System Design to Calculator CLI App.

Core Components of System Design:

1. Requirements
    - Functional
    - Non-functional
2. High-Level Design (HLD)
    - Architecture diagram
    - Main components/services
    - Data flow
3. Low-Level Design (LLD)
    - Classes, functions, APIs
    - Database schema (if any)
    - Detailed logic
4. Database Design
    - Schema or collections
    - Relationships or indexing
5. APIs (if needed)
    - Endpoints
    - Request/Response structure
6. Scalability
    - Load balancing
    - Caching
    - Horizontal scaling
7. Fault Tolerance
    - Backup
    - Retry mechanisms
8. Security
    - Input validation
    - Authentication / Authorization
9. Monitoring & Logging
    - Logs
    - Alerts / dashboards
10. Future Scope
    - Extensibility
    - Feature additions

Requirements

1. Functional Requirements

1. Basic Operations: Support addition, subtraction, multiplication, division.
2. Input Parsing: Read user input in the form operand1 operator operand2 (e.g. 5 * 3).
3. Result Display: Print the result to the console.
4. Command Loop: Allow multiple calculations until the user types an exit command (e.g. quit or exit).
5. Error Handling:

- Handle invalid operators (e.g. %).
- Detect non-numeric inputs.
- Prevent division by zero.

2. Non-Functional Requirements

1. Usability: Clear prompts and error messages.
2. Maintainability: Modular code with separate functions for each operation.
3. Reliability: Correct calculations under all valid inputs.
4. Performance: Instant response for any single operation.
5. Portability: Runs on any system with Python 3.x installed.

High-Level Design (HLD):

1. User Interface (CLI)

- Accepts user input via command line
- Displays results or errors
- Input formats can grow (e.g., sin(30) or bin(10) in future)

2. Main Controller

- Handles loop
- Parses user input
- Calls appropriate function (basic now, expandable later)

3. Calculator Engine

- Phase 1: Basic operations (+, –, *, /)
- Phase 2 (Future Scope):
- Scientific (sin, cos, tan, log, sqrt)
- Programmable (binary, hex, conversions, etc.)

4. Function Modules (optional in future)

- Split logic into separate files when expanding

- basic.py
- scientific.py
- programmable.py

5. File Structure (current and future-ready)

```
calculator.py               # entry
basic_operations.py         # basic arithmetic logic
scientific_operations.py    #  trig/log functions -> Future scope of the app
programmable_functions.py   #  binary, hex, etc. -> Future scope of the
```

6. Flow

```
[ User Input ]
    ↓
[ main() -> parse input ]
    ↓
[ calculate() -> select module ]
    ↓
[ Run function -> return result ]
    ↓
[ Show output or error ]
    ↓
[ Repeat or exit ]
```

Low-Level Design (LLD)

Perfect bro, here is the Low-Level Design (LLD) explained as theory only — no code, just clear structure.

Low-Level Design (LLD)

The LLD breaks your calculator system into individual components and responsibilities at a detailed level.

1. Input Handler
    - Reads the expression from the user through the terminal.
    - Validates that the input format is correct (e.g., number operator number).
    - In future, should handle functions like sin(30) or bin(5) too.

2. Operation Dispatcher
    - Analyzes the operator from the input (+, -, *, /, etc.).
    - Maps it to the corresponding logic function.
    - Acts like a controller deciding where the request should go.

3. Arithmetic Logic Module
    - Contains basic arithmetic functions (add, subtract, multiply, divide).
    - Each operation is separated into its own function for modularity.
    - In future, more logic will be added for scientific or programmable operations.

4. Result Handler
    - Takes the output of the operation and prepares it for display.
    - Handles formatting, rounding (if needed), or error messages.
    - Ensures a clean and consistent output to the user.

5. Error Handler
    - Catches common issues like:
    - Invalid input format
    - Unsupported operators
    - Division by zero
    - Shows user-friendly messages without crashing the program.

6. History Logger (Optional/Advanced)
    - After each successful calculation, saves the input and result to a local database (like SQLite).
    - Useful for later showing a history of calculations or debugging.
    - Will interface with the database layer.

7. Function Module Loader (Future Scope)
    - If the user adds a custom function (e.g., double(x) = x * 2), this component will load and evaluate it.
    - Helps make the calculator programmable.

Database Structure Design

The goal is to support:

- Logging calculation history
- Storing custom user-defined functions
- Keeping the system extensible for future scientific/programmable operations

Database Type

- For CLI app: SQLite (lightweight, local file-based)
- For web app/API (later): Can switch to PostgreSQL or MongoDB

Database Structure Overview

calculator.db

```
|
├── calculation_history
|     ├── id
|     ├── expression
|     ├── result
|     └── timestamp
|
└── user_functions
    ├── id
    ├── name
    ├── formula
    └── created_at
```

API Desig

This is needed if you plan to:

- Turn your calculator into a web service
- Connect it to a frontend UI
- Expose calculation, history, and programmable functions through HTTP endpoints

You can build this using FastAPI or Flask.

1. Define Purpose
2. Choose API Type (REST)

3. Decide HTTP Methods
4. Design Endpoints
5. Define Request and Response Models
6. Set Up Routing Structure
7. Connect to Database
8. Handle Errors and Validation
9. Organize Folder Structure
10. Plan for Authentication (if needed in future)
11. Add Documentation (Swagger/OpenAPI)
12. Test with Postman or curl

Folder Structure for API Project

```
calculator_api/
├── main.py
├── models.py
├── db.py
├── routes/
│   ├── calculate.py
│   ├── history.py
│   └── functions.py
└── utils/
    └── parser.py
```

Scalability Steps

1. Identify Load Points

Analyze which functions get used heavily — like continuous calculations or frequent history requests. Use logs or metrics to find hotspots where performance may degrade under traffic.

2. Separate Concerns

Split responsibilities clearly:

- Frontend (CLI now, later web/GUI) handles input/output
- Backend performs logic and talks to DB
- Database stores history/functions

This modularity allows independent upgrades, scaling, and debugging.

3. Use Modular Codebase

Break logic into multiple files based on role:

- basic_operations.py for +, -, *, /
- scientific.py for sin, log, sqrt
- db.py for database interactions

Makes code reusable, testable, and future-proof.

4. Enable Horizontal Scaling

When deployed, run multiple copies of the backend service on cloud. A load balancer sends requests to free instances. Helps handle more users without slowing down.

5. Use Caching

If users repeatedly calculate the same expression, store result in memory (like using a Python dict or Redis). Next time, serve instantly without recalculation.

6. Optimize Database Queries

Prevent performance issues when the database grows:

- Add indexes on timestamp or expression
- Use LIMIT when fetching recent history
- Avoid full scans for every request

7. Add Queue System

If you introduce heavy or async tasks (e.g., resume parsing, large function evaluation), push them to a background worker (e.g., Celery). Keeps the app responsive and scalable.

8. Use Load Balancer

Put a load balancer (like Nginx or cloud provider's) in front of your backend instances. It:

- Spreads requests evenly
- Avoids server overload
- Improves fault tolerance

9. Use Cloud Deployment

Host:

- Frontend on Netlify or Vercel
- Backend API on Render, Railway
- DB on Railway or hosted PostgreSQL

Scales globally, needs no manual infra setup, and offers autoscaling.

10. Monitor Performance Metrics

Use tools (or simple logging at first) to track:

- Response time of APIs
- Number of users/calculations
- Errors like divide by zero

This helps detect bottlenecks before they affect users.

Security

1. Input validation
   Ensure all user inputs (via CLI or API) are checked before processing — e.g., prevent dividing by zero, invalid characters, or unsupported functions.

2. Exception handling
   Wrap critical logic in try-except blocks so that unexpected errors (like float conversion, DB connection issues) don't crash the app.

3. Retry logic for failed operations
   If an operation fails (like saving history to DB), try again a few times before giving up — helps in cases of temporary issues.

4. Graceful fallback responses
   Instead of crashing or printing raw errors, return user-friendly messages like "Invalid expression" or "Calculation failed. Try again."

5. Logging errors
   Log all failures with details (error message, timestamp, input) to a log file or DB for debugging and monitoring.

6. Isolate failures (module-level handling)
   Make sure a bug in one module (e.g., scientific mode) doesn't affect others (like basic calculator). Isolate logic in try blocks or services.

7. Backup data (for history or user functions)
   Regularly back up user calculation history and saved functions, especially if stored in a file or DB — prevents total loss.

8. Avoid crashes on invalid input
   Even if the user enters weird/empty input, the app should not break — catch it early and handle it cleanly.

9. Use default values where needed
   If configs or optional data is missing (like mode = None), fall back to sensible defaults to keep the app running.

10. Monitor and alert on failures
    Set up a basic alert/logging system to notify (console, file, email) when repeated failures or exceptions occur — helps you act fast.

1. Input Sanitization
   Clean all user inputs to prevent injection attacks — especially important when moving from CLI to web or API (e.g., prevent code execution or DB injection).

2. Secure Data Storage
   If you store user history or saved functions, encrypt sensitive fields or use hashed formats to prevent misuse if the DB is exposed

.

3. Authentication & Authorization
   When the app supports multiple users (via API), secure it with user login (token/session-based). Only allow access to each user's own data.

4. Avoid Code Injection
   Never use eval() directly on user inputs. Use parsers or safe evaluators (e.g., ast.literal_eval or a math expression parser) to prevent malicious code execution.

5. Rate Limiting
   Limit how many times a user or IP can call your API/calculator in a given time window — prevents abuse or DDoS attacks.

6. Use HTTPS in API/Web
   If deployed, use HTTPS to encrypt data in transit — especially login credentials and sensitive calculations.

7. Role-Based Access (Future)
   If you expand roles (admin/user), ensure only admins can view all history, modify config, etc. Use proper permission checks.

8. Secure API Keys/Secrets
   If you integrate third-party services (e.g., logging, cloud DB), store API keys securely using environment variables — never hardcode them.

9. Update Dependencies
   Keep libraries/frameworks updated (like Flask, FastAPI, or math libs) to avoid known security vulnerabilities.

10. Log and Alert on Suspicious Activity
    Monitor and log things like too many failed login attempts, malformed inputs, or repeated heavy usage — helps detect potential attacks early.

Observability / Monitoring

1. Logging
   Log all important events like calculations, errors, user actions, DB failures — helps track what's happening in the system.

2. Structured Logs
   Format logs in JSON or key-value pairs so they're easy to parse and search later using tools (like ELK, Datadog).

3. Error Tracking
   Use tools like Sentry (or custom error handlers) to collect and notify you of exceptions in real time, with stack traces and context.

4. Health Checks
   Create a /health endpoint (for web/API) that returns the app's status (up/down) so you can monitor uptime.

5. Uptime Monitoring
   Use external tools like UptimeRobot or Pingdom to get alerts if your service goes offline.

6. Performance Metrics
   Track app performance — response time, CPU usage, memory consumption, etc., using Prometheus, Grafana, or lightweight CLI logs.

7. Alerting System
   Set thresholds (e.g., error rate > 5%, response time > 1s) and trigger alerts (email, Slack, etc.) when crossed.

8. Request Tracing
   For APIs, assign a unique ID to each request and log it across services to trace end-to-end execution and debug issues faster.

9. Analytics
   Track usage patterns — which features (basic, scientific) are used most, what times users are most active, etc., to plan future improvements.

10. Storage Monitoring
    Track storage size (especially DB/file logs) so it doesn't grow endlessly — setup rotation or cleanup jobs.

DevOps / Deployment Strategy

1. Version Control (Git)
   Use Git to track code changes. Keep the project in a repo (GitHub/GitLab) for collaboration, history, and CI/CD.

2. CI/CD Pipelines
   Set up pipelines (e.g., GitHub Actions) to auto-run tests, lint checks, and deploy on push — saves time and avoids manual errors.

3. Dockerize the App
   Use Docker to containerize your calculator backend (if it becomes an API). This ensures consistency across all environments.

4. Use Environment Variables
   Store all secrets, keys, configs (like DB URL, mode) in .env files or secret managers — keeps the code clean and secure.

5. Infrastructure as Code (IaC)
   If deploying complex infra (DB, backend, Redis), use tools like Terraform or Pulumi to define and manage infra via code.

6. Cloud Deployment Platforms
   Use platforms like Render, Railway, Vercel, Heroku, or AWS to host your app depending on the stack (CLI to API/web).

7. Rollback Strategy

Have versioned deployments so if a bug is introduced in production, you can quickly roll back to the previous working version.

8. Zero Downtime Deployment
   Use tools or strategies (like blue-green deployment) to deploy updates without interrupting active users.

9. Automated Testing in CI
   Include unit and integration tests in your CI process to ensure code quality before deployment.

10. Monitoring in DevOps
    Integrate observability tools (from previous section) into your CI/CD and infra for automated alerts and performance insights.

This system design document outlines the high-level and low-level planning, scalability, fault tolerance, deployment, and API design strategies for building a modular and extensible calculator CLI application. Starting from basic operations, the system is structured to support future enhancements like scientific and programmable functions. The design ensures that the application is maintainable, scalable, and production-ready when expanded into a full-stack or cloud-hosted solution.