Delta Cyber Security Limited

Our Case Studies

Delta Cyber Security Limited
J-12, Kazi Nazrul Islam Road, Mohammadpur, Dhaka- 1206,
Bangladesh
Tel: +8801765-403041
Email: info@deltacsl.com
Website: www.deltacsl.com

Submitted by: Delta Cyber Security Limited, Bangladesh

Case Study: IoT Based Home Control System from Conventional Limitations to Smart Architecture

Overview

Traditional home automation systems often rely on rigid architectures that require frequent updates, lack modular scalability, and fail to integrate seamlessly with evolving devices and user needs. Our goal was to design a **next generation IoT based Home Control Solution** using a *System Oriented Layered Development (SOLD)* model — a structured framework focused on stability, optimization, layering, and dynamic adaptability.

Objective: Replace the conventional single stack IoT control model with a modular, intelligent architecture capable of handling real time data, flexible integrations, and low latency user control.

Role: System Architect responsible for designing the overall system structure, data flow, and communication logic between IoT devices, backend services, and user interfaces.

Understanding the Problem

Conventional IoT home systems were facing:

- Frequent breakdowns due to tight coupling between modules (e.g., UI directly bound to hardware logic).
- Difficulty scaling new device types or protocols (e.g., Zigbee, BLE, MQTT).
- Latency in command execution under multi user load.
- No consistent data model for storing device states or logs.

In short: *systems worked, but not well enough to evolve.*

Root Cause Analysis

Through system tracing and user feedback, several root causes were identified:

- **Rigid Monolithic Design:** Single layer control loop tied to UI caused dependency issues.
- **Unstructured Data Flow:** No consistent input output model; data validation occurred inconsistently.
- **Reactive Rather Than Predictive Logic:** Traditional models only responded to commands; no optimization or learning layer existed.

- **No Unified Storage Mechanism:** Device states and logs scattered across temporary memory or inconsistent databases.

Solution Ideation & Model Merging

After mapping potential approaches — microservices, hybrid event driven architecture, and modular SOLD we merged key principles:

- **System Oriented Layering (S):** Divide the system into Device Layer, Control Layer, Data Layer, and Interface Layer.
- **Optimization Layer (O):** Introduce a rule engine to manage load, redundancy, and predictive response.
- **Logical Flow (L):** Ensure data follows defined logic from input to decision to execution.
- **Dynamic Adaptability (D):** Implement modular components allowing real time updates or device additions without refactoring.

Thus, the **SOLD Model** emerged providing both structure and agility.

Solution Architecture

Architecture Type: Modular Layered IoT Framework

Layers Defined:

1. **Device Layer:** Embedded sensors and controllers using MQTT over BLE/WiFi.
2. **Control Layer:** Core logic engine processing incoming events and managing device state changes.
3. **Data Layer:** Centralized database with schema for devices, states, and logs.
4. **Interface Layer:** Web and mobile UI with RESTful and WebSocket APIs for real time updates.
5. **Optimization Layer:** SOLD logic — monitors latency, command frequency, and power usage to adjust communication paths dynamically.

Data Flow and Storage

- **Input:** Device signal (sensor or user command).
- **Process:** Control Layer applies SOLD logic → interprets state changes → validates rules.
- **Output:** Actuator command → state update → dashboard visualization.
- **Storage:** PostgreSQL for structured device data; InfluxDB for time series logs; Redis for transient state caching.

API and Integration

All communications exposed via MQTT and REST APIs.
This hybrid interface allowed:

- Local LAN control (MQTT) for low latency.
- Cloud access (REST/WebSocket) for remote management and AI assisted routines.
  Each endpoint documented and tested under modular unit test suites.

UI/UX and Control

Developed a responsive dashboard:

- Real time device visualization.
- Scene control (grouping lights, AC, curtains).
- Predictive "smart routine" recommendations from SOLD optimization logs.

Implementation & Delivery

Development Environment:

- Backend: Python (Flask + MQTT client).
- Frontend: ReactJS (WebSocket updates).
- Database: PostgreSQL + InfluxDB + Redis cache.
- Hardware: ESP32 nodes, BLE sensors, and central MQTT broker.

Testing & Validation:

- 10 simulated households (50+ devices) under varied load.
- Latency improved by **42%** compared to traditional polling models.
- Update failure rate dropped by **65%**.

Result and Impact

The SOLD based home control architecture achieved:

- **Stable, Predictable Behavior:** Reduced dependency conflicts.
- **Flexible Expansion:** New devices integrated with zero backend changes.
- **Faster Execution:** Sub 300ms response time for most actions.

- **Ease of Maintenance:** Modular design allowed independent upgrades.

This approach transformed the IoT system from a *reactive control network* into a *smart adaptive environment.*

Lessons and Knowledge Sharing

The key insight was that **IoT scalability isn't only about hardware or code — it's about architecture logic.** By applying structured reasoning through SOLD (System Oriented Layered Development), the system became *both simple to operate and powerful to extend.*

Case Study: Custom ERP Development Transforming Fragmented Operations into a Unified Intelligent System

Overview

Many businesses operate with multiple disconnected tools spreadsheets for accounting, standalone CRM systems, and manual HR workflows leading to data silos, duplication, and slow decision making. Our objective was to design a **Custom ERP Platform** that unifies all major business functions sales, finance, operations, inventory, and HR under a single adaptive architecture.

Using the **SOLD (System Oriented Layered Development)** model, we built an ERP solution that not only solved current inefficiencies but could *evolve* with the business.

Role: System Architect responsible for defining the architecture, process model, and integration logic across all ERP modules.

Understanding the Problem

The client's existing setup included:

- Independent software for accounts, HR, and sales no central data control.
- Delayed reporting due to manual reconciliation.
- Repetitive data entry and inconsistent records.
- High cost and complexity of scaling new modules (e.g., adding a purchase or project management unit).

The ERP had to deliver **real time visibility**, **data consistency**, and **cross departmental intelligence** all without overhauling legacy systems overnight.

Root Cause Analysis

Deep analysis revealed that conventional ERP implementations often fail due to:

- **Rigid Monolithic Architecture:** Every new module increases system complexity exponentially.
- **Unclear Data Ownership:** Data lives in multiple silos with no unified schema.
- **Weak Process Mapping:** Business logic coded directly into interfaces rather than separated as reusable workflows.
- **Poor Integration Strategy:** APIs treated as afterthoughts, leading to synchronization failures.

These were not just technical flaws they were **architectural misalignments** between process logic and data flow.

Solution Ideation & Model Merging

We evaluated several approaches off the shelf ERP customization, microservices based rebuild, and hybrid API driven models.

The **SOLD framework** (System Oriented Layered Development) was chosen as the foundation because it ensured:

- Clear separation of logic and interface.
- Unified data schema and validation pipeline.
- Adaptable module linking for future scaling.

By merging **domain driven design** with **SOLD layering**, the ERP became both structured and flexible.

Solution Architecture

Architecture Type: Modular Layered ERP Framework (SOLD based)

Layers Defined:

1. **System Layer:** Core engine orchestrating module interaction (Sales ↔ Inventory ↔ Accounting).
2. **Optimization Layer:** Workflow rules, performance tuning, and error recovery.
3. **Logical Layer:** Business process logic reusable across modules (e.g., approval chain, invoice generation).
4. **Dynamic Layer:** UI/API endpoints that adapt as modules evolve.

Data Flow and Storage

- **Input:** User transactions (purchase order, invoice, attendance).
- **Processing:** Logical Layer validates and enriches data.
- **Storage:** Central relational schema (PostgreSQL) with indexed relationships.
- **Access:** APIs deliver real time status to dashboards and mobile clients.

Data flow is **bidirectional**, meaning updates in one module immediately reflect across others via internal event triggers.

API and Integration

- REST APIs for web and mobile access.
- MQTT/WebSocket bridge for real time notifications (e.g., new task assignment, stock updates).
- External integrations: Payment gateways, third party accounting, and HR attendance devices.

Every module communicates through standardized contracts ensuring interoperability and reducing dependency risk.

UI/UX and Process Visualization

The interface design focused on clarity and adaptability:

- Task based navigation replacing traditional menu overload.
- Dynamic dashboards showing KPIs per department.
- Role based access ensuring each user sees only relevant functions.

The UI was built with **React** and **charting visualizations** for live analytics.

Implementation & Testing

Tech Stack:

- Backend: Python (FastAPI), PostgreSQL, Redis cache.
- Frontend: ReactJS with modular component library.
- CI/CD: GitHub Actions, Docker, Nginx based deployment.

Testing Approach:

- Unit testing for each API endpoint.
- Integration testing for cross module operations (Sales ↔ Finance ↔ Inventory).
- Load testing with concurrent 200 users.

Result:

- 48% faster report generation.
- 70% reduction in data inconsistency issues.
- New module deployment time reduced from weeks to **days**.

Result and Impact

The new ERP system delivered:

- **Unified Business Intelligence:** Real time visibility across all departments.
- **Modular Growth:** New modules added without impacting existing ones.
- **Operational Efficiency:** Reduced human error and manual reconciliation.
- **Long Term Adaptability:** Architecture ready for AI driven forecasting and IoT integration (asset tracking, environment sensors).

Lessons and Knowledge Sharing

- Architecture defines sustainability *not* just performance.
- The SOLD framework proved that structuring by logic layers rather than features leads to resilience.
- ERP design is not about replicating business processes it's about **re-engineering them for digital efficiency**.

Case Study: SaaS Platform Development Building Scalable, Adaptive, and Multi-Tenant Systems Using SOLD Architecture

Overview

The shift from on-premise software to SaaS has transformed how businesses deliver value but it has also introduced challenges in scalability, data management, and customization.

Our objective was to architect a **SaaS platform** capable of supporting **multiple clients (tenants)**, each with unique configurations, without duplicating code or compromising performance.

We used the **SOLD (System-Oriented Layered Development)** framework to design a structure that supports **continuous deployment**, **horizontal scaling**, and **real-time analytics** all while maintaining data isolation and predictable behavior.

Role: System Architect responsible for defining architecture, database multi-tenancy strategy, and service orchestration.

Understanding the Problem

Traditional SaaS implementations often fail to balance **scalability, configurability, and maintainability**.
We identified key limitations in conventional approaches:

- Code duplication for each client's customization.
- Performance degradation with increased tenant load.
- Weak data partitioning leading to privacy and compliance risks.
- Rigid deployment pipelines causing downtime during upgrades.

These challenges demanded an architecture that could scale seamlessly both technically and commercially.

Root Cause Analysis

Upon evaluating existing systems and SaaS design patterns, several root causes emerged:

- **Monolithic Codebase:** All tenants shared the same logic with hard-coded variations.
- **Single-Database Structure:** Tenant data stored in one schema with no robust isolation mechanism.
- **Manual Provisioning:** Each new client setup required developer involvement.
- **Limited Observability:** No centralized logging or metric collection to monitor performance per tenant.

This created operational friction and limited growth beyond a few dozen clients.

Solution Ideation & Model Merging

We explored three major directions:

1. **Pure Multi-Tenant Database Model** (shared schema).
2. **Isolated Database per Tenant** (separate schema or instance).
3. **Hybrid Layered Model** combining shared logic with flexible isolation.

The **SOLD** framework offered a balanced approach:

- **System Layer:** Central control of tenants and global configurations.
- **Optimization Layer:** Smart routing and resource balancing per tenant.
- **Logical Layer:** Modular services (auth, billing, analytics) with reusable logic.
- **Dynamic Layer:** Tenant-specific extensions, UI themes, and APIs.

This merging ensured both performance and flexibility.

Solution Architecture

Architecture Type: Multi-Layered SaaS Framework (SOLD Model)

Core Principles:

- **Single Codebase, Multi-Tenant Execution**
- **API-Driven Communication** (REST + WebSocket)
- **Horizontal Scalability via Containers (Docker/Kubernetes)**
- **CI/CD Integration for Continuous Updates**

Layers Overview:

1. **System Layer:** Manages tenant onboarding, subscription, and routing.
2. **Optimization Layer:** Dynamic resource allocation, caching, and auto-scaling.
3. **Logical Layer:** Business services like authentication, billing, reporting.
4. **Dynamic Layer:** UI customization, extensions, and webhook integrations per tenant.

Data Flow and Storage

- **Input:** Tenant requests through domain-based routing (e.g., tenantA.app.com).
- **Processing:** Middleware applies authentication and routing to proper tenant context.
- **Storage:** Hybrid database model PostgreSQL with schema-based isolation + S3 for file storage.
- **Data Security:** Role-based access, per-tenant encryption, and tokenized API communication.

This approach achieved **logical isolation** without infrastructure duplication.

API and Integration

The system exposed modular APIs to allow:

- External integrations (CRM, payment gateways, analytics).
- Tenant-specific webhooks and REST extensions.
- Internal microservices communication through asynchronous messaging (RabbitMQ).

All APIs followed **OpenAPI standards** for versioning and documentation.

UI/UX and Tenant Customization

- Built with **React + Tailwind** for fast and adaptive front-end rendering.
- Theme and layout dynamically loaded based on tenant configuration.
- Admin console provided:
    - Tenant onboarding and billing control.
    - Feature toggles for per-client activation.
    - Usage analytics dashboard.

Implementation & Testing

Tech Stack:

- Backend: Python (FastAPI), PostgreSQL, Redis, RabbitMQ.
- Frontend: ReactJS, WebSocket real-time updates.
- Deployment: Docker, Kubernetes, Nginx ingress, CI/CD via GitHub Actions.

Testing:

- Load testing under 1,000+ concurrent users.
- Tenant isolation validation zero data leakage under simulated cross-access tests.
- Failover simulation using container restarts and horizontal autoscaling.

Result:

- Onboarding time reduced from 2 days to **10 minutes**.
- Average API latency dropped by **38%**.
- Zero downtime during version upgrades.

Result and Impact

The SOLD-driven SaaS architecture achieved:

- **True Scalability:** System scaled horizontally across nodes with auto-balancing.
- **Multi-Tenant Isolation:** Each tenant logically independent yet part of shared infrastructure.
- **Operational Agility:** Continuous updates deployed without service interruption.
- **Faster Growth:** Platform could onboard 100+ new clients per month without developer bottleneck.

This model converted complexity into a competitive advantage enabling faster delivery, lower cost, and higher reliability.

Lessons and Knowledge Sharing

- SaaS success depends less on features and more on **architecture discipline**.
- SOLD provided a structured approach to manage scalability, flexibility, and maintainability.
- Separation of logic, optimization, and dynamics proved essential for sustainable SaaS growth.