# System Design Document (SDD)  
  
## 1. Introduction  
### 1.1 Purpose  
The purpose of this document is to provide a comprehensive technical specification for the development of a web-based calculator application for Tapestry. This document outlines the system architecture, design components, integration points, technical requirements, and implementation details to ensure alignment with the project requirements and stakeholder needs as outlined in the Business Requirements Document (BRD).  
  
### 1.2 Scope  
The scope of this system includes:  
- Development of a web-based calculator application.  
- Implementation of basic arithmetic operations (addition, subtraction, multiplication, and division).  
- Error handling for edge cases like division by zero.  
- Deployment on Azure with guidance for Tapestry's internal team.  
- Exposing the application via a REST API.  
  
The project will not cover:  
- Advanced mathematical functions beyond basic arithmetic.  
- Complex UI/UX design elements.  
- Long-term maintenance and support beyond initial deployment.  
  
### 1.3 Objectives  
The key objectives of the system are:  
- Develop a web application that performs addition, subtraction, multiplication, and division.  
- Ensure the application handles edge cases such as division by zero.  
- Provide a simple and clean user interface.  
- Deliver the application as a REST API for ease of integration.  
- Deploy the application on Azure with guidance provided to Tapestry's internal team.  
  
## 2. System Architecture  
### 2.1 High-Level Design  
The system architecture consists of a Flask-based web application deployed on Azure. The application will expose REST API endpoints for arithmetic operations and handle user interactions through a simple web interface.  
  
### 2.2 Major Components  
- \*\*Flask Application\*\*: The core component that handles HTTP requests and performs arithmetic operations.  
- \*\*WSGI Server\*\*: Serves the Flask application and manages incoming requests.  
- \*\*Azure Services\*\*: Provides the infrastructure for hosting, scaling, and managing the application.  
  
### 2.3 Architecture Style  
The chosen architecture style is a client-server model, where the client interacts with the server via REST API endpoints. This style is justified as it allows for easy integration with other systems and provides a clear separation between the client and server components.  
  
## 3. Design Constraints  
### 3.1 Technical Constraints  
- The application will be developed using Python and Flask.  
- The application will be deployed on Azure.  
- Tapestry's internal team has some familiarity with Flask for future maintenance.  
  
### 3.2 Performance Constraints  
- The application should respond to API requests within a reasonable time frame.  
- The system should handle a growing number of users without significant degradation in performance.  
  
### 3.3 Other Constraints  
- The project must be completed within 6 weeks from the date of contract signing.  
- The budget for the project will be determined based on the proposals received.  
  
## 4. System Components  
### 4.1 Component 1: Flask Application  
- \*\*Description\*\*: Handles HTTP requests, performs arithmetic operations, and returns results.  
- \*\*Interfaces\*\*: Exposes REST API endpoints for addition, subtraction, multiplication, and division.  
- \*\*Input/Output Specifications\*\*: Accepts numerical inputs via API requests and returns results in JSON format.  
  
### 4.2 Component 2: WSGI Server  
- \*\*Description\*\*: Serves the Flask application and manages incoming HTTP requests.  
- \*\*Interfaces\*\*: Interfaces with the Flask application and the Azure infrastructure.  
- \*\*Input/Output Specifications\*\*: Receives HTTP requests and forwards them to the Flask application.  
  
## 5. Data Design  
### 5.1 Database Schema  
No database is required for the initial version of the application. Future iterations may include database integration for additional functionalities.  
  
### 5.2 Data Flow  
Data flows from the client to the server via HTTP requests. The server processes the requests and returns the results to the client.  
  
### 5.3 Data Storage and Retrieval  
No data storage is required for the initial version of the application. Future iterations may include data storage for additional functionalities.  
  
## 6. Security Considerations  
### 6.1 Security Architecture  
- \*\*Authentication\*\*: Not required for the initial version.  
- \*\*Authorization\*\*: Not required for the initial version.  
- \*\*Encryption\*\*: Use HTTPS for secure communication.  
  
### 6.2 Data Privacy and Protection  
- Follow best practices in web security, including input validation and sanitization.  
- Ensure compliance with relevant data protection regulations.  
  
## 7. User Interface Design  
### 7.1 Wireframes/Mockups  
The user interface will be simple and intuitive, allowing users to perform basic arithmetic operations easily.  
  
### 7.2 User Interaction and Navigation  
Users will interact with the application through a web interface or via API requests. The interface will provide input fields for numerical values and buttons for arithmetic operations.  
  
## 8. Testing and Validation  
### 8.1 Testing Strategies  
- \*\*Unit Testing\*\*: Use pytest for testing individual functions.  
- \*\*Integration Testing\*\*: Use Postman or similar tools for testing API endpoints.  
- \*\*System Testing\*\*: Perform end-to-end testing to ensure the application meets all requirements.  
  
### 8.2 Validation  
Validate the system design against the business requirements outlined in the BRD. Ensure that all functional and non-functional requirements are met.  
  
## 9. Deployment Considerations  
### 9.1 Deployment Environment  
- \*\*Hardware\*\*: Azure infrastructure.  
- \*\*Software\*\*: Python, Flask, WSGI server, Docker.  
- \*\*Network\*\*: Secure and reliable network connection.  
  
### 9.2 Backup and Recovery  
Implement backup and recovery strategies to ensure data integrity and availability.  
  
### 9.3 Failover Strategies  
Use Azure's auto-scaling and failover features to ensure system availability in case of failures.  
  
## 10. Appendices  
### 10.1 Glossary of Terms  
- \*\*API\*\*: Application Programming Interface  
- \*\*BRD\*\*: Business Requirements Document  
- \*\*Flask\*\*: A micro web framework written in Python  
- \*\*JSON\*\*: JavaScript Object Notation  
- \*\*REST\*\*: Representational State Transfer  
- \*\*WSGI\*\*: Web Server Gateway Interface  
  
### 10.2 References  
- Business Requirements Document (BRD)  
- Meeting Transcript  
- SDD Template  
  
This SDD accurately captures the technical solution, architecture, and implementation details, aligning with the project requirements and stakeholder needs as outlined in the BRD.