October 16 - Code Conversion

Friday, October 17, 2025 9:14 AM

Section 1 – AS/400 RPG Modernization and Assessment

Purpose

To document the evolution of legacy RPG and its supporting constructs (DDS, ILE, Data Areas, and DB2/400 access), and to define how automated tools can analyze, decompose, and prepare this legacy system for migration to a modern, modular, cloud-native NodeJS architecture.

Key Notes

1. RPG Language Evolution

- RPG III introduced structured constructs (IF/END, DO, subroutines).
- RPG IV (ILE RPG) delivered free-format syntax, arithmetic functions, and componentization.
- **ILE (Integrated Language Environment)** enables modularity through *service programs*, *activation groups*, and *dynamic binding* — foundational for decomposing monolithic AS/400 code.

2. DDS (Data Definition Specifications)

- Defines both **database (PF/LF)** and **UI layout (DSPF)**.
- Logical Files (LFs) may embed business logic within data-access paths and must be re-implemented in the target ORM layer.
- DDS ties UI elements tightly to logic — modernization requires decoupling into front-end views and REST APIs.

3. Dynamic vs Static Calls

- CALL (dynamic) hides dependencies and complicates analysis.
- CALLP (static/prototyped) allows compile-time validation; map these directly to service calls in NodeJS.

4. Data Access Patterns

- RPG uses both *native record-level I/O* and *embedded SQL*.
- Each access style must be mapped appropriately to ORM patterns and database transactions.

5. Data Areas (DTAARA) & Shared Work Files – Hidden Dependencies

- **Data Areas (DTAARA):** global storage shared across programs — modern equivalents should become configuration stores or in-memory cache.
- **Shared Physical Files:** temporary "work files" written by one job and read by another; in the target stack use message queues or transactional staging tables.

6. Automated Assessment & Analysis Tools

- Tools like **Fresche X-Analysis**, **i400hub RPG/Code Inspector**, and **Blu Insights** parse RPG, COBOL, and CL code to generate dependency graphs, call stacks, and data flows.
- These provide the "as-is" blueprint for the modernization effort.

7. Security and Quality Scanning

- Tools such as **ARCAD Code Checker** and **Sonar for RPG** enforce coding standards and detect code smells or vulnerabilities.
- Particularly critical is identifying potential **SQL injection** risks in dynamically constructed queries prior to migration.

8. Dependency Analysis Depth

- Automated parsers capture:
    - Logical dependencies (CALL, CALLB, CALLP).
    - Inclusion dependencies (COPY, INCLUDE).
    - Resource dependencies (EXTNAME, DTAARA).
- Ensures that modules are migrated in the correct order and no hidden links are broken.

9. Assessment Deliverables / Blueprint Artifacts

- The analysis should produce:
    - **Program interaction graphs** (call hierarchies).
    - **Complexity heatmaps** (high-risk code zones).
    - **Dead-code inventory** for elimination.
    - **Field and table mapping sheets** for ORM conversion.
- These outputs form the foundation for metadata and mapping CSVs in Seaboard's AI conversion pipeline.

Actions / Next Steps

- **Inventory Programs and Dependencies:** Execute Blu Insights / X-Analysis to generate call graphs and DDS mappings.
- **Extract Data Models:** Convert PF/LF definitions into ER models for ORM schema design.
- **Classify Programs:** Tag as interactive, batch, service, or utility to guide layer placement.
- **Assess Hidden Dependencies:** Document Data Areas and shared work files; design modern equivalents (queues or cache).
- **Run Quality and Security Scans:** Use ARCAD/Sonar to create a pre-migration baseline and track issues.
- **Build Dependency Map:** Capture CALL / CALLP / COPY / INCLUDE / EXTNAME relationships in metadata store.
- **Generate Assessment Dashboard:** Summarize modules by complexity, dependencies, and migration priority.

Questions / Considerations

- Which ILE service programs are shared across modules and require first-phase migration?
- How can indicator-based DDS logic be converted into modern UI frameworks (visibility / validation rules)?
- Should native record I/O be wrapped temporarily or replaced immediately with ORM calls?
- What is the preferred tool stack for dependency visualization (X-Analysis vs Blu Insights)?
- How will cross-module data areas be governed to prevent state conflicts post-migration?

✅ **Connection to Seaboard Source → Target Architecture**

This section represents the **source-architecture baseline** underpinning Seaboard's modernization.
It directly supports your current initiative to build **metadata linkages (module ↔ program ↔ job)**, perform **AI-based dependency analysis**, and plan **module-wise conversion** with traceability.
Once Section 1's deliverables are generated (as-is blueprint, dependency map, and quality baseline), they flow into Sections 2 and 3 for target-architecture design and Strangler Fig implementation.

Section 2 – Strategic Framework for Modernization: The Strangler Fig Approach

Purpose

To define the **coexistence framework** that enables Seaboard's legacy AS/400 applications and the new NodeJS-based platform to run in parallel during migration.
This section introduces the **Strangler Fig pattern** and **Change Data Capture (CDC)** as the two pillars ensuring functional continuity, data integrity, and user-transparent transition.

Key Notes

1. Strangler Façade (Proxy Layer)

- The façade acts as a unified **router and mediator** for all requests—legacy and modern.
- Implemented typically as an **API Gateway** or **reverse proxy** (e.g., Kong, Express Gateway, NGINX).
- Decision logic per request:
  - If the requested functionality is already migrated → route to the new NodeJS microservice.
  - If it still resides in the legacy RPG system → forward to AS/400.
- Routing configuration evolves incrementally as modules migrate.
- This layer provides a **seamless user experience**—no front-end disruption when back-end ownership changes.

2. Coexistence and Gradual Transition

- The façade enables **incremental rollout** by module, supporting hybrid traffic (modern + legacy).
- Governance rules must document which endpoints belong to which system to avoid routing drift.
- Logs from the proxy layer provide traceability and aid rollback if a migrated API misbehaves.
- In Seaboard's architecture this façade also supports **feature flag toggling** for module activation.

3. Data Synchronization via Change Data Capture (CDC)

- During the coexistence period, **both systems share business data**; consistency between DB2/400 and PostgreSQL is essential.
- Batch ETL is too slow; near-real-time synchronization is required.
- **CDC** uses IBM i journals to capture insert/update/delete events and replicate them to the target DB.

4. CDC Implementation on IBM i

1. **Journaling** – Every change to a physical file is recorded in a Journal Receiver with before/after images.
2. **Enabling CDC** – Run STRJRNPRF (Start Journal Physical File) and set DATA CAPTURE CHANGES so DB2 logs full-row images.
3. **CDC Tools** – Use CData Sync, Precisely Connect CDC, Qlik Replicate or Debezium connectors to stream journal entries to PostgreSQL.

- These tools apply changes continuously so both databases stay aligned with millisecond-level latency.

5. Data Integrity and Rollback Control

- Journaling provides a historical audit trail enabling point-in-time recovery.
- Any CDC failure should trigger an **alert and retry policy**; checkpoints must ensure no duplicate writes.
- The same CDC stream can populate a **data lake** for analytics without impacting production loads.

6. Complementary Coexistence Mechanisms (from full document)

- **Anti-Corruption Layer (ACL):** wraps legacy APIs to shield new services from legacy idiosyncrasies.
- **Contract Validation:** every migrated endpoint must pass golden-scenario parity checks before being exposed through the façade.
- **Versioning Strategy:** the proxy maintains versioned routes (/v1/legacy/…, /v2/modern/…) to support gradual cut-over.
- **Security Context Propagation:** session tokens and audit IDs must flow end-to-end across legacy and modern boundaries.

Actions / Next Steps

- Design and implement the **Strangler Façade Gateway**, defining route tables for legacy vs modern services.
- Document **routing rules** and maintain them in version control with module status.
- Select and configure a **CDC tool** (CData Sync / Precisely Connect / Debezium) for DB2→PostgreSQL replication.
- Enable **journaling** on all physical files participating in CDC; verify DATA CAPTURE CHANGES.
- Set up **monitoring dashboards** for CDC lag, replication errors, and journal receiver growth.
- Establish a **rollback plan** – how to revert to legacy routes or data if synchronization fails.
- Integrate **feature flags** within the façade to toggle individual module endpoints.

Questions / Considerations

- What gateway technology best fits Seaboard's IBM i network constraints (API Gateway vs custom NodeJS proxy)?
- Should CDC operate as continuous streaming or micro-batch (every few seconds)?
- How is referential integrity enforced when transactions span legacy and modern tables?
- How do we handle bidirectional updates (legacy writes ↔ modern writes)?
- What are the SLAs for CDC latency and data consistency verification?

✅ **Connection to Seaboard Source → Target Architecture**
Section 2 translates Seaboard's high-level modernization goals into an executable coexistence plan.
It directly supports the team's current strategy to **maintain hybrid operations**—70 modules still in legacy while a few run on modern NodeJS.
The façade aligns with your discussion about **routing logic + feature flagging**, and CDC embodies the **data-sync requirement** you raised with Siva's team.
Together, they form the **bridge** between Section 1's "as-is" RPG assessment and Section 3's target NodeJS architecture.

Section 3 – Designing the Target 10-Layer NodeJS Architecture

Purpose

To define the **target-state architecture** for the Seaboard modernization initiative — a **10-layer NodeJS blueprint** that enables modularity, scalability, security, and future cloud-native extensibility.
This section establishes clear design patterns (Dependency Injection, RESTful APIs, Multi-layered Security) to ensure the new architecture mirrors AS/400's business logic fidelity while eliminating its rigidity.

Key Notes

1. Architectural Overview – The 10-Layer Blueprint

A fully decoupled 10-layer architecture promotes clarity and evolution.
Each layer is a self-contained domain that communicates only through defined interfaces:

Layer	Responsibility
1. Presentation Layer (Client)	UI or external consumers (web, mobile, partner apps).
2. API Gateway Layer	Routes incoming requests, applies rate limits, logging, and authentication.
3. Controller / API Layer	Handles REST endpoints; converts HTTP input into service commands.
4. Service Layer	Business logic orchestration; manages workflows and transactional consistency.

5. Domain Model Layer	Domain entities and aggregates that encapsulate core business rules.
6. Data Access / Repository Layer	Repository interfaces for interacting with persistence logic.
7. Persistence Layer (ORM)	ORM framework (e.g., Sequelize, Prisma) mapping between domain and DB.
8. Database Layer	Physical data store (PostgreSQL or Aurora); normalized schema replacing DDS.
9. Cross-Cutting Concerns Layer	Logging, caching, monitoring, auditing, configuration.
10. Integration / ACL Layer	Legacy connectors, message queues, and anti-corruption wrappers.

Additional Blueprint Details (from full document):

- Introduces **API composition and gateway federation** (GraphQL or BFF) for complex queries.
- Includes **observability hooks** (OpenTelemetry, Winston, or Elastic APM) for full-stack traceability.
- Emphasizes **12-Factor App** compliance for deployment and scaling.

2. Managing Dependencies – Implementing Dependency Injection (DI)

- Hardcoded dependencies cause tight coupling; DI **inverts control** by injecting dependencies externally.
- Enables **testability** and **maintainability**—modules rely on interfaces, not concrete implementations.
- Example: instead of a service creating its own repository, the repository instance is injected at runtime.
- Supports frameworks such as **InversifyJS** or **TypeDI** to manage lifecycle and scope of injected objects.
- Facilitates **mocking** and **unit testing** for faster CI/CD validation.

3. API Design – RESTful Strategy to Replace Procedural Calls

Legacy systems follow procedural calls (CALL 'PROGRAM' PARM(data)); NodeJS architecture replaces this with RESTful APIs.
Key Design Guidelines:

- **Resource-Oriented URIs:** Use nouns (/customers, /orders) instead of verbs.
- **Standard HTTP Methods:**
    - GET – Retrieve data
    - POST – Create data
    - PUT/PATCH – Update data
    - DELETE – Remove data
- **HTTP Status Codes:** Use proper codes (200 OK, 201 Created, 400 Bad Request, 404 Not Found, 500 Internal Server Error).
- **API Versioning:** Version URIs (e.g., /api/v1/customers) to allow safe evolution.
- **Data Transfer Objects (DTOs):** Clearly define request/response payloads to decouple public APIs from internal domain structures.
- **Error Handling & Validation:** Integrate middleware for schema validation (e.g., **Zod**, **Joi**) to ensure predictable API responses.
- **API Documentation:** Generate OpenAPI/Swagger specs automatically for traceability.

From extended blueprint:

- Introduces **API composition layer** to combine multiple microservice responses.
- Includes **rate-limiting and throttling policies** at the gateway to prevent abuse.
- Defines **API Governance Board**—ensuring contract uniformity across modules.

4. Securing the New Architecture – Multi-Layered Strategy

Security must be embedded throughout, not added later.
A robust NodeJS security posture includes:

- **Authentication & Authorization:**
    - Use OAuth 2.0 / JWT for token-based auth.
    - Integrate with LDAP or Azure AD for enterprise SSO.
- **Input Validation & Sanitization:** Prevent SQL injection, XSS, CSRF.
- **Secrets Management:** Centralize using Vault, AWS Secrets Manager, or Azure Key Vault.
- **Transport Security:** Enforce HTTPS/TLS 1.3 and secure cookies.
- **Access Logging:** Track all API calls for auditing (link with Section 2 façade logs).
- **DevSecOps Pipeline:** Integrate Snyk, OWASP Dependency Check, and dynamic scanning into CI/CD.
- **Least Privilege Policy:** Define role-based access per microservice and database schema.
- **Audit Trails:** Maintain immutable logs for compliance (SOX, HIPAA, etc.).

5. Observability and Monitoring (Additional from full document)

- Use **structured logging (Winston, Pino)** for consistent log correlation.
- Add **APM tools** like New Relic or Datadog for latency and error tracking.
- Implement **health checks** and **circuit breakers** using tools like **Resilience4Node**.
- Define **SLOs** for each microservice and integrate alerting through Prometheus / Grafana.

Actions / Next Steps

- Define module-level blueprints aligning to the 10-layer architecture.
- Implement a **DI container** and refactor service/repository dependencies.
- Create a **REST API guideline document** (naming conventions, DTOs, error format).
- Develop **API Gateway policies** (auth, rate limits, versioning, logging).
- Integrate **security scanning** and secrets management in DevOps pipeline.
- Implement **centralized logging and monitoring** for API and DB interactions.
- Validate architecture through one pilot module before scaling to all 70+ modules.

Questions / Considerations

- How should existing DDS-based field validation rules map into REST DTO validation?
- Will NodeJS services directly use PostgreSQL, or should a data service layer abstract it?
- Which DI library (InversifyJS, Awilix, TypeDI) best aligns with Seaboard's coding conventions?
- How should authentication propagate across microservices — centralized SSO or service-to-service tokens?
- Should we integrate GraphQL or gRPC for inter-service communication later?

Layer	Purpose
1. Presentation / API Gateway	Handles UI or external requests, routing, and API security.

✅ **Connection to Seaboard Source → Target Architecture**
Section 3 finalizes the **target state** discussed throughout your earlier conversations — transforming the AS/400 monolith into a **modular, governed NodeJS ecosystem**.

It connects directly to:

- Section 1's **dependency and DDS analysis** (informing ORM and API modeling).
- Section 2's **proxy + CDC coexistence model** (ensuring safe transition).
- Your ongoing AI-driven code and architecture generation experiments, where each layer's metadata (services, repositories, APIs) can be auto-mapped and validated through LLM-based orchestration.

2. Controller / API Layer	Defines REST endpoints, request validation, and response shaping.
3. Service Layer	Implements core business logic and orchestration.
4. Domain / Repository Layer	Encapsulates domain models and data access.
5. Persistence (ORM + DB)	Connects with PostgreSQL (replacing DB2), using Sequelize or Prisma.
6. Cross-Cutting & Integration	Logging, monitoring, caching, and legacy connectors (ACL).

This structure retains the **core separation of concerns** — clean, testable, and scalable — while avoiding over-engineering in early phases.
The additional layers from the original 10-layer model (like a dedicated Domain Model Layer or split API Gateway) can be added later as the system stabilizes.

Section 4 – Phased Migration & Data Reconciliation Strategy

Purpose

To define Seaboard's controlled migration methodology ensuring continuity, traceability, and integrity during the gradual transition from RPG to NodeJS.
This section converts the conceptual Strangler-Fig pattern into an actionable execution model emphasizing modular rollout, continuous data sync, and verification throug reconciliation.

Key Notes

1. **Incremental Cut-Over Model**
   • Migration proceeds module-by-module, with legacy and modern components running in parallel.
   • A Routing Façade decides request paths; CDC maintains data consistency.
   • Each phase includes a formal validation cycle before legacy shutdown.
2. **Continuous Synchronization via CDC**
   • Post-bulk load, CDC monitors DB2/400 journals and streams delta changes to PostgreSQL in near real time
   Architectural Blueprint for Mig...
   .
   • Both systems operate on current data during the coexistence window.
3. **Multi-Layered Data Reconciliation**
   • **Pre-Migration Profiling:** Identify duplicates, nulls, and inconsistencies before movement.
   • **Level 1 – Record Counts:** Match row counts source vs target.
   • **Level 2 – Column Aggregates:** Checksum numeric totals (SUM, AVG, MIN, MAX).
   • **Level 3 – Field-Level Sampling:** Validate encoding and precision on subset records.
   • **Level 4 – Business Rule Validation:** Run identical transactions in both systems to prove functional equivalence
   Architectural Blueprint for Mig...
   .
   • Use automated diff tools (e.g., Datafold) to record and audit results.
4. **Post-Migration Audit & Decommissioning**
   • After stabilization, run performance and support reviews; confirm no legacy dependencies.
   • Retire RPG objects and update routing rules in the Façade
   Architectural Blueprint for Mig...
   .

Actions / Next Steps

• Define phase criteria (criticality, complexity, dependencies).
• Set CDC latency SLA and alerting thresholds.
• Automate data reconciliation in CI/CD.
• Prepare rollback and cut-over checklists.

Questions / Considerations

• What is the max acceptable CDC lag (ms)?
• Who signs off on data parity and module stabilization?

✅ **Connection to Seaboard Source → Target Architecture**
Section 4 operationalizes Sections 2 & 3 by defining how coexistence and 10-layer architecture merge during live cut-over. It connects directly to your CDC and proxy des discussions with Siva's team.

Section 5 – Application Component Mapping (RPG → NodeJS Equivalents)

Purpose

To standardize how each legacy artifact (type, file, routine) translates to its modern counterpart, enabling AI-driven conversion.

Key Notes

1. **Mapping Matrix**
   • DSPF → React components (JSX + state hooks).
   • RPG logic → Node controllers/services.
   • PF/LF files → Sequelize ORM models.
   • Indicators → Boolean flags or middleware states.
   • Subroutines → Reusable utilities.
2. **Transformation Standards**
   • REST endpoints replace CALL/CALLB invocations.
   • Adopt DTOs for input/output contracts.
   • Inject services via DI frameworks (InversifyJS, TypeDI).
3. **Validation & Error Handling**
   • Use middleware (Joi/Zod) for payload checks.
   • Centralize exception logging.

Actions / Next Steps

• Compile sample mappings and train AI translator.
• Define unit-test templates for mapped components.
✅ **Connection**
This section is the "translation grammar" for the AI engine you are building.

Section 6 – Testing & Validation for Functional Parity

Purpose

To prove that the modern system produces identical business results to RPG, maintaining functional parity.

Key Notes

1. **Test Pyramid Re-Definition**
   • Emphasis on fast unit and contract tests over monolithic E2E tests.
   • Contract tests validate API schema consistency across services
   Architectural Blueprint for Mig…
   .
2. **Performance & Scalability Benchmarks**
   • Use k6/JMeter to match AS/400 throughput benchmarks.
   • Track latency, throughput, error rates under load
   Architectural Blueprint for Mig…
   .
3. **Post-Migration Audit & Decommissioning**
   • Audit KPIs and user feedback post-stabilization.
   • Remove legacy routes and source objects upon approval
   Architectural Blueprint for Mig…
   .

✅ **Connection**
Validates Section 4's CDC output and ensures Section 3's 10-layer modules perform to parity.

Section 7 – Special Case: Migrating Synon / CA 2E Applications

Purpose

To outline a separate strategy for Seaboard's modules developed in Synon/CA 2E, which generate RPG code from a model-based repository.

Key Notes

1. **Model-Driven Transformation**
   • Focus on Synon design model as source of truth, not generated RPG
   Architectural Blueprint for Mig…
   .
   • Prevents cryptic, line-by-line translation.
2. **Architectural Consistency**
   • Synon already enforces MVC-like structure (DB, logic, UI) that maps cleanly to Node's Model-Service-Controller pattern
   Architectural Blueprint for Mig…
   .
3. **Handling Custom Exits**
   • External EXCUSRPGM logic requires manual migration as stand-alone services.

✅ **Connection**
Links back to Section 1's dependency maps to detect and segregate custom code paths in CA 2E modules.

Section 8 – AI as a Modernization Accelerator

Purpose

To demonstrate how GenAI reduces manual effort across discovery, translation, validation, and testing.

Key Notes

1. **AI-Powered Discovery & Analysis**
   • LLMs (e.g., watsonx Code Assistant) parse RPG to plain English and extract business rules and dependencies
   Architectural Blueprint for Mig…
   .
2. **AI-Assisted Code Transformation**
   • Converts RPG → JavaScript/TypeScript using intelligent refactoring rather than syntax conversion.
   • Human-in-the-loop review ensures alignment with Section 3's architecture
   Architectural Blueprint for Mig…
   .
3. **AI-Enhanced Data Migration & Validation**
   • Automates data profiling, mapping, and reconciliation for legacy datasets
   Architectural Blueprint for Mig…
   .
4. **AI-Generated Testing**
   • Creates test cases and scripts from legacy logic to ensure functional parity
   Architectural Blueprint for Mig…
   .

✅ **Connection**
Directly supports your AI prompt framework for program conversion and automated testing POCs.

Section 9 – Navigating Pitfalls & Ensuring Success

Purpose

To identify non-technical risks that jeopardize modernization and to propose mitigation governance and knowledge-transfer models.

Key Notes

1. **Knowledge Deficit & Talent Attrition**
   • RPG expertise is vanishing; must capture tribal knowledge via pair programming and automated documentation (X-Analysis)
   Architectural Blueprint for Mig…
   .
2. **Managing Legacy Code Complexity**
   • Hidden dependencies cause migration risk; dependency graph tools mitigate this.

3. **Organizational Change Management**
   • Shift RPG teams from maintenance to mentorship.
   • Establish Modernization Center of Excellence (COE) for governance.

✅ **Connection**
Ensures Seaboard's long-term continuity and reduces the risk of post-migration skill gaps.