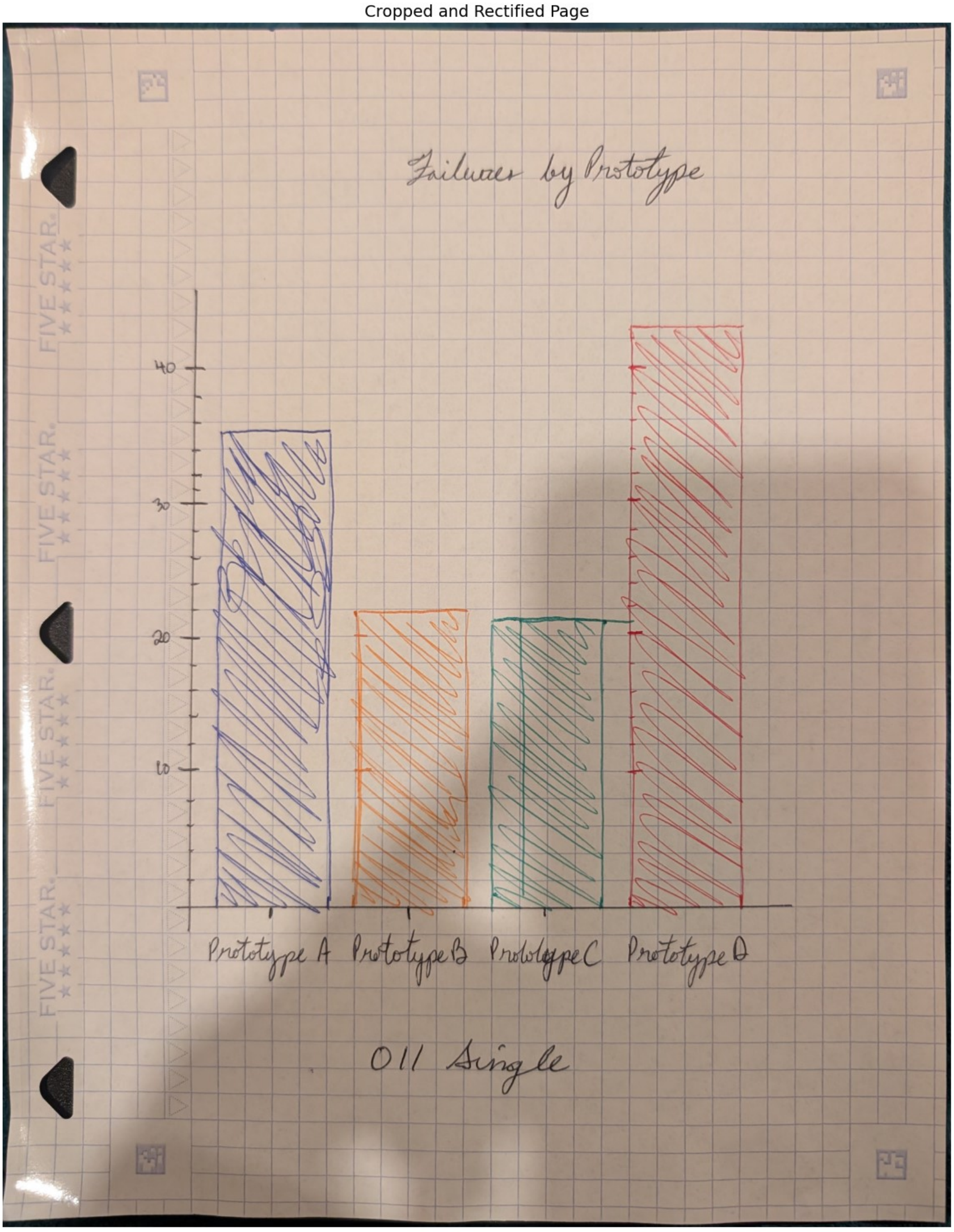
```
def visualize_cropped_result(cropped_image, title="Cropped and Rectified Page", figsize=(12, 15)):
    """
    Visualize the cropped/rectified image result.

    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 [93]:

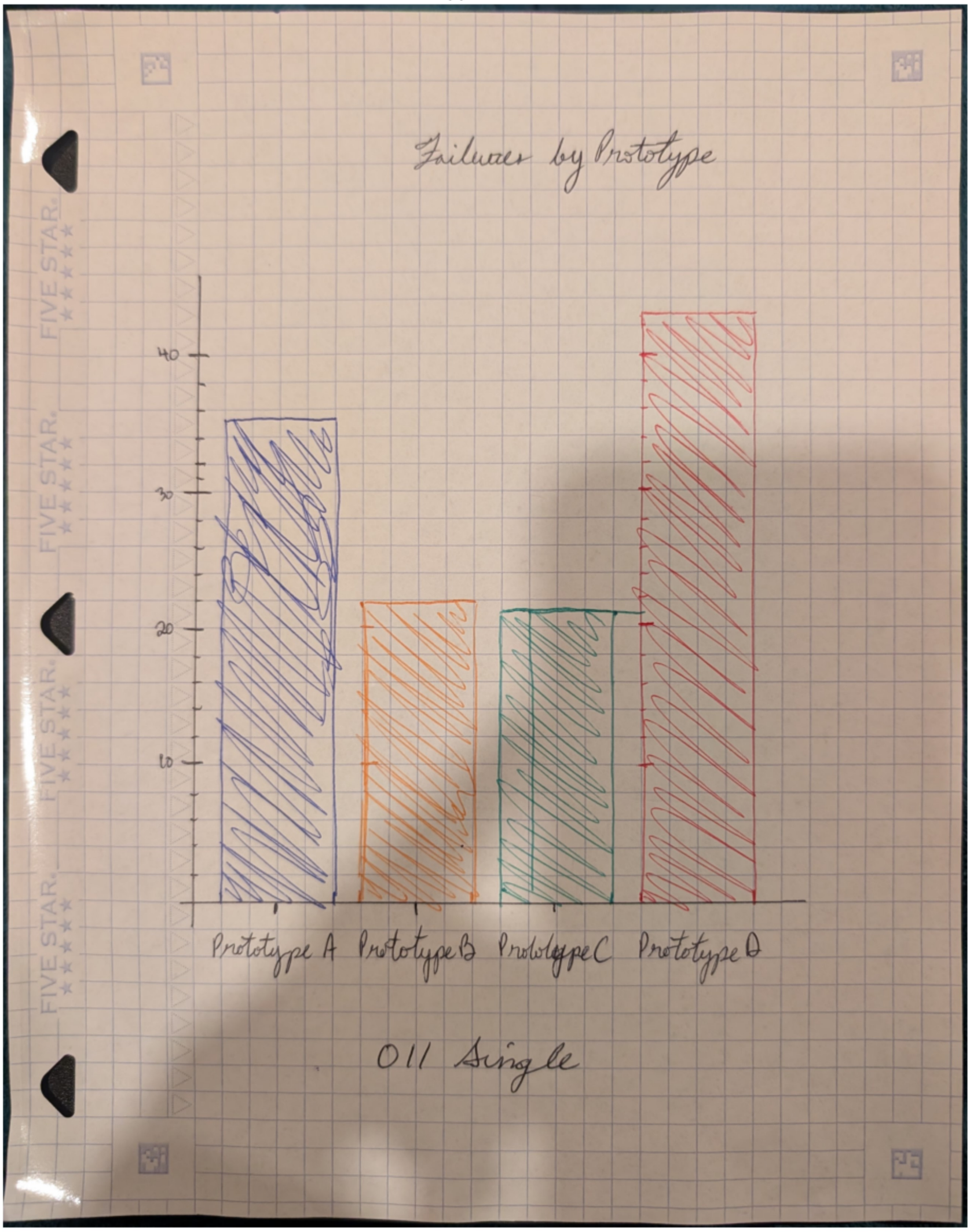
```
# Load and detect
image_path = 'hand_drawn_notes/bc_011_single-004.jpg'
image = cv2.imread(image_path)
rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
results = detect_qc_square(rgb_image, corner='auto')

# 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 [48]:
```
NotImplementedError
```

Out[48]:
```
NotImplementedError
```

Gallery of Processed Images

In [49]:
```
# 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

  image_paths = [

    ...

  ]  # your 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['corner'], 'white')
          rect = patches.Rectangle((x, y), w, h, edgecolor=color, facecolor='none', lw=2)
          ax.add_patch(rect)
          ax.text(x, y - 5, f"{result['corner']} {result.get('confidence',0):.0%}", color='white',
                  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)
```

```
  Cell In[49], line 3
    import matplotlib.pyplot as plt
    ^
IndentationError: unexpected indent
```