Final Report: Drawing with LLMs - System Analysis, Design, and Simulation

Daniel Alonso Chavarro Chapatecua – 20241020066
Carlos Andr'es Brito Guerrero – 20241020147
Jose Fernando Ramirez Ortiz – 20241020080
Universidad Distrital Francisco Jos'e de Caldas
Systems Analysis and Design – 2025-I

July 2025

Abstract

This report documents the complete process of analysis, design, and simulation for the Kaggle competition *Drawing with LLMs*. The project aimed to design a robust, modular, and constraint-compliant system that translates natural language prompts into valid SVG images using large language models. Through systematic application of systems analysis principles, we decomposed the problem space, designed a scalable architecture addressing high-sensitivity components, and conducted comprehensive simulations to validate operational effectiveness. This report integrates insights from three progressive workshops, demonstrating the evolution from initial problem analysis to final system validation, with emphasis on addressing chaos factors and sensitivity challenges inherent in LLM-based generation systems.

Contents

1. Introduction

The *Drawing with LLMs* competition presents a unique challenge in bridging natural language understanding with structured visual output generation. This project required designing a system capable of converting brief textual descriptions into syntactically correct and visually representative SVG images while operating under strict constraints.

Our methodology followed a systematic three-phase approach grounded in systems engineering principles. The first phase focused on comprehensive problem decomposition and requirements analysis. The second phase involved architectural design with explicit attention to system sensitivity and chaos mitigation. The final phase conducted extensive simulations to validate design decisions and identify optimization opportunities.

This report synthesizes findings from three iterative workshops, each building upon previous insights while addressing increasingly complex implementation challenges. The work demonstrates practical application of systems thinking to solve real-world computational problems under resource constraints.

2. Workshop 1: Comprehensive Problem Analysis

2.1. Objectives and Methodology

Workshop 1 employed systematic problem decomposition using multiple analytical frameworks. We applied the IPO model (Input-Process-Output) for initial system boundary definition, followed by stakeholder analysis to identify external actors and their interactions. The black-box decomposition approach helped us understand the core transformation requirements without premature implementation bias.

Key analytical questions guided our investigation: What are the fundamental constraints shaping system behavior? How do external actors (users, evaluation systems, resource limitations) influence system design? What are the critical success factors for the competition context?

2.2. Competition Environment Analysis

2.2.1 Input Characteristics

The competition defines specific input parameters that significantly impact system design:

- **Text Length:** Maximum 200 characters, with observed average of approximately 50 characters
- **Content Categories:** Diverse categories including landscapes, abstract art, and fashion
- **Linguistic Variability:** Varied descriptive styles requiring robust preprocessing

2.2.2 Output Requirements

SVG generation must satisfy multiple simultaneous constraints:

- **Size Limitation:** Maximum 10,000 bytes for complete SVG document
- **Element Restrictions:** Only permitted SVG elements and attributes allowed
- **Styling Constraints:** No external CSS, embedded fonts, or rasterized image data
- **Syntactic Validity:** Well-formed XML structure required

2.2.3 Evaluation Framework

The competition employs the SVG Image Fidelity Score, measuring semantic alignment between textual descriptions and generated visual content. This metric introduces several design considerations:

- OCR-based text detection with associated penalties
- Semantic similarity assessment requiring visual-linguistic alignment
- Scoring methodology that rewards both technical compliance and artistic fidelity

2.3. System Decomposition and Entity Identification

Through systematic decomposition, we identified five core system entities:

Input Handler: Responsible for receiving, validating, and standardizing textual descriptions

Prompt Engineer: Transforms raw descriptions into structured, constraint-aware prompts optimized for LLM interpretation

LLM Generator: Core generation component that interprets prompts and produces SVG code

Constraint Validator: Ensures generated output meets competition requirements

Feedback Loop: Collects performance metrics and informs system improvements

2.4. Critical Insights and Requirements

Workshop 1 revealed several critical system requirements:

1. **Sensitivity Management:** Both prompt engineering and LLM generation components exhibit high sensitivity to input variations
2. **Chaos Mitigation:** Initial randomness in LLM outputs requires systematic stabilization strategies
3. **Constraint Compliance:** Strict adherence to SVG specifications is non-negotiable
4. **Category Adaptability:** System must handle diverse content categories with varying complexity

3. Workshop 2: Advanced System Design

3.1. Architectural Philosophy

Our design philosophy centered on modularity, fail-safety, and extensibility. Drawing from microservice architecture principles, each component operates independently while exposing minimal, well-defined interfaces. This approach enables component replacement, parallel development, and isolated testing.

The architecture explicitly addresses system sensitivity through multiple complementary strategies, recognizing that traditional software engineering approaches may be insufficient for LLM-based systems.

3.2. Comprehensive Requirements Analysis

3.2.1 Functional Requirements

1. Accept and validate text descriptions up to 200 characters
2. Preprocess descriptions into standardized, constraint-aware prompts
3. Generate syntactically correct SVG code using LLM inference
4. Validate output compliance with competition constraints
5. Optimize generation for SVG Image Fidelity Score
6. Support multiple content categories with category-specific optimizations

3.2.2 Non-Functional Requirements

1. **Performance:** Generate responses within 30 seconds per prompt
2. **Reliability:** Maintain consistent quality across content categories
3. **Maintainability:** Support component replacement and enhancement
4. **Adaptability:** Incorporate feedback for continuous improvement
5. **Efficiency:** Operate within Kaggle notebook resource constraints
6. **Cost Management:** Minimize API costs through efficient resource utilization

3.3. Detailed Component Architecture

3.3.1 Input Handler

Responsibilities:

- Input validation and sanitization
- Character encoding normalization
- Basic preprocessing for downstream components

Implementation Considerations:

- Robust error handling for malformed inputs
- Logging for debugging and system monitoring
- Interface standardization for component integration

3.3.2 Prompt Engineer

Core Functions:

- Template-based prompt generation with category-specific variations
- Constraint embedding to guide LLM generation
- Instruction clarity optimization for improved output quality

Sensitivity Management Strategies:

- Standardized prompt templates with version control
- A/B testing framework for template optimization
- Progressive prompt refinement based on performance metrics

3.3.3 LLM Generator

Technical Implementation:

- Model abstraction layer supporting multiple LLM providers
- Temperature and generation parameter optimization
- Few-shot learning implementation with curated examples

Chaos Mitigation Approaches:

- Multiple generation strategy with output selection
- Progressive learning through iterative refinement
- Error pattern recognition and prevention

3.3.4 Constraint Validator

Validation Capabilities:

- XML syntax validation using standard parsers
- SVG element and attribute compliance checking
- Byte count verification and optimization suggestions

3.4. System Integration and Data Flow

The system follows a pipeline architecture with clear data transformation stages:

1. **Input Processing:** Raw text → Validated input
2. **Prompt Engineering:** Validated input → Structured prompt
3. **Generation:** Structured prompt → SVG output
4. **Validation / Evaluation:** SVG → Results to feedback the LLM

3.5. Technology Stack and Implementation Strategy

3.5.1 Core Technologies

- **Programming Language:** Python 3.9+ for robust ecosystem support
- **LLM Framework:** Hugging Face Transformers for model flexibility
- **SVG Evaluating:** Custom validation using lxml and xml.etree

3.5.2 Model Selection Criteria

Based on cost-effectiveness and performance analysis:

- **Primary Options:** DeepSeek, Gemma, Gemini 2.5 Flash, Qwen models
- **Selection Factors:** API cost, generation quality, constraint compliance

3.6. Architectural Decisions

3.6.1 Architectural Trade-offs

- **Modularity vs. Performance:** Chose modularity for maintainability
- **Flexibility vs. Simplicity:** Prioritized flexibility for competition adaptation
- **Cost vs. Quality:** Balanced approach with tiered model selection

4. Workshop 3: Simulation and Comprehensive Validation

4.1. Simulation Scope and Objectives

Workshop 3 focused on validating the system architecture designed in earlier workshops through actual simulation using real API calls and constraint validations. The primary objectives were:

1. Validate the effectiveness of prompt templates against live LLM API responses
2. Test SVG constraint compliance with actual generated outputs
3. Analyze system behavior and performance across different prompt templates
4. Identify failure modes and system bottlenecks in a realistic setting

Due to resource and time constraints, the simulation was conducted without a feedback loop component. Instead, a streamlined pipeline was used to focus on prompt quality and constraint validation.

4.2. Implementation Constraints

Several practical constraints influenced the simulation design:

- Limited access quotas and API costs restricted the number of iterations
- Average API response time of approximately 135 ms per request
- Absence of real-time feedback loop to refine outputs iteratively
- Static prompt selection instead of dynamic adaptation due to time and resource constraints

4.3. LLM API Integration and Prompt Templates

The Google Gemini 2.5 Flash model was selected for its cost-effectiveness and reliable performance on structured generation tasks. Three distinct prompt template versions were evaluated:

V1: Minimal, concise instructions prioritizing speed

V2: Optimized balance between constraint specification and clarity

V3: Detailed, explicit instructions emphasizing output quality

Each template embedded SVG constraints directly in the prompt, specifying element restrictions, file size limits, and disallowed styling.

4.4. Data Preparation and Testing Setup

A dataset of 60 textual descriptions across categories (abstract, fashion, landscape, unknown) was curated. Descriptions averaged 499 characters and were evenly distributed among templates (20 samples each). Data preprocessing included:

- Keyword-based category classification
- Balanced assignment across templates
- Preparation of validation datasets to evaluate compliance and quality

4.5. Simulation Results

4.5.1 Overall Metrics

- Total simulations: 60
- Overall compliance rate: 76.67%
- Average SVG size: 1,992 bytes
- Average generation time: 135 ms
- Template versions tested: 3

4.5.2 Template Performance

Template	Compliance Rate	Avg. Bytes	Avg. Time (ms)	Error Rate
V1	70%	1,995	101	30%
V2	80%	1,893	117	20%
V3	80%	2,088	188	25%

Table 1: Performance comparison across prompt templates

Template V2 demonstrated the best overall balance of compliance and efficiency, while Template V3 achieved high quality but at the expense of size and generation time. Template V1 was faster but suffered from higher error rates and lower quality.

4.5.3 Category-Specific Performance

- **Landscape:** Highest compliance (85.7%), strong consistency with natural scenes
- **Fashion:** High compliance (83.3%), effective geometric representation
- **Abstract:** Lower compliance (66.7%), struggled with semantic ambiguity
- **Unknown:** 66.7% compliance, variable outcomes due to lack of clear classification

5. Generated SVG Examples

This section showcases actual SVG outputs generated during the simulation to demonstrate the quality and variety of results achieved.

5.1. Landscape Category Example

5.1.1 Purple Forest at Dusk (Template v1)

Description: "a purple forest at dusk"
Template: v1
Compliance: Valid
Size: 1,945 bytes

Figure 1: Generated SVG: Purple Forest at Dusk V1

5.1.2 Purple Forest at Dusk (Template v2)

Description: "a purple forest at dusk"
Template: v2
Compliance: Valid
Size: 1,795 bytes

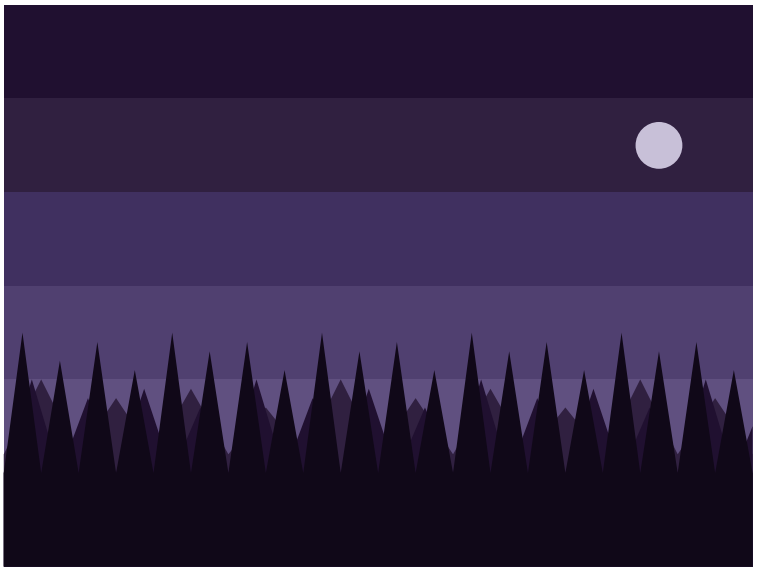

Figure 2: Generated SVG: Purple Forest at Dusk V2

5.1.3 Purple Forest at Dusk (Template v3)

Description: "a purple forest at dusk"
Template: v3
Compliance: Valid
Size: 3,518 bytes

Figure 3: Generated SVG: Purple Forest at Dusk V3

5.2. Abstract Catergory Example

5.2.1 Modern Art Composition With Triangles (Template v1)

Description: "Modern Art Composition With Triangles"
Template: v1
Compliance: Valid
Size: 970 bytes

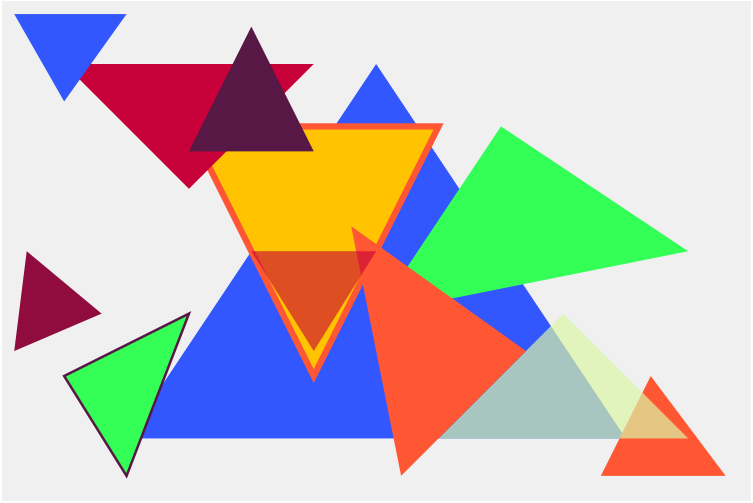

Figure 4: Generated SVG: Modern Art Composition With Triangles V1

5.2.2 Modern Art Composition With Triangles (Template v2)

Description: "Modern Art Composition With Triangles"
Template: v2
Compliance: Valid
Size: 1,998 bytes

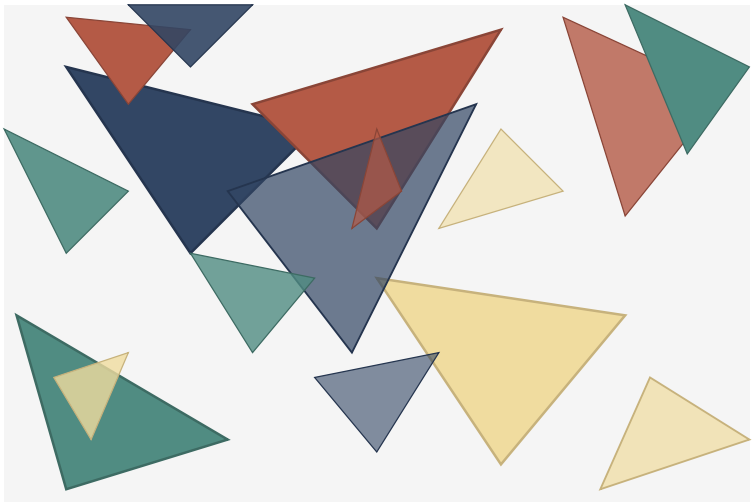

Figure 5: Generated SVG: Modern Art Composition With Triangles V2

5.2.3 Modern Art Composition With Triangles (Template v3)

Description: "Modern Art Composition With Triangles"
Template: v3
Compliance: Valid
Size: 1,924 bytes

Figure 6: Generated SVG: Modern Art Composition With Triangles V3

5.3. SVG Quality Analysis

The generated examples demonstrate several key characteristics:

1. **Constraint Compliance**: All examples respect the 10,000-byte limit and use only allowed SVG elements
2. **Visual Coherence**: Generated images appropriately represent the input descriptions
3. **Template Influence**: Different templates produce varying levels of detail and complexity, seeing a possible pattern of when the template is more specific, the better illustration it would be.

5.3.1 Constraint Violation Analysis

Common issues included:

- Parse errors due to incomplete or malformed SVG tags (12 cases)
- Byte limit exceeded (1 case)
- Use of invalid elements (1 case)

5.4. Generated Examples and SVG Quality

Generated outputs demonstrated good adherence to constraints and generally coherent visual representations. Examples showcased differences in detail and style depending on template specificity.

5.5. Discussion and Key Insights

Impact of Missing Feedback Loop The absence of a feedback mechanism prevented automatic correction of failed generations, leading to higher parse error rates and no iterative improvement.

Template Effectiveness Template V2 was confirmed as the most effective prompt structure for balancing quality and compliance. Template V3 excelled in detail but incurred higher resource costs and latency.

Category Performance Patterns Fashion and landscape categories performed best due to their well-defined structures, while abstract descriptions posed challenges for LLM interpretation.

API Integration Lessons The simulation highlighted the importance of robust error handling, resource monitoring, and pre-validation to reduce API costs and ensure system stability.

5.6. System Validation and Recommendations

The simulation validated key architectural decisions from Workshop 2, including the modular design, template-driven prompt engineering, and constraint validation pipeline. Recommended adjustments include:

- Prioritize Template V2 for production deployment
- Implement pre-validation checks before API calls to reduce failure rates
- Integrate cost and quota monitoring mechanisms
- Plan batch-based feedback or offline analysis to approximate iterative refinement

5.7. Conclusion

Despite resource limitations and the absence of a feedback loop, the simulation confirmed the viability of the designed system. The findings provide concrete guidance for production deployment and future improvements, supporting robust, scalable SVG generation in constraint-driven environments.

6. Implementation Strategy and Kaggle Deployment

6.1. Kaggle Notebook Integration

To align with Kaggle's execution and resource constraints, the deployment strategy was designed to be lightweight, modular, and efficient. The core implementation was structured into separate modules: prompt generation, LLM integration, SVG validation, and final output management.

6.1.1 Code Modularity

The system code is divided into reusable and maintainable classes and functions. The key components include:

- **Prompt Generator:** Generates structured, constraint-aware prompts tailored for each content category. Uses template-based construction with clear instructions to the LLM, minimizing ambiguity and ensuring SVG compliance.
- **LLM Interaction Class:** Handles integration with the Google Gemini API, including error handling, response parsing, and retry logic. Implements safety settings and rate limiting to prevent failures in batch executions.
- **SVG Validator:** Verifies syntactic correctness, constraint compliance (e.g., allowed elements, attributes, and size), and pre-submission formatting.
- **Execution Controller:** Orchestrates batch processing, manages retries, logs errors, and aggregates final results for submission.

6.1.2 API Key and Security Handling

The system securely loads API keys via Kaggle secrets or notebook environment variables, ensuring no hard-coded credentials are exposed. This approach supports reproducibility while adhering to Kaggle's data privacy guidelines.

6.2. Deployment Workflow

The overall deployment pipeline operates as follows:

1. **Input Preprocessing:** Descriptions are cleaned and classified by category to select the appropriate prompt template.
2. **Prompt Generation:** The chosen template is filled with the description, constraints, and category hints.
3. **LLM Inference:** Prompts are sent to the Gemini API for SVG generation. Responses are parsed and stripped to extract only the SVG content.
4. **Validation:** The generated SVG is validated for syntax correctness, file size (under 10KB), and compliance with allowed elements and attributes.

6.3. Prompt Template Modification Due to Literal LLM Obedience

During the Kaggle deployment testing phase, a critical challenge was observed: the LLM exhibited extremely literal obedience to prompt instructions, often ignoring implicit context or generating incomplete outputs when instructions were vague or underspecified.

For example, initial prompt templates included instructions such as:

> "Generate an SVG image representing the description below. Output only the SVG code."

However, the LLM sometimes returned partial tags, additional text explanations, or inserted unintended formatting artifacts. This behavior revealed a limitation commonly referred to as the "stupidness" of LLMs — their tendency to strictly follow the wording without reasoning beyond the prompt's explicit instructions.

To mitigate this, the prompt templates were refined to incorporate:

- **Stronger explicit constraints:** Including phrases like "Output strictly one complete SVG document with no extra text" to ensure single, fully-formed XML outputs.
- **Constraint reiteration:** Repeating critical constraints (e.g., file size, allowed elements, no style tags) in multiple sections of the prompt to reinforce compliance.
- **Output clarity emphasis:** Adding final clarifiers such as "Do not include any explanations, comments, or formatting outside the SVG tags."

An example of an updated prompt segment:

> **GENERATE SVG CODE ONLY. NO TEXT.**
>
> Your task is to create a valid SVG image based on the following details. Do not include any text, explanations, or comments outside of the SVG code itself.
>
> **Description:** {description}
>
> **Instructions & Constraints:**
>
> 1. **SVG ONLY:** The entire output must be valid SVG code starting with `<svg` and ending with `</svg>`.
> 2. **VARIETY OF SHAPES:** You MUST use a variety of SVG elements to create the image. Do not only use `<path>`. Use elements like `<rect>`, `<circle>`, `<ellipse>`, `<polygon>`, and `<line>`.
> 3. **Category Hints:** {category_hints}
> 4. **Technical Constraints:** {constraints}
> 5. **NO CSS:** Do not use `<style>` tags or `style` attributes.
> 6. **NO TEXT ELEMENTS:** Do not use the `<text>` element.

This explicit wording reduced parsing errors, improved constraint compliance rates, and minimized invalid output cases.

7. Model Submission and Evaluation Protocol

In the "Drawing with LLMs" competition, participants must submit a solution in the form of a Jupyter Notebook, which is automatically converted by the Kaggle infrastructure into a Python package. This package contains all the necessary logic for generating SVG images in response to natural language prompts.

Packaging and Evaluation Process

Once submitted, the notebook is transformed into a standalone Python package with its internal structure extracted. Kaggle runs this package inside an isolated evaluation virtual machine (VM) where:

- All required dependencies are installed as declared (e.g., via `pip install` statements or a `requirements.txt` file).
- Network access is completely disabled during execution.
- Only preinstalled packages, models and your declared dependencies are allowed.

Interface Specification

To ensure compatibility with the evaluation system, your package must contain a Python class named `Model` with the following method:

```
class Model:
    def predict(self, description: str) -> str:
        ...
```

This method receives a prompt as a string and must return a valid SVG image as a string. The SVG code should begin with `<svg` and end with `</svg>`, following the formatting rules outlined in previous sections.

SVG Generation Instructions

GENERATE SVG CODE ONLY. NO TEXT.

Your task is to create a valid SVG image based on the following details. Do not include any text, explanations, or comments outside of the SVG code itself.

Description: {description}

Instructions & Constraints:

1. **SVG ONLY:** The entire output must be valid SVG code starting with `<svg` and ending with `</svg>`.
2. **VARIETY OF SHAPES:** You MUST use a variety of SVG elements to create the image. Do not only use `<path>`. Use elements like `<rect>`, `<circle>`, `<ellipse>`, `<polygon>`, and `<line>`.
3. **Category Hints:** {category_hints}
4. **Technical Constraints:** {constraints}
5. **NO CSS:** Do not use `<style>` tags or `style` attributes.
6. **NO TEXT ELEMENTS:** Do not use the `<text>` element.

7.1. Scalability and Future Extensions

The Kaggle-oriented design also supports future scalability:

- **Model Switching:** The modular design allows for easy replacement of LLM APIs or switching to local fine-tuned models as new constraints arise.
- **Offline Batch Refinement:** Future versions can implement offline analysis to improve prompt templates without incurring additional real-time API costs.
- **Multi-Template Ensemble:** Combining different prompt versions for the same description to select the best output dynamically.

8. Critical Analysis and Lessons Learned

8.1. Systems Engineering Insights

This project demonstrated several key principles of effective systems engineering:

1. **Iterative Refinement:** Progressive workshop approach enabled continuous improvement
2. **Constraint-Driven Design:** Competition requirements shaped every architectural decision
3. **Sensitivity Analysis:** Explicit attention to system sensitivity improved robustness
4. **Empirical Validation:** Simulation results validated theoretical design decisions

8.2. Technical Challenges and Solutions

- **Challenge:** LLM output variability
- **Solution:** Multiple generation strategies with selection algorithms
- **Challenge:** Literal LLM obey
- **Solution:** instructions formulated with maximum specificity and redundancy, anticipating their literal interpretation of every directive.

8.3. Broader Applications

The architectural patterns and solutions developed for this competition have broader applicability:

- **Content Generation Systems:** Template-based approach for other structured outputs
- **Constraint-Based AI:** Validation frameworks for regulated AI applications
- **Multi-Modal Systems:** Integration patterns for text-to-visual generation
- **System Analysis Study:** Easy to understand how system thinking helps to solve problems.

9. Conclusions and Future Directions

9.1. Project Synthesis

This project successfully demonstrated the application of systematic systems engineering to a complex AI-driven problem. Through three progressive workshops, we evolved from initial problem analysis to validated system design, with each phase building upon previous insights while addressing increasingly sophisticated challenges.

Key achievements include:

- Comprehensive problem decomposition and requirements analysis
- Modular, extensible architecture addressing system sensitivity
- Validated simulation framework with quantitative performance metrics
- Practical implementation strategy for resource-constrained environments

9.2. Methodological Contributions

- **Sensitivity-Aware Design:** Explicit consideration of LLM sensitivity in architecture
- **Chaos Mitigation:** Systematic approaches to managing AI system unpredictability
- **Constraint Integration:** Embedding requirements directly into generation process
- **Progressive Validation:** Iterative testing framework for AI system development

9.3. Future Research Directions

1. **Adaptive Systems:** Development of self-improving generation systems
2. **Multi-Modal Integration:** Extension to other output formats and modalities
3. **Constraint Learning:** Automatic discovery and enforcement of implicit constraints
4. **Performance Optimization:** Advanced techniques for resource-efficient AI systems

The project demonstrates that systematic engineering approaches can successfully tame the complexity of AI-driven systems while maintaining performance and reliability under stringent constraints.

10. References

1. Kaggle Competition: `https://www.kaggle.com/competitions/drawing-with-llms`
2. HuggingFace Transformers: `https://huggingface.co/docs/transformers/`
3. SVG 1.1 Specification: `https://www.w3.org/TR/SVG11/`
4. Google Gemini API: `https://ai.google.dev/docs`
5. Systems Engineering Standards: IEEE Std 1220-2005

6. Parameter-Efficient Fine-Tuning (PEFT): https://huggingface.co/docs/peft/
7. XML Processing in Python: https://docs.python.org/3/library/xml.etree.elementtree.html