# Design Specification Document (SDS)

for

# Chart Understanding LLM System

with Chart2Table Pipeline & Model Comparison

Version 1.0

December 11, 2025

Supervisors:
**Dr. Ahmed Tawfik**
**Dr. Cherif Salama**
**Dr. Hossam Sharara**

Department of Computer Science and Engineering
The American University in Cairo

# Contents

# 1 Introduction

## 1.1 Purpose

This Design Specification Document (SDS) provides a comprehensive technical blueprint for the Chart Understanding System with Multi-Strategy Prompting and Chain-of-Models Architecture. The document serves as the authoritative reference for the system's architecture, design decisions, and implementation specifications.

The primary objectives of this document are:

- Define the complete system architecture including hardware, software, and communication components
- Detail the design of the multi-model pipeline integrating Qwen2.5-VL-7B-Instruct with Chart2Table extraction models
- Document the data flow and processing mechanisms
- Provide interface specifications for all system components
- Establish traceability between requirements and design elements

## 1.2 Document Convention

The following conventions are used throughout this document:

- **Bold text**: Indicates important terms, component names, or emphasis
- *Italic text*: Denotes technical terminology or references
- `Monospace font`: Represents code, file names, API endpoints, or configuration values
- Blue hyperlinks: Cross-references to other sections or external resources
- Numbered lists: Sequential steps or ordered elements
- Bulleted lists: Non-sequential items or features

**Terminology Standards:**

- LLM: Large Language Model (Qwen2.5-VL-7B-Instruct)
- VLM: Vision-Language Model
- Chart2Table: PaddlePaddle-based chart data extraction model
- DePlot: Google's Pix2Struct-based chart delinearization model
- CoT: Chain-of-Thought reasoning
- RTM: Requirements Traceability Matrix

## 1.3 Intended Audience

This document is intended for:

1. **Software Development Team**: Students implementing the system components, API endpoints, and model integration pipelines
2. **Supervisors**: Stakeholders tracking progress and ensuring alignment with requirements
3. **Maintenance Personnel**: Future maintainers requiring understanding of system design and interfaces
4. **Academic Reviewers**: Faculty and evaluators assessing the graduation project

## 1.4 Additional Information

**Document Version History:**

- Version 1.0 (Current): Initial release incorporating complete system design

**Related Documentation:**

- Requirements Specification Document (RSD)
- Research Paper: "Table First Guarded Reasoning: Outperforming Larger MLLMs in Chart Question Answering"
- API Documentation
- Github Repository

**Development Environment:**

- Primary Framework: PyTorch 2.0+, Transformers 4.35+
- Backend: FastAPI, Uvicorn
- Frontend: PyQt6
- Model Serving: Distributed GPU architecture
- Dataset: HuggingFaceM4/ChartQA benchmark

## 1.5 Contact Information/SDS Team Members

| Name | **Contact** |
|---|---|
| Andrew Aziz | andrewwassem@aucegypt.edu |
| Kirolous Fouty | kirolous_fouty@aucegypt.edu |
| MagdElDin AbdalRaaof | magdeldin@aucegypt.edu |
| Mohamed Abbas | Mohamed_Ragab02@aucegypt.edu |
| Shaza Ali | shazamamdouh@aucegypt.edu |
| Tarek Kassab | tarek_kassab@aucegypt.edu |

## 1.6 References


1. Qwen Team. "Qwen2.5-VL: Vision-Language Model Documentation." Alibaba Cloud, 2024.
2. Liu, F., et al. "DePlot: One-shot visual language reasoning by plot-to-table translation." ACL 2023.
3. PaddlePaddle Team. "PP-StructureV3: Layout Analysis and Table Recognition." Baidu, 2024.
4. Masry, A., et al. "ChartQA: A Benchmark for Question Answering about Charts with Visual and Logical Reasoning." ACL 2022.
5. Wei, J., et al. "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models." NeurIPS 2022.
6. Requirements Specification Document v1.0
7. Research Paper: "Table First Guarded Reasoning: Outperforming Larger MLLMs in Chart Question Answering"

# 2 Design Overview

## 2.1 Background Information

The Chart Understanding System represents a sophisticated approach to automated chart analysis and question answering, addressing the critical challenge of extracting insights from visual data representations. The system builds upon recent advances in vision-language models and structured prompting techniques.

**Problem Domain:** Charts and graphs are ubiquitous in academic research, business analytics, and data journalism. However, interpreting these visualizations often requires:

- Visual perception of chart elements (axes, legends, data points)
- Numerical reasoning and calculation capabilities
- Domain knowledge for contextual understanding
- Synthesis of visual and textual information

**Solution Approach:** The system implements a hybrid architecture combining:

1. **Primary VLM**: Qwen2.5-VL-7B-Instruct for visual understanding and reasoning
2. **Auxiliary Models**: Chart2Table (PaddlePaddle) or DePlot (Pix2Struct) for structured data extraction
3. **Multi-Strategy Framework**: Eight distinct prompting strategies ranging from baseline to hybrid chain-of-thought approaches
4. **Distributed Architecture**: Microservice-based design for scalability and modularity

**Key Innovations:**

- Chain-of-models architecture combining visual and tabular reasoning
- Hybrid prompting strategies (Chart2Table-CoT, DePlot-CoT)
- Comparative evaluation framework across multiple strategies
- Real-time dual-model comparison interface

## 2.2 Constraints

### 2.2.1 Technical Constraints

1. **Hardware Resources**:
   - GPU memory: 24GB minimum per model server
   - VRAM allocation: 75% maximum per process
   - Multi-GPU deployment required for full pipeline
   - CUDA 11.8+ compatibility required
2. **Model Limitations**:
   - Qwen2.5-VL: 4096 token input limit, 512-1024 token output
   - Chart2Table: Variable accuracy on complex multi-chart figures
   - DePlot: 2048 patch limit for image processing
   - No fine-tuning capability (inference-only deployment)
3. **Performance Requirements**:
   - Base model inference: 1-3 seconds per query
   - Chain-of-models (with Chart2Table): 3-7 seconds total

- API timeout: 60 seconds maximum
- Concurrent request handling: 4-8 simultaneous queries

4. **Software Dependencies**:
   - PyTorch framework (no TensorFlow migration possible)
   - PaddlePaddle environment isolation required
   - Python 3.9-3.11 compatibility only
   - No browser storage APIs in artifact environment

#### 2.2.2 Operational Constraints

1. **Deployment Environment**:
   - Local network deployment only (no cloud hosting)
   - Fixed IP addressing for server endpoints
   - Manual model loading on system startup
   - No automatic scaling or load balancing
2. **Data Constraints**:
   - Image formats: PNG, JPG, JPEG only
   - Maximum image size: 10MB per upload
   - No persistent storage for user data
   - In-memory state management only
3. **Integration Constraints**:
   - REST API communication only
   - Synchronous request-response pattern
   - No message queue or async processing
   - Limited error recovery mechanisms

### 2.3 Design Trade-offs

#### 2.3.1 Architecture Trade-offs

| **Decision** | **Advantages** | **Disadvantages** |
|---|---|---|
| Microservice Architecture | Modularity, independent scaling, fault isolation | Increased complexity, network overhead, latency |
| Separate PaddlePaddle Server | Environment isolation, GPU memory optimization | Additional deployment overhead, API latency |
| REST API over gRPC | Simplicity, wide tooling support, debugging ease | Lower performance, higher bandwidth usage |
| No Persistent Storage | Simplicity, no database management, data privacy | Loss of session history, repeated processing |
| Synchronous Processing | Predictable behavior, simpler error handling | User wait time, no background processing |

---

### 2.3.2 Model Selection Trade-offs

| Choice | Rationale | Trade-off |
|---|---|---|
| Qwen2.5-VL-7B vs 72B | Fits in single GPU, faster inference, lower latency | Reduced reasoning capability vs larger variant |
| Chart2Table vs DePlot | Better table structure, Chinese chart support | Requires separate PaddlePaddle environment |
| BFloat16 Precision | Memory efficiency, speed improvement, hardware support | Minimal accuracy loss ( 0.1% vs FP32) |
| Multiple Strategy Evaluation | Comprehensive comparison, research value | 8x computational cost, extended testing time |

### 2.3.3 User Experience Trade-offs

- **Dual-Model Comparison**: Provides valuable insights but doubles inference time
- **No Streaming Responses**: Simpler implementation but perceived latency for users
- **Manual Endpoint Configuration**: Flexibility for testing but requires technical knowledge
- **Image Upload Requirement**: Ensures quality inputs but limits URL-based queries

## 2.4 User Characteristics

### 2.4.1 Primary User Personas

**Persona 1: Research Scientist**

- **Technical Expertise**: High (PhD-level, programming proficiency)
- **Domain Knowledge**: Deep understanding of data analysis and visualization
- **Primary Goal**: Extract precise numerical data from published charts for meta-analysis
- **Key Requirements**: Accuracy over speed, detailed extraction tables, comparison capabilities
- **Usage Pattern**: Batch processing of 10-100 charts, requires reproducible results

**Persona 2: Data Analyst**

- **Technical Expertise**: Medium (familiar with data tools, limited ML knowledge)
- **Domain Knowledge**: Strong business analytics background
- **Primary Goal**: Quickly answer questions about chart trends and comparisons
- **Key Requirements**: Speed, ease of use, natural language interaction
- **Usage Pattern**: Interactive exploration of 5-20 charts per session

**Persona 3: ML Engineering Student**

- **Technical Expertise**: Medium to High (learning phase)
- **Domain Knowledge**: Growing understanding of VLMs and prompting
- **Primary Goal**: Understand and compare different prompting strategies
- **Key Requirements**: Educational value, transparency, experimentation support
- **Usage Pattern**: Testing various strategies on benchmark datasets

#### 2.4.2 User Skill Requirements

- **Minimal Requirements**: Ability to formulate clear questions about charts, basic file upload operations
- **Advanced Features**: Understanding of model endpoints, interpretation of strategy comparisons
- **System Administration**: Python environment management, GPU configuration, API server deployment

#### 2.4.3 Accessibility Considerations

- Clear visual feedback for processing status
- Keyboard navigation support in PyQt6 interface
- Resizable windows and adjustable text display
- Error messages with actionable guidance
- No reliance on color alone for status indication

# 3 System Architecture

## 3.1 Hardware Architecture

### 3.1.1 Hardware Component Overview

The system requires a distributed GPU computing environment to support concurrent model serving:

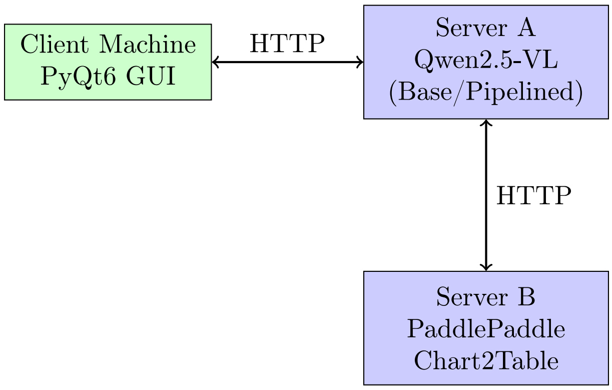


Figure 1: Hardware Deployment Architecture

### 3.1.2 Server Specifications

**Server A: Base Model Server / Pipelined Model Server**

- **Purpose**: Hosts Qwen2.5-VL-7B-Instruct model
- **GPU**: NVIDIA GPU with 24GB+ VRAM (e.g., RTX 3090, RTX 4090, A5000)
- **CPU**: 8+ cores recommended for preprocessing
- **RAM**: 32GB+ system memory
- **Storage**: 50GB+ for model weights and dependencies
- **Network**: Gigabit Ethernet for inter-server communication
- **IP Address**: 10.40.32.8 (Base), 10.40.32.12 (Pipelined)
- **Port**: 8001

**Server B: PaddlePaddle Server**

- **Purpose**: Hosts Chart2Table extraction model
- **GPU**: NVIDIA GPU with 12GB+ VRAM (can use 30% GPU memory)
- **CPU**: 4+ cores sufficient
- **RAM**: 16GB+ system memory
- **Storage**: 30GB+ for PaddlePaddle models
- **Network**: Gigabit Ethernet
- **IP Address**: localhost (deployed on Pipelined Server)
- **Port**: 8000

**Client Machine**

- **Purpose**: Runs PyQt6 GUI application

- **GPU**: Not required
- **CPU**: Dual-core or better
- **RAM**: 4GB+ system memory
- **OS**: Windows 10/11, Linux, or macOS
- **Network**: Standard Ethernet/WiFi connection

#### 3.1.3 Hardware Communication

- **Protocol**: HTTP/1.1 over TCP/IP
- **Bandwidth Requirements**:
  - Client to Server: 1-5 Mbps (image uploads)
  - Server to PaddlePaddle: 0.5-2 Mbps (base64 images)
- **Latency Tolerance**: $<$100ms network RTT preferred
- **Concurrency**: Support for 4-8 simultaneous client connections

### 3.2 Software Architecture

#### 3.2.1 Architectural Pattern

The system employs a **Multi-Tier Microservice Architecture** with three primary tiers:

1. **Presentation Tier**: PyQt6 desktop application (Client)
2. **Application Tier**: FastAPI model servers (Server A, Server B)
3. **Processing Tier**: ML model inference engines (Qwen, Chart2Table)

#### 3.2.2 Component Diagram

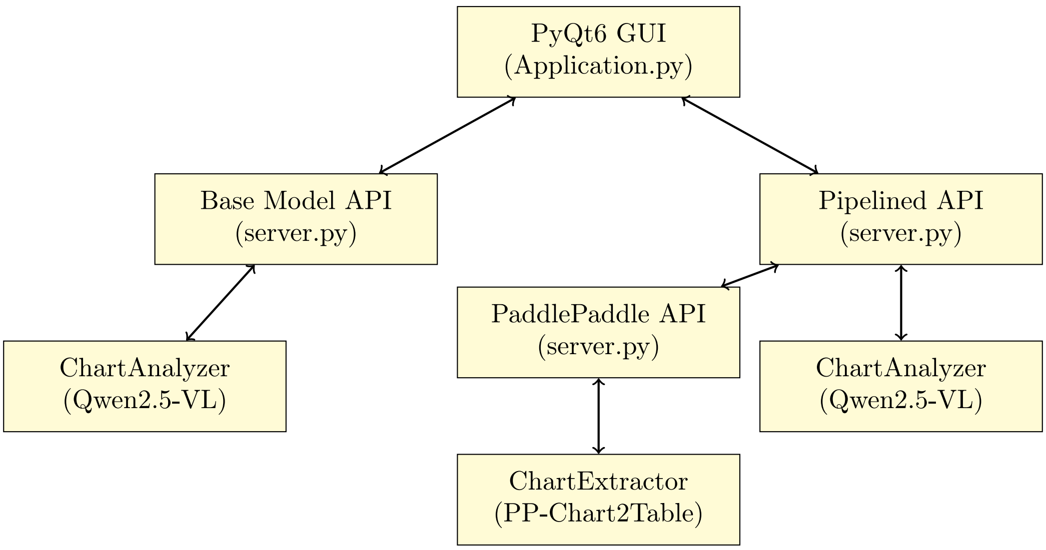


Figure 2: Software Component Architecture

#### 3.2.3 Core Software Components

**1. Client Application (Application.py)**

```
class CompareApp(QMainWindow):
    - model1_url_input: QLineEdit
    - model2_url_input: QLineEdit
    - question_edit: QTextEdit
    - image_label: QLabel

    Methods:
    - on_attach(): Upload image
    - on_send(): Send to both servers
    - on_worker_finished(): Handle response
```

**Features**:

- Dual-endpoint configuration
- Image upload and preview
- Concurrent request handling via QThread workers
- Side-by-side response comparison
- JSON export capabilities

**2. Base Model Server (Base Model Server/server.py)**

```
class ChartAnalyzer:
    - model: Qwen2_5_VLForConditionalGeneration
    - processor: AutoProcessor

    Methods:
    - generate_response(image, prompt)

Endpoint: POST /predict
    Input: image (file), question (form)
    Output: {"answer": str}
```

**Features**:

- Direct Qwen inference without preprocessing
- Supports all non-chain strategies
- GPU memory management (75% allocation)
- BFloat16 precision for efficiency

**3. Pipelined Model Server (Pipelined Model Server/Model Server/server.py)**

```
class ChartAnalyzer + ChartExtractor:

Endpoint: POST /predict
    Process:
    1. Extract table via Chart2Table API
    2. Clean table output
    3. Build chart2table_cot prompt
    4. Generate Qwen response

    Output: {
        "answer": str,
        "cleaned_table": str,
        "strategy": "chart2table_cot"
    }
```

**Features**:

- Chain-of-models execution
- Automatic prompt construction
- Timing metrics for each stage
- Error handling with detailed traces

4. **PaddlePaddle Server (PaddlePaddle Server/server.py)**

```
class ChartExtractor:
    - model: create_model('PP-Chart2Table')

    Methods:
    - extract(image): str

Endpoints:
    - POST /extract: File upload
    - POST /extract-base64: Base64 image
    - GET /health: Health check

Output: {
    "success": bool,
    "table_clean": str,
    "rows": list[dict],
    "headers": list[str]
}
```

**Features**:

- PaddlePaddle environment isolation
- Table structure parsing
- 30% GPU memory allocation
- Dual input format support

#### 3.2.4 Software Dependencies

| Component | Version | Purpose |
|---|---|---|
| PyTorch | 2.0+ | Deep learning framework for Qwen |
| Transformers | 4.35+ | Model loading and inference |
| qwen-vl-utils | Latest | Vision input processing |
| FastAPI | 0.100+ | REST API framework |
| Uvicorn | 0.23+ | ASGI server |
| PyQt6 | 6.5+ | Desktop GUI framework |
| PaddlePaddle | 2.5+ | Chart2Table model framework |
| PaddleOCR | 2.7+ | OCR and structure recognition |
| Pillow | 10.0+ | Image processing |
| NumPy | 1.24+ | Numerical operations |
| Requests | 2.31+ | HTTP client library |

### 3.3 Communication Architecture

#### 3.3.1 Communication Protocols

**REST API Communication**

- **Protocol**: HTTP/1.1
- **Format**: Multipart form-data (image upload), JSON (responses)
- **Encoding**: UTF-8 for text, Base64 for images

- **Timeout**: 60 seconds per request
- **Error Handling**: HTTP status codes + JSON error messages

#### 3.3.2 API Specifications

**Base/Pipelined Model Server API**

```
POST /predict
Content-Type: multipart/form-data

Request:
    question: string (required)
    image: file (required, PNG/JPG/JPEG)

Response (Base):
{
    "answer": "string"
}

Response (Pipelined):
{
    "answer": "string",
    "cleaned_table": "string",
    "strategy": "chart2table_cot"
}

Error Response:
{
    "answer": "Error message",
    "error": "string",
    "trace": "string"  // optional
}
```

**PaddlePaddle Server API**

```
GET /health
Response: {
    "status": "healthy",
    "service": "PaddlePaddle Chart2Table API"
}

POST /extract-base64
Content-Type: application/json

Request:
{
    "image_base64": "string"
}

Response:
{
    "success": true,
    "raw_table": "string",
    "table_clean": "string",
    "rows": [...],
    "headers": [...],
    "num_rows": int,
    "num_cols": int
}
```

#### 3.3.3 Data Flow Diagrams

**Scenario 1: Base Model Request**

1. User uploads image + types question in GUI
2. GUI sends POST to Base Model Server (10.40.32.8:8001/predict)
3. Server saves image to temp file
4. ChartAnalyzer processes with Qwen directly
5. Server returns JSON response
6. GUI displays answer

**Scenario 2: Pipelined Model Request**

1. User uploads image + types question in GUI
2. GUI sends POST to Pipelined Server (10.40.32.12:8001/predict)
3. Server saves image to temp file
4. Server sends image to PaddlePaddle API (localhost:8000/extract-base64)
5. PaddlePaddle returns extracted table
6. Server cleans table and builds CoT prompt
7. ChartAnalyzer generates response with table
8. Server returns JSON with answer + table
9. GUI displays both answer and extracted table

**Scenario 3: Dual Comparison Request**

1. User configures both endpoints and clicks "Send to both models"
2. GUI spawns two QThread workers concurrently
3. Worker A sends request to Base Model Server
4. Worker B sends request to Pipelined Server
5. Both workers emit finished signals with responses
6. GUI displays side-by-side comparison

#### 3.3.4 Network Configuration

| **Component** | **Address** | **Port** | **Purpose** |
|---|---|---|---|
| Base Server | 10.40.32.8 | 8001 | Qwen direct inference |
| Pipelined Server | 10.40.32.12 | 8001 | Chain-of-models |
| PaddlePaddle API | localhost | 8000 | Chart extraction |
| Client | Dynamic | N/A | GUI application |

# 4 Data Design

## 4.1 Database Management System Files

The system does **not employ a traditional database management system**. All data is handled ephemerally in-memory or as temporary files. This design choice prioritizes:

- Simplicity and reduced infrastructure requirements
- Data privacy (no persistent storage of user uploads)
- Stateless service design for easier scaling
- Elimination of database administration overhead

## 4.2 Non-Database Management System Files

### 4.2.1 Configuration Files

**1. Model Configuration (server.py - Config class)**

```
class Config:
    # GPU Settings
    GPU_MEMORY_FRACTION = 0.75
    ENABLE_TF32 = True

    # Model Paths
    LLM_MODEL = "Qwen/Qwen2.5-VL-7B-Instruct"

    # API Configuration
    PADDLE_API_URL = "http://localhost:8000"
    PADDLE_API_TIMEOUT = 30

    # Token Limits
    MAX_INPUT_TOKENS = 4096
    MAX_NEW_TOKENS = 128
```

**2. PaddlePaddle Configuration**

```
os.environ['FLAGS_fraction_of_gpu_memory_to_use'] = '0.3'
```

### 4.2.2 Input Data Files

**Image Files (Temporary)**

- **Location**: Server temp directory
- **Filename**: `temp_upload.jpg`
- **Format**: RGB JPEG
- **Lifecycle**: Created on request, overwritten on next request
- **Access**: Read-only by model inference

**Benchmark Dataset (Optional - Evaluation Only)**

- **File**: `benchmark.json`
- **Format**: JSON array of objects
- **Structure**:

```
[
  {
    "id": "chart_0",
    "image_path": "benchmark_images/chart_0.png",
    "question": "What is the value of X?",
    "answer": ["42"],
    "source": "HuggingFaceM4/ChartQA"
  },
  ...
]
```

- **Size**: 2000 examples, 500MB total with images

#### 4.2.3 Output Data Files

**Evaluation Results (Research Mode)**

- **File**: `results_*.json`
- **Purpose**: Store strategy comparison metrics
- **Structure**:

```
{
  "strategy_name": {
    "accuracy": 0.75,
    "correct_count": 150,
    "total": 200,
    "avg_inference_time": 2.3,
    "avg_total_time": 5.1,  // for chain strategies
    "results": [
      {
        "id": "chart_0",
        "question": "...",
        "predicted": "...",
        "ground_truth": "...",
        "correct": true,
        "inference_time": 2.1,
        "extracted_table": "..."  // for chain strategies
      }
    ]
  }
}
```

**Exported User Results**

- **Format**: Plain text (.txt) or JSON (.json)
- **Content**: User-selected answer text or full response JSON
- **Location**: User-specified via file dialog

#### 4.2.4 Log Files

**Server Logs**

- **Format**: Standard output (stdout) via Uvicorn
- **Content**: Request timestamps, errors, model loading status
- **Persistence**: Terminal session only (not written to disk)
- **Level**: INFO for normal operation, ERROR for failures

**Example Log Output**:

```
INFO: Loading Qwen2.5-VL ChartAnalyzer...
INFO: Model loaded successfully!
INFO: Server ready.
INFO: Uvicorn running on http://0.0.0.0:8001
INFO: 10.40.32.1:54321 - "POST /predict HTTP/1.1" 200 OK
```

# 5 Detailed Design

## 5.1 Hardware Detailed Design

### 5.1.1 GPU Memory Management Strategy

**Memory Allocation Pattern**:

```
# Qwen Server (75% allocation)
torch.cuda.set_per_process_memory_fraction(0.75, 0)
torch.backends.cuda.matmul.allow_tf32 = True

# Model Loading
model = Qwen2_5_VLForConditionalGeneration.from_pretrained(
    model_path,
    torch_dtype=torch.bfloat16,  # ~50% memory vs FP32
    device_map="auto"  # Automatic GPU placement
)
```

**Memory Breakdown (24GB GPU)**:

- Model weights (BFloat16): 14GB
- KV cache during inference: 2-4GB
- Activation memory: 1-2GB
- Image embeddings: 500MB
- Reserved for system: 6GB (25%)

### 5.1.2 Thermal and Power Considerations

- **Expected GPU Load**: 70-90% during inference, <10% idle
- **Power Draw**: 250-350W per GPU under load
- **Cooling**: Active cooling required, maintain <80°C junction temp
- **Power Supply**: 750W+ recommended for workstation deployments

## 5.2 Software Detailed Design

### 5.2.1 Core Algorithm: ChartAnalyzer.generate_response()

**Algorithm Purpose**: Generate textual response for chart + question using Qwen2.5-VL
**Inputs**:

- `image_path`: String path to chart image
- `prompt`: String containing question and strategy-specific instructions
- `max_new_tokens`: Integer (default 512)
- `temperature`: Float (default 0.1 for factual tasks)

**Algorithm Steps**:

1. **Message Construction**:

```
messages = []
messages.append({
    "role": "user",
    "content": [
        {"type": "image", "image": image_path},
        {"type": "text", "text": prompt}
    ]
})
```

2. **Chat Template Application**:

```
text = self.processor.apply_chat_template(
    messages,
    tokenize=False,
    add_generation_prompt=True
)
```

3. **Vision Input Processing**:

```
from qwen_vl_utils import process_vision_info
image_inputs, video_inputs = process_vision_info(messages)
```

4. **Tokenization**:

```
inputs = self.processor(
    text=[text],
    images=image_inputs,
    videos=video_inputs,
    padding=True,
    return_tensors="pt"
).to(self.model.device)
```

5. **Generation**:

```
with torch.no_grad():
    generated_ids = self.model.generate(
        **inputs,
        max_new_tokens=max_new_tokens,
        temperature=temperature,
        top_p=0.9,
        do_sample=temperature > 0
    )
```

6. **Decoding**:

```
generated_ids_trimmed = [
    out_ids[len(in_ids):]
    for in_ids, out_ids in zip(inputs.input_ids, generated_ids)
]
response_text = self.processor.batch_decode(
    generated_ids_trimmed,
    skip_special_tokens=True
)[0]
```

**Output**: String containing model's response
**Complexity Analysis**:

- Time: O(T) where T = `max_new_tokens` (autoregressive generation)
- Space: O(N + I) where N = input tokens, I = image embedding size

### 5.2.2 Prompting Strategy Implementation

**Strategy 1: Baseline**

```
@staticmethod
def baseline(question: str) -> str:
    return f""Question: {question}
Answer the question using a single word or phrase.
End your response with: Final Answer: [your answer]""
```

**Design Rationale**:

- Minimal prompt overhead for speed comparison

- Explicit output format constraint
- Direct question-answer paradigm

**Strategy 2: Zero-Shot Chain-of-Thought**

```
@staticmethod
def zero_shot_cot(question: str) -> str:
    return f""Question: {question}
Let us solve this step by step with careful analysis.
*Use digits (e.g., 3, 20.5) for all numbers*
1. Identify chart type and analyze axes/legend:
2. Extract relevant numbers from the chart:
3. Determine calculation/reasoning needed:
4. Perform calculation step-by-step:
5. Verify answer makes sense:
6. Final Answer:""
```

**Design Rationale**:

- Decomposes reasoning into explicit steps
- Enforces numerical output format
- Includes verification step to reduce errors

**Strategy 3: Chart2Table + CoT Hybrid (Key Innovation)**

```
@staticmethod
def chart2table_cot(question: str, chart2table_table: str) -> str:
    return f""You are analyzing a chart using both visual
    information and extracted data.

**Extracted Table Data from Chart:**
{chart2table_table}

**Question:** {question}

**Instructions:** Solve step-by-step using BOTH
the table data and the visual chart.

1. **Understand the Question**
2. **Analyze Chart Visually**
3. **Cross-Reference with Table Data**
4. **Perform Calculations** (show work)
5. **Verify Answer**
6. **Provide Final Answer**

**Your Step-by-Step Solution:**""
```

**Design Rationale**:

- Combines structured data (eliminates OCR errors) with visual context
- CoT framework reduces calculation mistakes
- Cross-verification between table and chart catches inconsistencies
- Expected to achieve highest accuracy (hypothesis)

#### 5.2.3 Data Cleaning Algorithm: clean_chart2text_output()

**Purpose**: Convert Chart2Table's linearized output to readable format

**Input Format Example**:

```
"Year<0x09>Sales<0x0A>2020<0x09>100<0x0A>2021<0x09>150"
```

**Algorithm**:

```python
def clean_chart2text_output(table_text: str) -> str:
    # 1. Replace hex newline
    cleaned = table_text.replace("<0x0A>", "\n")

    # 2. Replace hex tab with pipe
    cleaned = cleaned.replace("<0x09>", "␣|␣")

    # 3. Normalize spaces around pipes
    cleaned = cleaned.replace("|", "␣|␣")
    cleaned = "␣".join(cleaned.split())

    # 4. Restore newlines and remove empty lines
    lines = [line.strip() for line in cleaned.split('\n')]
    cleaned = "\n".join([l for l in lines if l])

    return cleaned
```

**Output Format Example**:

```
Year | Sales
2020 | 100
2021 | 150
```

#### 5.2.4 Answer Evaluation Algorithm

**Function**: `evaluate_answer(predicted, ground_truth, tolerance=0.05)`
**Purpose**: Compare model prediction against ground truth with flexible matching
**Evaluation Pipeline**:

1. **Handle ChartQA List Format**:

```python
if isinstance(ground_truth, list):
    gt_list = [str(g) for g in ground_truth]
else:
    gt_list = [str(ground_truth)]
```

2. **JSON Extraction** (for structured outputs):

```python
json_match = re.search(r'```json\s*(\{.*?\})\s*```', predicted)
if json_match:
    data = json.loads(json_match.group(1))
    predicted = str(data["answer"])
```

3. **Pattern-Based Answer Extraction**:

```python
patterns = [
    r'final\s+answer\s*:?\s*(.+?)(?:\n|$)',
    r'the\s+answer\s+is\s*:?\s*(.+?)(?:\n|$)',
]
```

4. **Text Normalization**:
   - Convert to lowercase
   - Word-to-digit conversion (e.g., "five" → "5")
   - Remove currency symbols ($, €, £)
   - Remove thousands separators (1,234 → 1234)

5. **Matching Strategies** (evaluated in order):
   (a) Exact string match
   (b) Substring match (for non-numeric answers)
   (c) Numerical comparison with 5% relative tolerance

(d) Percentage format mismatch handling (0.25 vs 25%)

**Tolerance Formula**:

```
relative_error = abs(pred_num - gt_num) / abs(gt_num)
is_correct = relative_error <= 0.05  # 5% tolerance
```

## 5.3 Communication Detailed Design

### 5.3.1 HTTP Request-Response Cycle

**Client-Side Request (ModelWorker)**:

```
class ModelWorker(QThread):
    def run(self):
        files = {}
        data = {"question": self.question}
        if self.image_bytes:
            files["image"] = ("image.jpg",
                              self.image_bytes,
                              "image/jpeg")

        resp = requests.post(
            self.url,
            files=files,
            data=data,
            timeout=self.timeout
        )

        json_resp = resp.json()
        self.finished_signal.emit(json_resp, self.model_name)
```

**Server-Side Handling (FastAPI)**:

```
@app.post("/predict")
async def predict(
    question: str = Form(...),
    image: UploadFile = File(...)
):
    # Read image
    image_bytes = await image.read()
    pil = Image.open(io.BytesIO(image_bytes))

    # Save temporarily
    pil.save("temp_upload.jpg")

    # Process (varies by server type)
    # - Base: Direct Qwen inference
    # - Pipelined: Chart2Table -> Qwen

    # Return JSON
    return JSONResponse({
        "answer": llm_output,
        # Additional fields for pipelined
    })
```

### 5.3.2 Error Handling Strategy

**Error Categories and Responses**:

| Error Type | Detection | Response |
|---|---|---|
| Network Timeout | `requests.exceptions.Timeout` | Retry prompt, increase timeout setting |

| Error Type | Detection | Response |
|---|---|---|
| Connection Refused | `ConnectionError` | Check server status, verify endpoints |
| Invalid Image | `PIL.UnidentifiedImageError` | Return 400 with error message |
| GPU OOM | `torch.cuda.OutOfMemoryError` | Restart server, reduce batch size |
| Model Loading Fail | Exception during init | Log full trace, halt server |
| Chart2Table API Down | HTTP error from /health | Graceful degradation, disable pipelined mode |

**Error Response Format**:

```
{
    "answer": "Server error: {brief_message}",
    "error": "detailed_error_string",
    "trace": "full_traceback"  // optional, debug mode
}
```

# 6 Interfacing to External Systems

## 6.1 HuggingFace Model Hub Integration

**Purpose**: Download pre-trained model weights
**Models Downloaded**:

- `Qwen/Qwen2.5-VL-7B-Instruct` ( 15GB)
- `google/deplot` (optional, 2GB)

**Integration Method**:

```
from transformers import Qwen2_5_VLForConditionalGeneration

model = Qwen2_5_VLForConditionalGeneration.from_pretrained(
    "Qwen/Qwen2.5-VL-7B-Instruct",
    torch_dtype=torch.bfloat16,
    device_map="auto",
    trust_remote_code=True  # Required for custom code
)
```

**Authentication**: Not required (public models)
**Offline Mode**: Models cached locally in `~/.cache/huggingface/`

## 6.2 PaddlePaddle Model Zoo Integration

**Purpose**: Load PP-Chart2Table for data extraction
**Integration Method**:

```
from paddlex import create_model

model = create_model('PP-Chart2Table')
```

**Model Source**: Baidu's PaddleX framework
**Requirements**: Separate conda/venv with PaddlePaddle installed

## 6.3 ChartQA Dataset Interface

**Purpose**: Load benchmark dataset for evaluation
**Integration Method**:

```
from datasets import load_dataset

chartqa = load_dataset("HuggingFaceM4/ChartQA", split="test")
```

**Data Format**:

- **Fields**: image (PIL.Image), query (str), label (list[str])
- **Size**: 2000 test examples
- **License**: Apache 2.0

**Usage Pattern**:

1. Download dataset on first run (cached locally)
2. Save images to `benchmark_images/` directory
3. Create JSON manifest with local paths
4. Use manifest for batch evaluation

## 6.4 External Dependencies Summary

| System | Type | Interface Method |
|---|---|---|
| HuggingFace Hub | Model Repository | `transformers.from_pretrained()` |
| PaddleX Model Zoo | Model Repository | `paddlex.create_model()` |
| HuggingFace Datasets | Data Repository | `datasets.load_dataset()` |
| PyPI | Package Repository | `pip install` |
| Conda | Package Manager | `conda install paddlepaddle-gpu` |

# 7 Usability Design Approach

## 7.1 User Interface Design Principles

The GUI design adheres to the following principles:

1. **Simplicity**: Minimal learning curve with clear, self-explanatory controls
2. **Transparency**: Show what's happening (processing status, timing metrics)
3. **Flexibility**: Support various workflows (single query, comparison, batch)
4. **Responsiveness**: Non-blocking UI with progress indicators
5. **Error Recovery**: Clear error messages with actionable guidance
6. **Consistency**: Uniform layout and interaction patterns

**Design Constraints Addressed**:

- Must work with fixed server endpoints (no dynamic discovery)
- Support concurrent requests without blocking UI
- Display both brief answers and detailed JSON responses
- Allow result export in multiple formats

## 7.2 Screens Implemented

### 7.2.1 Main Application Window

**Layout**: Horizontal split with left input panel and right comparison panel
**Components**:
**Left Panel (Input Section)**:

- **Endpoint Configuration**:
  - Model A URL: `QLineEdit` (default: http://10.40.32.8:8001/predict)
  - Model B URL: `QLineEdit` (default: http://10.40.32.12:8001/predict)
- **Question Input**: `QTextEdit` with placeholder "Type the user's question here..."
- **Image Controls**:
  - "Attach image" button: Opens file picker (PNG/JPG/JPEG)
  - "Clear image" button: Removes attached image
  - Image preview: 360x270px with aspect ratio preserved
- **Action Controls**:
  - "Send to both models" button: Triggers dual comparison
  - Progress bar: 0-2 range showing completed requests

**Right Panel (Results Section)**:

- **Model A Result Box**:
  - Info label: Shows confidence/timing metrics
  - Answer display: `QTextEdit` (read-only, scrollable)
  - Action buttons: "Show raw JSON", "Save text", "Save raw JSON"
- **Model B Result Box**:
  - Identical structure to Model A
  - Side-by-side for easy comparison
- **Resizable Splitter**: Allows adjusting relative panel sizes

#### 7.2.2 Dialog Windows

**Raw JSON Viewer Dialog**:

- Large text area (700x500px) with full JSON response
- Attempts to unescape backslash sequences for readability
- Buttons: "Close", "Save"

**File Save Dialogs**:

- Standard OS file picker
- Filters: "Text files (*.txt)" or "JSON files (*.json)"
- Default names: Based on content type

**Message Boxes**:

- **Warnings**: Empty question, missing image, invalid endpoints
- **Errors**: Network failures, server errors
- **Information**: Save confirmations, operation results

### 7.3 User Interaction Flow

#### 7.3.1 Primary Use Case: Dual Model Comparison

1. **Setup Phase**:
   (a) User verifies/modifies endpoint URLs
   (b) User clicks "Attach image" and selects chart file
   (c) Image appears in preview area
2. **Query Phase**:
   (a) User types question in text area
   (b) User clicks "Send to both models"
   (c) UI disables send button, shows "Waiting for response..." in both panels
   (d) Progress bar animates ($0 \rightarrow 1 \rightarrow 2$)
3. **Response Phase**:
   (a) As each model responds:
      - Info label updates with timing/confidence
      - Answer appears in text area
      - Progress bar increments
   (b) When both complete:
      - Send button re-enables
      - User can compare answers side-by-side
4. **Inspection Phase**:
   (a) User clicks "Show raw JSON" to see full response
   (b) User examines extracted table (if pipelined model)
   (c) User compares reasoning between models
5. **Export Phase** (Optional):
   (a) User clicks "Save text" or "Save raw JSON"
   (b) System opens file picker
   (c) User selects location and filename
   (d) Confirmation message appears

### 7.3.2 Alternate Flow: Single Model Query

User can query only one model by:

1. Leaving one endpoint blank
2. Observing that only the configured model receives the request
3. Reviewing result in the corresponding panel

## 7.4 Wireframe

**Main Window Wireframe**:

```
+-----------------------------------------------------------------+
|  Model Comparator                                        [_][][X]|
+-----------------------------------------------------------------+
| [Input Panel]                   | [Results Panel]              |
|                                 |                              |
| Model A endpoint:               | +------------------------+   |
| [http://10.40.32.8:8001/predict] | | Model A              |   |
|                                 | | Info: confidence: 0.9 |   |
| Model B endpoint:               | | round-trip: 2300 ms   |   |
| [http://10.40.32.12:8001/predict]| |                      |   |
|                                 | | [Answer Text Area]    |   |
| Question:                       | | The answer is 42.5    |   |
| +------------------------------+ | | based on the chart... |   |
| | What is the maximum value     | | |                      |   |
| | shown in the bar chart?       | | | [Show raw JSON]      |   |
| |                               | | | [Save text]          |   |
| +------------------------------+ | | [Save raw JSON]      |   |
|                                 | +------------------------+   |
| [Attach image] [Clear image]    |                              |
|                                 | +------------------------+   |
| +---------------------------+   | | Model B               |   |
| |                           |   | | Info: total: 5100 ms  |   |
| |   [Image Preview]         |   | |                       |   |
| |   360 x 270 px            |   | | [Answer Text Area]    |   |
| |                           |   | | The maximum value     |   |
| +---------------------------+   | | is 42.5.              |   |
|                                 | |                       |   |
| [Send to both models]           | | [Show raw JSON]       |   |
| [] 1/2            | | [Save text]            |   |
|                                 | | [Save raw JSON]       |   |
+---------------------------------+--------------------------+ |
| Status: Waiting for Model B...                               |
+-----------------------------------------------------------------+
```

**Raw JSON Viewer Dialog Wireframe**:

```
+-------------------------------------------------------+
| Raw JSON                                      [_][][X]|
+-------------------------------------------------------+
| +---------------------------------------------------+ |
| | {                                                 | |
| |   "answer": "The maximum value is 42.5",         | |
| |   "cleaned_table": "Category | Value\nA | 25...", | |
| |   "strategy": "chart2table_cot",                 | |
| |   "_client_elapsed_ms": 5100                     | |
| | }                                                 | |
| |                                                   | |
| |                                                   | |
| +---------------------------------------------------+ |
|                                                       |
|                          [Save]  [Close]              |
+-------------------------------------------------------+
```

**File Save Dialog Wireframe**:

```
+----------------------------------------------------+
| Save Answer Text                           [_][][X]|
+----------------------------------------------------+
| Save in: [My Documents ]                           |
|                                                    |
| [] Desktop                                         |
| [] Documents                                       |
| [] Downloads                                       |
| [] chart_results                                   |
|                                                    |
| File name: [answer_chart_0.txt          ]          |
| Save as type: [Text files (*.txt) ]                |
|                                                    |
|                              [Save]  [Cancel]      |
+----------------------------------------------------+
```

**Error Message Box Wireframe**:

```
+-------------------------------------+
|   Warning                      [X]  |
+-------------------------------------+
|                                     |
| Empty question                      |
|                                     |
| Please type a question before       |
| sending.                            |
|                                     |
|                    [OK]             |
+-------------------------------------+
```

## 7.5 Accessibility Features

**Keyboard Navigation**:

- Tab order: Endpoints → Question → Attach → Clear → Send → Model A buttons → Model B buttons
- Enter key in question field: Trigger send action
- Ctrl+A in text areas: Select all
- Ctrl+C: Copy selected text

**Visual Accessibility**:

- High contrast mode compatible
- Resizable text areas and windows
- Clear focus indicators on interactive elements
- Progress bar with both visual and textual feedback

**Error Prevention**:

- Disable send button during processing
- Validation before submission (question + image required)
- Confirmation dialogs for destructive actions (future: clear all results)

# 8 Requirement Traceability Matrix

## 8.1 Purpose of the RTM

The Requirements Traceability Matrix (RTM) serves as a critical project management tool that establishes and maintains bidirectional traceability between requirements, design elements, and implementation components throughout the project lifecycle.

**Primary Objectives**:

1. **Completeness Verification**: Ensure all requirements from the Requirements Specification Document (RSD) have corresponding design and implementation
2. **Impact Analysis**: Enable rapid assessment of how design or implementation changes affect requirements
3. **Test Coverage**: Facilitate test plan development by mapping requirements to test cases
4. **Project Progress Tracking**: Provide visibility into implementation status against requirements
5. **Compliance Documentation**: Demonstrate full requirements coverage for academic review
6. **Change Management**: Support controlled evolution of requirements with traceable modifications

**Benefits to Stakeholders**:

- **Development Team**: Clear understanding of what must be implemented and why
- **QA Team**: Structured basis for test case development and validation
- **Project Managers**: Visibility into project completeness and risk areas
- **Academic Reviewers**: Evidence of systematic design and implementation process

## 8.2 Matrix Structure

The RTM employs a multi-column structure linking requirements to design and implementation artifacts:

**Column Definitions**:

1. **Req ID**: Unique identifier from RSD (e.g., FR-1.1, NFR-2.3)
2. **Requirement Description**: Brief statement of the requirement
3. **Priority**: Critical / High / Medium / Low
4. **Design Component**: Reference to SDS section/component implementing requirement
5. **Implementation File**: Source code file(s) realizing the requirement
6. **Test Case ID**: Reference to test case validating the requirement
7. **Status**: Not Started / In Progress / Completed / Verified
8. **Notes**: Additional context, dependencies, or issues

## 8.3 Requirements Tracking Process

### 8.3.1 Traceability Workflow

1. **Requirement Capture Phase**:
   - Requirements documented in RSD
   - Each requirement assigned unique ID
   - Requirements reviewed and approved by stakeholders
2. **Design Phase**:

- RTM populated with requirement IDs from RSD
- Design components mapped to requirements in SDS
- RTM updated with design component references
- Design reviews verify all requirements addressed

3. **Implementation Phase**:
   - Developers reference RTM to understand requirements
   - Implementation files recorded in RTM
   - Status updated as components are completed
   - Code reviews verify requirement fulfillment
4. **Testing Phase**:
   - Test cases developed based on RTM
   - Test case IDs added to RTM
   - Test execution verifies requirements
   - Status updated to "Verified" upon successful testing
5. **Maintenance Phase**:
   - RTM consulted for impact analysis of changes
   - New requirements added with full traceability
   - Deprecated requirements marked and archived

### 8.3.2 Change Control Process

When requirements change:

1. Change request documented with rationale
2. RTM reviewed to identify affected design/implementation
3. Impact assessment performed
4. Changes approved by project stakeholders
5. RTM updated with new/modified requirements
6. Affected components updated and re-tested
7. Document version control maintained

## 8.4 RTM Entries

**Functional Requirements Traceability**:

| Req ID | Description | Pri | Design Component | Implementation | Status |
|---|---|---|---|---|---|
| FR-1.1 | Support configurable endpoint URLs for two models | Crit | §5.3 GUI Design, QLineEdit widgets | Application.py: lines 45-46, model1_url_input, model2_url_input | Complete |
| FR-1.2 | Send identical data to both endpoints simultaneously | Crit | §5.2 ModelWorker threading | Application.py: lines 180-195, on_send() method | Complete |
| FR-2.1 | Accept PNG, JPG, JPEG, BMP image formats | Crit | §5.3 File dialog filter | Application.py: line 128, on_attach() | Complete |

| Req ID | Description | Pri | Design Component | Implementation | Status |
|---|---|---|---|---|---|
| FR-3.4 | Implement 8+ prompting strategies | High | §4.2 PromptingStrategies class | server.py: lines 150-350, all strategy methods | Complete |
| FR-4.1 | Extract tabular data using Chart2Table | High | §4.2 ChartExtractor | PaddlePaddle Server/server.py: ChartExtractor class | Complete |
| FR-4.3 | Clean extracted data | Crit | §4.2 clean_chart2text_output() | server.py: lines 95-108 | Complete |
| FR-5.1 | Remove escape sequences from responses | Med | §5.3 Response processing | Application.py: lines 210-215, on_worker_finished() | Complete |
| FR-6.2 | Evaluate with 5% tolerance | Crit | §4.2 evaluate_answer() | deplot_qwen.py: lines 450-550 | Complete |

**Non-Functional Requirements Traceability**:

| Req ID | Description | Pri | Design Component | Implementation | Status |
|---|---|---|---|---|---|
| NFR-1.1 | Process queries in <10 seconds | High | §3.1 GPU memory optimization | server.py: Config.GPU MEMORY FRACTION = 0.75 | Complete |
| NFR-1.4 | Non-blocking UI | Crit | §5.2 QThread workers | Application.py: ModelWorker class | Complete |
| NFR-2.3 | Handle GPU OOM gracefully | High | §4.3 Error handling | server.py: try-catch in generate_response() | Complete |
| NFR-2.4 | Request timeouts | High | §5.2 HTTP timeout | Application.py: line 30, timeout=60 | Complete |
| NFR-3.1 | Validate user inputs | High | §5.3 Input validation | Application.py: lines 168-174, on_send() validation | Complete |
| NFR-3.3 | Temporary file cleanup | Med | §4.1 File management | server.py: overwrite temp_upload.jpg | Complete |
| NFR-4.6 | Modular architecture | High | §3.2 Microservices | Separate server files for each component | Complete |
| NFR-4.11 | Intuitive GUI | High | §5.3 UI design | Application.py: entire GUI implementation | Complete |

# 9 Glossary of Terms

## 9.1 Model and AI Terminology

**Autoregressive Generation** A text generation approach where each token is predicted based on all previously generated tokens in sequence.

**Batch Size** The number of samples processed together in a single forward pass through a neural network. In this system, batch_size=1 for real-time inference.

**BFloat16 (Brain Floating Point 16)** A 16-bit floating point format developed by Google Brain, offering reduced memory usage while maintaining FP32 dynamic range.

**Chain-of-Models (CoM)** An architecture where multiple specialized models are sequentially applied, with the output of one model serving as enhanced input to the next.

**Chain-of-Thought (CoT)** A prompting technique that encourages language models to generate intermediate reasoning steps before producing a final answer, improving performance on complex tasks.

**Chart2Table** The process of converting visual chart representations into structured tabular data. Also refers to PaddlePaddle's PP-Chart2Table model.

**CUDA (Compute Unified Device Architecture)** NVIDIA's parallel computing platform and API model for GPU programming.

**DePlot** Google's Pix2Struct-based model that translates plot images into linearized table representations for downstream language model processing.

**Device Map** PyTorch's automatic device placement strategy that distributes model layers across available GPUs and CPU memory.

**Few-Shot Learning** A learning paradigm where models are provided with a small number of examples (typically 1-10) in the prompt to guide their behavior on new tasks.

**Inference** The process of using a trained model to make predictions on new, unseen data. Distinguished from training, which adjusts model parameters.

**KV Cache (Key-Value Cache)** A mechanism in transformer models that stores computed key and value tensors to avoid redundant calculations during autoregressive generation.

**Large Language Model (LLM)** Deep neural networks with billions of parameters trained on vast text corpora, capable of understanding and generating human-like text.

**Linearized Table** A text representation of tabular data where rows and columns are encoded as a sequence with special delimiter tokens (e.g., <0x0A> for newlines).

**Multimodal Model** An AI model trained to process and understand multiple types of data modalities (e.g., text, images, audio) simultaneously.

**Pix2Struct** A vision-language model architecture designed for structured visual understanding tasks, including chart and table extraction.

**Prompting Strategy** A systematic approach to formulating inputs to language models to elicit desired outputs, including techniques like zero-shot, few-shot, and chain-of-thought.

**Qwen2.5-VL** Alibaba Cloud's vision-language model family, capable of processing both images and text. The 7B-Instruct variant has 7 billion parameters optimized for instruction following.

**Temperature** A hyperparameter controlling randomness in text generation. Lower values (0.1-0.3) produce more deterministic outputs, higher values (0.7-1.0) increase diversity.

**TF32 (TensorFloat-32)** NVIDIA's precision format for Ampere+ GPUs that balances FP32 range with improved performance, automatically used in matrix multiplications.

**Token** The basic unit of text processing in language models. Can represent words, subwords, or characters depending on the tokenization scheme.

**Top-p Sampling (Nucleus Sampling)** A generation strategy that samples from the smallest set of tokens whose cumulative probability exceeds threshold p.

**Vision-Language Model (VLM)** A type of multimodal model specifically designed to understand relationships between visual and textual information.

**VRAM (Video Random Access Memory)** GPU memory used to store model weights, activations, and intermediate computations during inference.

**Zero-Shot Learning** The ability of models to perform tasks without any task-specific training examples, relying only on instructions in the prompt.

## 9.2 Software Engineering Terminology

**API (Application Programming Interface)** A set of protocols, tools, and definitions for building software applications and enabling system communication.

**ASGI (Asynchronous Server Gateway Interface)** A Python standard for asynchronous web servers and applications, used by FastAPI and Uvicorn.

**Base64 Encoding** A binary-to-text encoding scheme that represents binary data in ASCII string format, used for transmitting images over JSON APIs.

**Endpoint** A specific URL path in a REST API that accepts requests and returns responses (e.g., /predict, /extract-base64).

**FastAPI** A modern Python web framework for building REST APIs with automatic documentation, type validation, and async support.

**Microservice Architecture** A software design pattern where applications are composed of small, independent services that communicate via APIs.

**Multipart Form-Data** An HTTP content type used for uploading files along with text data, encoding each part with a boundary delimiter.

**PyQt6** Python bindings for the Qt application framework, used for building cross-platform desktop GUI applications.

**QThread** PyQt's threading class that enables concurrent operations without blocking the GUI event loop.

**REST (Representational State Transfer)** An architectural style for distributed systems based on stateless client-server communication over HTTP.

**RTM (Requirements Traceability Matrix)** A document linking requirements to design components, implementation files, and test cases for project tracking.

**Signal-Slot Mechanism** Qt's inter-object communication pattern where signals emitted by one object trigger slots (methods) in connected objects.

**Stateless Service** A service design where each request contains all information needed to process it, with no persistent session state on the server.

**Uvicorn** A high-performance ASGI server for Python web applications, commonly used with FastAPI.

## 9.3 Data and Evaluation Terminology

**Benchmark Dataset** A standardized dataset used to evaluate and compare model performance across different approaches.

**ChartQA** A benchmark dataset containing chart images paired with questions and answers for evaluating visual question answering systems.

**Ground Truth** The correct, verified answer used as a reference for evaluating model predictions.

**PlotQA** A dataset focused on question answering over plot and graph images, containing synthetic charts with structured data.

**Relative Tolerance** A percentage-based margin of error allowed when comparing numerical predictions to ground truth (e.g., 5% = 0.05).

**Test Split** A subset of a dataset reserved for final model evaluation, kept separate from training and validation data.

## 9.4 System and Network Terminology

**GPU (Graphics Processing Unit)** Specialized hardware accelerator for parallel computations, essential for deep learning inference.

**HTTP/1.1** The Hypertext Transfer Protocol version used for client-server communication in this system.

**Localhost** A hostname referring to the current machine (IP address 127.0.0.1), used for local API communication.

**Port** A numerical identifier for network endpoints (e.g., 8000, 8001) allowing multiple services on one IP address.

**POST Request** An HTTP method for sending data to servers, used for image uploads and question submissions.

**RTT (Round-Trip Time)** The duration for a network packet to travel from source to destination and back, affecting perceived latency.

**Timeout** Maximum duration to wait for a network operation before considering it failed (default 60 seconds in this system).

## 9.5 Framework-Specific Terminology

**PaddlePaddle** Baidu's open-source deep learning platform, used for the Chart2Table extraction model.

**PaddleOCR** A family of OCR tools from PaddlePaddle, including document structure recognition capabilities.

**PP-StructureV3** PaddlePaddle's document analysis system supporting layout detection and table recognition.

**PyTorch** Facebook's open-source deep learning framework, used for Qwen model inference in this project.

**Transformers Library** Hugging Face's library providing pre-trained models and utilities for NLP and vision-language tasks.

**qwen-vl-utils** A utility package for processing vision inputs in Qwen vision-language models.

## 9.6 Project-Specific Terminology

**Base Model Server** The deployment configuration running only Qwen2.5-VL without Chart2Table preprocessing.

**Pipelined Model Server** The deployment configuration integrating Chart2Table extraction before Qwen inference.

**Dual Comparison Mode** The GUI feature allowing simultaneous queries to two model endpoints for side-by-side evaluation.

**Strategy** A specific prompting approach implemented in the system (e.g., baseline, zero_shot_cot, chart2table_cot).

**Hybrid Strategy** A prompting approach combining multiple techniques, such as chart2table_cot (Chain-of-Models + Chain-of-Thought).

# 10 Appendices

## 10.1 Appendix A: Configuration Templates

### 10.1.1 Server Configuration Template

```
# server_config.py
class Config:
    # ============== GPU Settings ==============
    GPU_MEMORY_FRACTION = 0.75  # Adjust based on GPU size
    ENABLE_TF32 = True  # Set False for pre-Ampere GPUs
    DEVICE = "cuda"  # or "cpu" for CPU-only inference

    # ============== Model Settings ==============
    LLM_MODEL = "Qwen/Qwen2.5-VL-7B-Instruct"
    MAX_INPUT_TOKENS = 4096
    MAX_NEW_TOKENS = 512

    # ============== API Settings ==============
    HOST = "0.0.0.0"  # Listen on all interfaces
    PORT = 8001  # Change for multiple servers
    PADDLE_API_URL = "http://localhost:8000"
    PADDLE_API_TIMEOUT = 30

    # ============== Inference Settings ==============
    DEFAULT_TEMPERATURE = 0.1
    DEFAULT_TOP_P = 0.9

    # ============== Debug Settings ==============
    VERBOSE_DEBUG = False
```

### 10.1.2 Client Configuration Template

```
# client_config.py
class ClientConfig:
    # Default endpoint URLs
    MODEL_A_URL = "http://10.40.32.8:8001/predict"
    MODEL_B_URL = "http://10.40.32.12:8001/predict"

    # Request settings
    REQUEST_TIMEOUT = 60  # seconds

    # UI settings
    IMAGE_PREVIEW_SIZE = (360, 270)  # width x height
    WINDOW_SIZE = (1000, 650)
```

## 10.2 Appendix B: Deployment Checklist

**Pre-Deployment**:

1. Verify GPU availability (`nvidia-smi`)
2. Install CUDA 11.8+ and drivers
3. Create virtual environments (PyTorch, PaddlePaddle)
4. Download model weights from HuggingFace
5. Configure firewall rules for ports 8000, 8001
6. Update endpoint URLs in client configuration

**Server Startup**:

1. Start PaddlePaddle server: `uvicorn paddle_api_server:app -host 0.0.0.0 -port 8000`
2. Verify PaddlePaddle health: `curl http://localhost:8000/health`
3. Start Pipelined server: `python server.py` (waits for PaddlePaddle)
4. Start Base server on separate machine: `python server.py`
5. Verify model loading (check console output)

**Client Launch**:

1. Launch GUI: `python Application.py`
2. Verify endpoint connectivity (test with sample image)
3. Check response formatting and timing

## 10.3 Appendix C: Troubleshooting Guide

**Common Issues and Solutions**:

| **Issue** | **Cause** | **Solution** |
|---|---|---|
| CUDA Out of Memory | GPU memory exceeded | Reduce GPU MEMORY FRACTION, close other GPU processes |
| Model loading hangs | Network/download issue | Check internet, use cached models |
| PaddlePaddle import error | Wrong environment | Activate PaddlePaddle conda env |
| Connection refused | Server not running | Start servers in correct order (Paddle first) |
| Timeout errors | Network latency high | Increase timeout value, check network |
| Image upload fails | Invalid format | Verify PNG/JPG/JPEG format |
| GUI freezes | Blocking operation | Check ModelWorker threading implementation |
| Escape sequences in output | Display formatting issue | Implemented in v1.0, update client |
| Chart2Table extraction empty | Model initialization failed | Check PaddleX installation, verify GPU access |

## 10.4 Appendix D: Code Style Guidelines

**Python Conventions (PEP 8 Compliant)**:

- **Indentation**: 4 spaces (no tabs)
- **Line Length**: Max 100 characters (except URLs/imports)
- **Naming**:
  - Classes: PascalCase (e.g., `ChartAnalyzer`)
  - Functions/methods: snake_case (e.g., `generate_response`)
  - Constants: UPPER_SNAKE_CASE (e.g., `MAX_NEW_TOKENS`)
  - Private members: _leading_underscore
- **Imports**: Grouped (standard library, third-party, local) with blank lines
- **Docstrings**: Google style for all public functions/classes

- **Type Hints**: Encouraged for function signatures

**Example Docstring**:

```
def generate_response(
    self,
    image_path: str,
    prompt: str,
    max_new_tokens: int = 512
) -> str:
    """
␣␣␣␣Generate␣response␣for␣chart␣question␣using␣Qwen2.5-VL.

␣␣␣␣Args:
␣␣␣␣␣␣␣␣image_path:␣Path␣to␣chart␣image␣file
␣␣␣␣␣␣␣␣prompt:␣Text␣prompt␣including␣question␣and␣strategy
␣␣␣␣␣␣␣␣max_new_tokens:␣Maximum␣tokens␣to␣generate␣(default␣512)

␣␣␣␣Returns:
␣␣␣␣␣␣␣␣Generated␣text␣response␣from␣the␣model

␣␣␣␣Raises:
␣␣␣␣␣␣␣␣FileNotFoundError:␣If␣image_path␣does␣not␣exist
␣␣␣␣␣␣␣␣torch.cuda.OutOfMemoryError:␣If␣GPU␣memory␣exhausted
␣␣␣␣"""
```

## 10.5 Appendix E: Version Control Strategy

**Repository Structure**:

```
chart-understanding-system/
 client/
    Application.py
 servers/
    base_model/
       server.py
    pipelined_model/
       server.py
    paddlepaddle_server/
        server.py
 evaluation/
    deplot_qwen.py
    benchmark.json
 docs/
    SDS.pdf
    RSD.pdf
    API_Documentation.md
 tests/
    test_strategies.py
 requirements.txt
 requirements_paddle.txt
 README.md
```

## 10.6 Appendix F: References and Further Reading

**Key Research Papers**:

1. Wei, J., et al. (2022). "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models." *NeurIPS 2022*.

2. Liu, F., et al. (2023). "DePlot: One-shot visual language reasoning by plot-to-table translation." *ACL 2023*.

3. Masry, A., et al. (2022). "ChartQA: A Benchmark for Question Answering about Charts with Visual and Logical Reasoning." *ACL 2022*.

4. Methani, N., et al. (2020). "PlotQA: Reasoning over Scientific Plots." *WACV 2020*.

5. Bai, J., et al. (2023). "Qwen Technical Report." *arXiv:2309.16609*.

**Technical Documentation**:

1. Hugging Face Transformers: https://huggingface.co/docs/transformers

2. PaddlePaddle: https://www.paddlepaddle.org.cn/en

3. PyQt6 Documentation: https://www.riverbankcomputing.com/static/Docs/PyQt6/

4. FastAPI User Guide: https://fastapi.tiangolo.com/