Architecture:

- Frontend (React) communicates with the Backend (ASP.NET Core API).
- Frontend requests data from the Backend using RESTful APIs.
- Backend retrieves data from a JSON file or a database and provides it to the Frontend.
- Frontend renders data and user interfaces for practitioners, reports, and appointments.

Components:

Frontend:

- PractitionerList: Display two separate lists of practitioners (supervisors and others).
- FinancialReport: Display generated financial reports and allow clicking on report items.
- AppointmentList: Display breakdown data based on the clicked report item.
- AppointmentDetails: Display detailed information about a selected appointment.
- DatePickers: Allow selecting the date range.

Backend:

- API Controllers: Handle routes for practitioners, reports, and appointments.
- Service Layer: Perform business logic and data retrieval operations.
- Models: Define data structures for practitioners, reports, and appointments.

Deployment, Uptime, Security, and Maintenance:

Deployment:

- Deploy the frontend and backend applications as separate containers using Docker.
- Use an orchestration tool like Docker Compose or Kubernetes for managing multiple containers.
- Deploy containers to a cloud platform or on-premises infrastructure.

Uptime and Security:

- Use HTTPS for secure communication between the frontend and backend.
- Implement authentication and authorization mechanisms to restrict access to authorized users.
- Implement rate limiting and validation to prevent malicious requests.

- Use environment variables to manage sensitive configurations securely.

Maintenance:

- Implement logging and monitoring for tracking errors and performance issues.
- Regularly update libraries, dependencies, and frameworks to ensure security and stability.
- Perform automated backups of data and configurations.
- Implement a CI/CD pipeline for automated testing and deployment.

Future Requirements and Expandability:

- Design the backend with a modular structure to accommodate future functionalities.
- Implement versioning for APIs to ensure backward compatibility.
- Use design patterns like Dependency Injection for easy integration of new services.
- Consider implementing a microservices architecture if the application grows significantly.
- Use a flexible database schema that can adapt to future data requirements.

Testing Strategies:

- Implement unit tests, integration tests, and end-to-end tests for both frontend and backend components.
- Use testing frameworks like Jest for frontend and xUnit for backend testing.
- Perform mock testing for APIs to simulate responses and test different scenarios.
- Automate testing using CI/CD pipelines to ensure code quality and prevent regressions.

Docker Support:

- Create Dockerfiles for both frontend and backend.
- Build Docker images for each component.
- Use Docker Compose to manage the orchestration of multiple containers.
- Define environment variables for configurations in the Docker Compose file.

Summary:

The proposed solution involves a modular architecture, separation of frontend and backend, secure communication, deployment using Docker, testing strategies, and considerations for future scalability. This design approach ensures flexibility, maintainability, and readiness for future requirements.