Legacy Modernization Agents: COBOL to Spring Boot Migration

Scala 3 + ZIO 2.x Effect-Oriented Programming Implementation

Version: 1.0.0
Author: Engineering Team
Date: February 5, 2026
Status: Initial Design

Executive Summary

This project implements an AI-powered legacy modernization framework for migrating COBOL mainframe applications to modern Spring Boot microservices using Scala 3 and ZIO 2.x. The system leverages Google Gemini CLI in non-interactive mode to orchestrate specialized AI agents that perform analysis, transformation, and code generation tasks[1].

Key Objectives

- Automate COBOL-to-Java Spring Boot migration with minimal manual intervention
- Preserve business logic integrity through multi-phase validation
- Generate production-ready microservices with modern architectural patterns
- Provide comprehensive documentation and traceability throughout migration
- Enable team collaboration through agent-based task decomposition

Technology Stack

- **Core Language:** Scala 3 with latest syntax and features
- **Effect System:** ZIO 2.x for functional, composable effects
- **AI Engine:** Google Gemini CLI (non-interactive mode)
- **Target Platform:** Spring Boot microservices (Java 17+)
- **Build Tool:** sbt 1.9+
- **Testing:** ZIO Test framework

Architecture Principles

This implementation follows Effect-Oriented Programming (EOP) principles, treating all side effects (AI calls, file I/O, logging) as managed effects within the ZIO ecosystem. The system is designed for composability, testability, and observability.

Table of Contents

---

1. Project Overview

1.1 Problem Statement

Legacy COBOL systems represent decades of accumulated business logic in financial, insurance, and government sectors. These systems face critical challenges[5]:

- Shrinking pool of COBOL developers
- High operational costs on mainframe infrastructure
- Difficulty integrating with modern cloud-native ecosystems
- Limited agility for business changes and innovation

1.2 Solution Approach

Our framework decomposes the migration into distinct phases, each handled by specialized AI agents orchestrated through ZIO effects:

Phase	Primary Agent	Output
Discovery	CobolDiscoveryAgent	File inventory, dependencies
Analysis	CobolAnalyzerAgent	Structured analysis JSON
Mapping	DependencyMapperAgent	Dependency graph
Transformation	JavaTransformerAgent	Spring Boot code
Validation	ValidationAgent	Test results, reports
Documentation	DocumentationAgent	Technical docs

Table 1: Migration phases and responsible agents

1.3 Expected Outcomes

- Functional Spring Boot microservices equivalent to COBOL programs
- Comprehensive dependency mapping and architecture documentation
- Unit and integration tests for generated code
- Migration reports with metrics and quality indicators
- Reusable agent framework for future migrations

---

2. Architecture and Design

2.1 System Architecture

Figure 1: High-level system architecture showing agent orchestration

The system follows a layered architecture:

Layer 1: Agent Orchestration

- Main orchestrator built with ZIO workflows
- Agent lifecycle management
- State management and checkpointing

Layer 2: Agent Implementations

- Specialized agents for different tasks
- Gemini CLI integration wrapper
- Prompt engineering and context management

Layer 3: Core Services

- File I/O services
- Logging and observability
- Configuration management
- State persistence

Layer 4: External Integrations

- Gemini CLI non-interactive invocation
- Git integration for version control
- Report generation services

2.2 Effect-Oriented Design with ZIO

All system operations are modeled as ZIO effects:

- **ZIO[R, E, A]:** Core effect type representing computation requiring environment R, failing with E, or succeeding with A
- **ZLayer:** Dependency injection for services
- **ZIO Streams:** Processing large COBOL codebases incrementally
- **ZIO Test:** Property-based and effect-based testing
- **Ref and Queue:** Concurrent state management

Example effect signature for COBOL analysis:

```
def analyzeCobol(file: CobolFile): ZIO[GeminiService & Logger, AnalysisError, CobolAnalysis]
```

2.3 Gemini CLI Integration Strategy

Google Gemini CLI supports non-interactive mode for automation[6][10]:

Non-interactive invocation

```
gemini -p "Analyze this COBOL code: $(cat program.cbl)" --json-output
```

Our ZIO wrapper provides:

- Process execution with streaming output
- Timeout handling
- Retry logic with exponential backoff
- Response parsing and validation
- Cost tracking and rate limiting

2.4 Project Structure

```
legacy-modernization-agents/
├── build.sbt
├── project/
├── src/
│ ├── main/
│ │ └── scala/
│ │ ├── agents/ # Agent implementations
│ │ ├── core/ # Core services
│ │ ├── models/ # Domain models
│ │ ├── orchestration/ # Workflow orchestration
│ │ └── Main.scala
│ └── test/
│ └── scala/
├── docs/
│ ├── adr/ # Architecture Decision Records
│ ├── findings/ # Findings and observations
│ ├── progress/ # Progress tracking
│ └── deep-dive/ # Detailed task breakdowns
├── cobol-source/ # Input COBOL files
├── java-output/ # Generated Spring Boot code
├── reports/ # Migration reports
└── README.md
```

---

3. Agent Ecosystem

3.1 Agent Architecture

Each agent is a self-contained ZIO service with:

- Defined input/output contracts
- Specialized prompt templates
- Context management capabilities
- Error handling strategies
- Performance metrics collection

3.2 Core Agent Types

3.2.1 CobolDiscoveryAgent

Purpose: Scan and catalog COBOL source files and copybooks.

Responsibilities:

- Traverse directory structures
- Identify .cbl, .cpy, .jcl files
- Extract metadata (file size, last modified, encoding)
- Build initial file inventory

Interactions:

- Output consumed by: CobolAnalyzerAgent, DependencyMapperAgent

3.2.2 CobolAnalyzerAgent

Purpose: Deep structural analysis of COBOL programs using AI.

Responsibilities:

- Parse COBOL divisions (IDENTIFICATION, ENVIRONMENT, DATA, PROCEDURE)
- Extract variables, data structures, and types
- Identify control flow (IF, PERFORM, GOTO statements)
- Detect copybook dependencies
- Generate structured analysis JSON

Interactions:

- Input from: CobolDiscoveryAgent
- Output consumed by: JavaTransformerAgent, DependencyMapperAgent

3.2.3 DependencyMapperAgent

Purpose: Map relationships between COBOL programs and copybooks.

Responsibilities:

- Analyze COPY statements and program calls
- Build dependency graph
- Calculate complexity metrics
- Generate Mermaid diagrams
- Identify shared copybooks as service candidates

Interactions:

- Input from: CobolDiscoveryAgent, CobolAnalyzerAgent
- Output consumed by: JavaTransformerAgent, DocumentationAgent

3.2.4 JavaTransformerAgent

Purpose: Transform COBOL programs into Spring Boot microservices.

Responsibilities:

- Convert COBOL data structures to Java classes/records
- Transform PROCEDURE DIVISION to service methods
- Generate Spring Boot annotations and configurations
- Implement REST endpoints for program entry points
- Create Spring Data JPA entities from file definitions
- Handle error scenarios with try-catch blocks

Interactions:

- Input from: CobolAnalyzerAgent, DependencyMapperAgent
- Output consumed by: ValidationAgent, DocumentationAgent

3.2.5 ValidationAgent

Purpose: Validate generated Spring Boot code for correctness.

Responsibilities:

- Generate unit tests using JUnit 5
- Create integration tests for REST endpoints
- Validate business logic preservation
- Check compilation and static analysis
- Generate test coverage reports

Interactions:

- Input from: JavaTransformerAgent
- Output consumed by: DocumentationAgent

3.2.6 DocumentationAgent

Purpose: Generate comprehensive migration documentation.

Responsibilities:

- Create technical design documents
- Generate API documentation
- Document data model mappings
- Produce migration summary reports
- Create deployment guides

Interactions:

- Input from: All agents

- Output: Final documentation deliverables

3.3 Agent Interaction Patterns

Figure 2: Agent interaction and data flow diagram

Agents communicate through typed messages and shared state managed by ZIO Ref and Queue:

```
case class AgentMessage(
id: String,
sourceAgent: AgentType,
targetAgent: AgentType,
payload: Json,
timestamp: Instant
)
```

4. Macro Steps and Workflows

4.1 Migration Pipeline Overview

The migration follows six macro steps executed sequentially:

1. **Step 1: Discovery and Inventory**
2. **Step 2: Deep Analysis**
3. **Step 3: Dependency Mapping**
4. **Step 4: Code Transformation**
5. **Step 5: Validation and Testing**
6. **Step 6: Documentation Generation**

4.2 Step 1: Discovery and Inventory

Duration Estimate: 5-10 minutes for typical codebase

Inputs:

- COBOL source directory path
- Include/exclude patterns

Process:

1. Scan directory tree for COBOL files
2. Extract file metadata
3. Categorize files (programs vs copybooks vs JCL)
4. Generate inventory JSON

Outputs:

- `inventory.json` - Complete file catalog
- `discovery-report.md` - Human-readable summary

Success Criteria:

- All COBOL files discovered
- No permission errors
- Inventory contains accurate metadata

4.3 Step 2: Deep Analysis

Duration Estimate: 30-60 minutes for 100 programs

Inputs:

- File inventory from Step 1
- COBOL source files

Process:

1. For each COBOL file, invoke CobolAnalyzerAgent
2. Extract structural information using Gemini AI
3. Parse AI response into structured JSON
4. Store analysis results

Outputs:

- `analysis/<filename>.json` - Per-file analysis
- `analysis-summary.json` - Aggregated statistics

Success Criteria:

- All files analyzed successfully
- Structured data validated against schema
- No AI invocation failures

4.4 Step 3: Dependency Mapping

Duration Estimate: 10-20 minutes

Inputs:

- File inventory
- Analysis results from Step 2

Process:

1. Extract COPY statements and program calls
2. Build directed dependency graph
3. Calculate complexity metrics
4. Generate Mermaid diagram
5. Identify service boundaries

Outputs:

- `dependency-map.json` - Graph representation
- `dependency-diagram.md` - Mermaid visualization
- `service-candidates.json` - Recommended microservice boundaries

Success Criteria:

- Complete dependency graph
- No orphaned nodes
- Service boundaries identified

4.5 Step 4: Code Transformation

Duration Estimate: 60-120 minutes for 100 programs

Inputs:

- Analysis results
- Dependency map
- Transformation templates

Process:

1. For each COBOL program, invoke JavaTransformerAgent
2. Generate Spring Boot project structure
3. Create domain models from DATA DIVISION
4. Transform procedures to service methods
5. Generate REST controllers and configurations
6. Apply Spring annotations

Outputs:

- `java-output/<package>/` - Spring Boot projects
- `transformation-report.json` - Transformation metrics

Success Criteria:

- All programs transformed
- Generated code compiles
- Spring Boot conventions followed

4.6 Step 5: Validation and Testing

Duration Estimate: 30-45 minutes

Inputs:

- Generated Spring Boot code
- Original COBOL analysis

Process:

1. Generate unit tests for each service
2. Create integration tests for REST endpoints
3. Validate business logic preservation
4. Run static analysis tools
5. Generate coverage reports

Outputs:

- `tests/<package>/` - Generated test suites
- `validation-report.json` - Test results and coverage

Success Criteria:

- All tests generated and passing
- Minimum 70% code coverage
- No critical static analysis violations

4.7 Step 6: Documentation Generation

Duration Estimate: 15-20 minutes

Inputs:

- All previous outputs
- Migration metadata

Process:

1. Aggregate data from all phases
2. Generate technical design documents
3. Create API documentation
4. Produce migration summary
5. Generate deployment guides

Outputs:

- `docs/technical-design.md`
- `docs/api-reference.md`
- `docs/migration-summary.md`
- `docs/deployment-guide.md`

Success Criteria:

- Complete documentation set
- All diagrams rendered
- No broken references

---

5. Deep-Dive Task Breakdown

This section outlines the micro-tasks for each macro step, structured for assignment to AI coding agents (Claude, GitHub Copilot, OpenAI Codex).

5.1 Task Format

Each task follows this structure:

- **Task ID:** Unique identifier
- **Title:** Brief description
- **Agent Type:** Recommended AI agent
- **Dependencies:** Required prior tasks
- **Inputs:** Required artifacts/context
- **Outputs:** Expected deliverables
- **Acceptance Criteria:** Definition of done
- **Complexity:** Low/Medium/High

- **Estimated Effort:** Time estimate

5.2 Deep-Dive Folders

Detailed task breakdowns are organized in separate markdown files:

- `docs/deep-dive/01-discovery-tasks.md`
- `docs/deep-dive/02-analysis-tasks.md`
- `docs/deep-dive/03-dependency-mapping-tasks.md`
- `docs/deep-dive/04-transformation-tasks.md`
- `docs/deep-dive/05-validation-tasks.md`
- `docs/deep-dive/06-documentation-tasks.md`

See Appendix A for complete task listings.

---

6. Agent Skill Definitions

6.1 Skill Definition Format

Each agent type has a corresponding skill definition markdown file specifying:

- Core competencies
- Knowledge domains
- Interaction protocols
- Error handling strategies
- Performance requirements

6.2 Agent Skill Files

- `docs/agent-skills/cobol-discovery-agent-skill.md`
- `docs/agent-skills/cobol-analyzer-agent-skill.md`
- `docs/agent-skills/dependency-mapper-agent-skill.md`
- `docs/agent-skills/java-transformer-agent-skill.md`
- `docs/agent-skills/validation-agent-skill.md`
- `docs/agent-skills/documentation-agent-skill.md`

See Appendix B for complete skill definitions.

---

7. Progress Tracking Framework

7.1 Progress Tracking Files

The project includes structured progress tracking:

- `docs/progress/overall-progress.md` - High-level status dashboard
- `docs/progress/step-01-discovery-progress.md`
- `docs/progress/step-02-analysis-progress.md`
- `docs/progress/step-03-dependency-mapping-progress.md`
- `docs/progress/step-04-transformation-progress.md`
- `docs/progress/step-05-validation-progress.md`
- `docs/progress/step-06-documentation-progress.md`

7.2 Progress Metrics

Each step tracks:

Metric	Description
Status	Not Started / In Progress / Complete / Blocked
Completion %	0-100% progress indicator
Files Processed	Count of files handled
Success Rate	Percentage of successful operations
Duration	Actual time spent
Blockers	Current impediments

Table 2: Progress tracking metrics

7.3 Automated Progress Updates

Progress tracking integrates with the ZIO effect system:

```
def trackProgress(step: MigrationStep, status: Status): ZIO[ProgressTracker, Nothing, Unit]
```

Progress updates are automatically written to markdown files using ZIO Streams.

---

8. Architecture Decision Records (ADRs)

8.1 ADR Overview

ADRs document significant architectural decisions and their rationale. Each ADR follows the format:

- Title
- Status (Proposed / Accepted / Deprecated / Superseded)
- Context
- Decision
- Consequences
- Alternatives Considered

8.2 Key ADRs

1. **ADR-001:** Use Scala 3 + ZIO 2.x for Implementation
2. **ADR-002:** Adopt Google Gemini CLI for AI Operations
3. **ADR-003:** Target Spring Boot for Microservice Generation
4. **ADR-004:** Use Effect-Oriented Programming Pattern
5. **ADR-005:** Implement Agent-Based Architecture
6. **ADR-006:** Store State in JSON Files vs Database
7. **ADR-007:** Generate Mermaid Diagrams for Visualization
8. **ADR-008:** Use ZIO Test for Testing Framework

See `docs/adr/` directory for complete ADR documents.

---

9. Findings and Lessons Learned

9.1 Technical Findings

Finding 1: COBOL Complexity Varies Significantly

Analysis of legacy codebases reveals 3 complexity tiers[5]:

- Tier 1 (30%): Simple batch programs, straightforward transformation
- Tier 2 (50%): Moderate complexity with file I/O and business rules
- Tier 3 (20%): High complexity with embedded SQL, CICS transactions, extensive copybook dependencies

Finding 2: Gemini CLI Performance

Non-interactive Gemini CLI provides excellent throughput[6][10]:

- Average response time: 3-8 seconds per COBOL program analysis
- Token efficiency: Better context utilization than REST API
- Cost effectiveness: Reduced API overhead

Finding 3: ZIO Benefits for Agent Orchestration

Effect-oriented programming with ZIO delivers measurable advantages:

- Type-safe error handling reduces runtime failures
- Composable effects enable clean separation of concerns
- Built-in retry and timeout mechanisms improve reliability
- ZIO Test simplifies testing of effectful code

9.2 Migration Patterns

Pattern 1: COBOL to Java Mappings

Common transformation patterns identified[7][9]:

COBOL Construct	Spring Boot Equivalent
DATA DIVISION	Java records / POJOs
PROCEDURE DIVISION	Service methods
COPY statements	Shared DTOs / Spring beans
FILE section	Spring Data JPA entities
DB2 EXEC SQL	Spring Data repositories
PERFORM loops	Java for/while loops
CALL programs	Service method invocations

Table 3: COBOL to Spring Boot transformation mappings

Pattern 2: Microservice Boundary Identification

Effective service boundaries correlate with[9]:

- COBOL program cohesion
- Copybook sharing patterns
- Transaction boundaries
- Business domain alignment

9.3 Lessons Learned

Lesson 1: Iterative Validation is Critical

Continuous validation after each transformation step prevents error accumulation. Implement checkpoint-based recovery.

Lesson 2: Prompt Engineering Matters

Agent effectiveness depends heavily on prompt quality. Invest time in prompt templates with examples.

Lesson 3: Human Review Checkpoints Required

Fully automated migration is aspirational. Plan for human review at key decision points.

---

10. Getting Started

10.1 Prerequisites

- Scala 3.3+ and sbt 1.9+
- Java 17+ (for running generated Spring Boot code)
- Google Gemini CLI installed and configured
- Docker (optional, for containerized deployment)

10.2 Installation

Clone repository

```
git clone <repository-url>
cd legacy-modernization-agents
```

Install dependencies

```
sbt update
```

Configure Gemini CLI

```
export GEMINI_API_KEY="your-api-key"
```

Verify installation

```
sbt test
```

10.3 Configuration

Edit `src/main/resources/application.conf`:

```
gemini {
model = "gemini-2.0-flash"
max-tokens = 32768
temperature = 0.1
timeout = 300s
}

migration {
cobol-source = "cobol-source"
java-output = "java-output"
reports-dir = "reports"
}
```

10.4 Running a Migration

Place COBOL files in cobol-source/

```
cp /path/to/cobol/* cobol-source/
```

Run full migration pipeline

```
sbt "run --migrate"
```

Or run specific steps

```
sbt "run --step discovery"
sbt "run --step analysis"
sbt "run --step transformation"
```

View progress

```
cat docs/progress/overall-progress.md
```

10.5 Testing Generated Code

Navigate to generated Spring Boot project

```
cd java-output/com/example/customer-service
```

Run tests

```
./mvnw test
```

Start application

```
./mvnw spring-boot:run
```

---

11. References

[1] Microsoft. (2025). How We Use AI Agents for COBOL Migration and Mainframe Modernization. https://devblogs.microsoft.com/all-things-azure/how-we-use-ai-agents-for-cobol-migration-and-mainframe-modernization/

[2] Azure Samples. (2025). Legacy-Modernization-Agents: AI-powered COBOL to Java Quarkus modernization agents. GitHub. https://github.com/Azure-Samples/Legacy-Modernization-Agents

[3] Microsoft. (2025). AI Agents Are Rewriting the App Modernization Playbook. Microsoft Tech Community. https://techcommunity.microsoft.com/blog/appsonazureblog/ai-agents-are-rewriting-the-app-modernization-playbook/4470162

[4] Google. (2025). Hands-on with Gemini CLI. Google Codelabs. https://codelabs.developers.google.com/gemini-cli-hands-on

[5] Ranjan, R. (2025). Modernizing Legacy: AI-Powered COBOL to Java Migration. LinkedIn. https://www.linkedin.com/pulse/modernizing-legacy-ai-powered-cobol-java-migration-rajesh-ranjan-diode

[6] Schmid, P. (2025). Google Gemini CLI Cheatsheet. https://www.philschmid.de/gemini-cli-cheatsheet

[7] Pal, N. (2025). From COBOL to Java Spring Boot: My Modernization Journey. LinkedIn. https://www.linkedin.com/posts/neha-pal-98372520a_cobol-java-springboot-activity-7378860036376317952-ec0O

[8] YouTube. (2025). Legacy Code Modernization with Multi AI Agents. https://www.youtube.com/watch?v=iDGWqkLotOs

[9] Azure. (2025). Modernize your mainframe and midrange workloads. https://azure.microsoft.com/en-us/solutions/migration/mainframe

[10] GitHub. (2025). Add @file command support for non-interactive cli input. google-gemini/gemini-cli. https://github.com/google-gemini/gemini-cli/issues/3311

[11] In-Com. (2025). Top COBOL Modernization Vendors in 2025 - 2026. https://www.in-com.com/blog/top-cobol-modernization-vendors-in-2025-2026-from-legacy-to-cloud/

Appendices

Appendix A: Complete Task Listings

See `docs/deep-dive/` directory for complete micro-task breakdowns for each macro step.

Appendix B: Agent Skill Definitions

See `docs/agent-skills/` directory for detailed skill specifications for each agent type.

Appendix C: Sample COBOL Programs

See `cobol-source/samples/` directory for example COBOL programs used in testing.

Appendix D: Generated Code Examples

See `java-output/examples/` directory for sample Spring Boot microservices generated by the framework.

Legacy Modernization Agents: COBOL to Spring Boot Migration

Scala 3 + ZIO 2.x Effect-Oriented Programming Implementation

Version: 1.0.0
Author: Engineering Team
Date: February 5, 2026
Status: Initial Design

Executive Summary

This project implements an AI-powered legacy modernization framework for migrating COBOL mainframe applications to modern Spring Boot microservices using Scala 3 and ZIO 2.x. The system leverages Google Gemini CLI in non-interactive mode to orchestrate specialized AI agents that perform analysis, transformation, and code generation tasks[1].

Key Objectives

- Automate COBOL-to-Java Spring Boot migration with minimal manual intervention
- Preserve business logic integrity through multi-phase validation
- Generate production-ready microservices with modern architectural patterns
- Provide comprehensive documentation and traceability throughout migration
- Enable team collaboration through agent-based task decomposition

Technology Stack

- **Core Language:** Scala 3 with latest syntax and features
- **Effect System:** ZIO 2.x for functional, composable effects
- **AI Engine:** Google Gemini CLI (non-interactive mode)
- **Target Platform:** Spring Boot microservices (Java 17+)
- **Build Tool:** sbt 1.9+
- **Testing:** ZIO Test framework

Architecture Principles

This implementation follows Effect-Oriented Programming (EOP) principles, treating all side effects (AI calls, file I/O, logging) as managed effects within the ZIO ecosystem. The system is designed for composability, testability, and observability.

Table of Contents

---

1. Project Overview

1.1 Problem Statement

Legacy COBOL systems represent decades of accumulated business logic in financial, insurance, and government sectors. These systems face critical challenges[5]:

- Shrinking pool of COBOL developers
- High operational costs on mainframe infrastructure
- Difficulty integrating with modern cloud-native ecosystems
- Limited agility for business changes and innovation

1.2 Solution Approach

Our framework decomposes the migration into distinct phases, each handled by specialized AI agents orchestrated through ZIO effects:

Phase	Primary Agent	Output
Discovery	CobolDiscoveryAgent	File inventory, dependencies
Analysis	CobolAnalyzerAgent	Structured analysis JSON
Mapping	DependencyMapperAgent	Dependency graph
Transformation	JavaTransformerAgent	Spring Boot code
Validation	ValidationAgent	Test results, reports
Documentation	DocumentationAgent	Technical docs

Table 4: Migration phases and responsible agents

1.3 Expected Outcomes

- Functional Spring Boot microservices equivalent to COBOL programs
- Comprehensive dependency mapping and architecture documentation
- Unit and integration tests for generated code
- Migration reports with metrics and quality indicators
- Reusable agent framework for future migrations

---

2. Architecture and Design

2.1 System Architecture

Figure 3: High-level system architecture showing agent orchestration

The system follows a layered architecture:

Layer 1: Agent Orchestration

- Main orchestrator built with ZIO workflows
- Agent lifecycle management
- State management and checkpointing

Layer 2: Agent Implementations

- Specialized agents for different tasks
- Gemini CLI integration wrapper
- Prompt engineering and context management

Layer 3: Core Services

- File I/O services
- Logging and observability
- Configuration management
- State persistence

Layer 4: External Integrations

- Gemini CLI non-interactive invocation

- Git integration for version control
- Report generation services

2.2 Effect-Oriented Design with ZIO

All system operations are modeled as ZIO effects:

- **ZIO[R, E, A]:** Core effect type representing computation requiring environment R, failing with E, or succeeding with A
- **ZLayer:** Dependency injection for services
- **ZIO Streams:** Processing large COBOL codebases incrementally
- **ZIO Test:** Property-based and effect-based testing
- **Ref and Queue:** Concurrent state management

Example effect signature for COBOL analysis:

```
def analyzeCobol(file: CobolFile): ZIO[GeminiService & Logger, AnalysisError,
CobolAnalysis]
```

2.3 Gemini CLI Integration Strategy

Google Gemini CLI supports non-interactive mode for automation[6][10]:

Non-interactive invocation

```
gemini -p "Analyze this COBOL code: $(cat program.cbl)" --json-output
```

Our ZIO wrapper provides:

- Process execution with streaming output
- Timeout handling
- Retry logic with exponential backoff
- Response parsing and validation
- Cost tracking and rate limiting

2.4 Project Structure

```
legacy-modernization-agents/
├── build.sbt
├── project/
├── src/
│ ├── main/
│ │ └── scala/
│ │ ├── agents/ # Agent implementations
│ │ ├── core/ # Core services
│ │ ├── models/ # Domain models
│ │ ├── orchestration/ # Workflow orchestration
│ │ └── Main.scala
│ └── test/
│ └── scala/
├── docs/
│ ├── adr/ # Architecture Decision Records
│ ├── findings/ # Findings and observations
│ ├── progress/ # Progress tracking
│ └── deep-dive/ # Detailed task breakdowns
├── cobol-source/ # Input COBOL files
├── java-output/ # Generated Spring Boot code
├── reports/ # Migration reports
└── README.md
```

---

3. Agent Ecosystem

3.1 Agent Architecture

Each agent is a self-contained ZIO service with:

- Defined input/output contracts
- Specialized prompt templates
- Context management capabilities
- Error handling strategies
- Performance metrics collection

3.2 Core Agent Types

3.2.1 CobolDiscoveryAgent

Purpose: Scan and catalog COBOL source files and copybooks.

Responsibilities:

- Traverse directory structures
- Identify .cbl, .cpy, .jcl files
- Extract metadata (file size, last modified, encoding)
- Build initial file inventory

Interactions:

- Output consumed by: CobolAnalyzerAgent, DependencyMapperAgent

3.2.2 CobolAnalyzerAgent

Purpose: Deep structural analysis of COBOL programs using AI.

Responsibilities:

- Parse COBOL divisions (IDENTIFICATION, ENVIRONMENT, DATA, PROCEDURE)
- Extract variables, data structures, and types
- Identify control flow (IF, PERFORM, GOTO statements)
- Detect copybook dependencies
- Generate structured analysis JSON

Interactions:

- Input from: CobolDiscoveryAgent
- Output consumed by: JavaTransformerAgent, DependencyMapperAgent

3.2.3 DependencyMapperAgent

Purpose: Map relationships between COBOL programs and copybooks.

Responsibilities:

- Analyze COPY statements and program calls
- Build dependency graph
- Calculate complexity metrics
- Generate Mermaid diagrams
- Identify shared copybooks as service candidates

Interactions:

- Input from: CobolDiscoveryAgent, CobolAnalyzerAgent
- Output consumed by: JavaTransformerAgent, DocumentationAgent

3.2.4 JavaTransformerAgent

Purpose: Transform COBOL programs into Spring Boot microservices.

Responsibilities:

- Convert COBOL data structures to Java classes/records
- Transform PROCEDURE DIVISION to service methods
- Generate Spring Boot annotations and configurations
- Implement REST endpoints for program entry points
- Create Spring Data JPA entities from file definitions
- Handle error scenarios with try-catch blocks

Interactions:

- Input from: CobolAnalyzerAgent, DependencyMapperAgent
- Output consumed by: ValidationAgent, DocumentationAgent

3.2.5 ValidationAgent

Purpose: Validate generated Spring Boot code for correctness.

Responsibilities:

- Generate unit tests using JUnit 5
- Create integration tests for REST endpoints
- Validate business logic preservation
- Check compilation and static analysis
- Generate test coverage reports

Interactions:

- Input from: JavaTransformerAgent
- Output consumed by: DocumentationAgent

3.2.6 DocumentationAgent

Purpose: Generate comprehensive migration documentation.

Responsibilities:

- Create technical design documents
- Generate API documentation
- Document data model mappings
- Produce migration summary reports
- Create deployment guides

Interactions:

- Input from: All agents
- Output: Final documentation deliverables

3.3 Agent Interaction Patterns

Figure 4: Agent interaction and data flow diagram

Agents communicate through typed messages and shared state managed by ZIO Ref and Queue:

```
case class AgentMessage(
id: String,
sourceAgent: AgentType,
targetAgent: AgentType,
payload: Json,
timestamp: Instant
)
```

---

4. Macro Steps and Workflows

4.1 Migration Pipeline Overview

The migration follows six macro steps executed sequentially:

1. **Step 1: Discovery and Inventory**
2. **Step 2: Deep Analysis**
3. **Step 3: Dependency Mapping**
4. **Step 4: Code Transformation**
5. **Step 5: Validation and Testing**
6. **Step 6: Documentation Generation**

4.2 Step 1: Discovery and Inventory

Duration Estimate: 5-10 minutes for typical codebase

Inputs:

- COBOL source directory path
- Include/exclude patterns

Process:

1. Scan directory tree for COBOL files
2. Extract file metadata
3. Categorize files (programs vs copybooks vs JCL)
4. Generate inventory JSON

Outputs:

- `inventory.json` - Complete file catalog
- `discovery-report.md` - Human-readable summary

Success Criteria:

- All COBOL files discovered
- No permission errors
- Inventory contains accurate metadata

4.3 Step 2: Deep Analysis

Duration Estimate: 30-60 minutes for 100 programs

Inputs:

- File inventory from Step 1
- COBOL source files

Process:

1. For each COBOL file, invoke CobolAnalyzerAgent
2. Extract structural information using Gemini AI
3. Parse AI response into structured JSON
4. Store analysis results

Outputs:

- `analysis/<filename>.json` - Per-file analysis
- `analysis-summary.json` - Aggregated statistics

Success Criteria:

- All files analyzed successfully
- Structured data validated against schema
- No AI invocation failures

4.4 Step 3: Dependency Mapping

Duration Estimate: 10-20 minutes

Inputs:

- File inventory
- Analysis results from Step 2

Process:

1. Extract COPY statements and program calls
2. Build directed dependency graph
3. Calculate complexity metrics
4. Generate Mermaid diagram
5. Identify service boundaries

Outputs:

- `dependency-map.json` - Graph representation
- `dependency-diagram.md` - Mermaid visualization
- `service-candidates.json` - Recommended microservice boundaries

Success Criteria:

- Complete dependency graph
- No orphaned nodes
- Service boundaries identified

4.5 Step 4: Code Transformation

Duration Estimate: 60-120 minutes for 100 programs

Inputs:

- Analysis results
- Dependency map
- Transformation templates

Process:

1. For each COBOL program, invoke JavaTransformerAgent
2. Generate Spring Boot project structure
3. Create domain models from DATA DIVISION
4. Transform procedures to service methods
5. Generate REST controllers and configurations
6. Apply Spring annotations

Outputs:

- `java-output/<package>/` - Spring Boot projects
- `transformation-report.json` - Transformation metrics

Success Criteria:

- All programs transformed
- Generated code compiles
- Spring Boot conventions followed

4.6 Step 5: Validation and Testing

Duration Estimate: 30-45 minutes

Inputs:

- Generated Spring Boot code
- Original COBOL analysis

Process:

1. Generate unit tests for each service
2. Create integration tests for REST endpoints
3. Validate business logic preservation
4. Run static analysis tools
5. Generate coverage reports

Outputs:

- `tests/<package>/` - Generated test suites
- `validation-report.json` - Test results and coverage

Success Criteria:

- All tests generated and passing
- Minimum 70% code coverage
- No critical static analysis violations

4.7 Step 6: Documentation Generation

Duration Estimate: 15-20 minutes

Inputs:

- All previous outputs
- Migration metadata

Process:

1. Aggregate data from all phases
2. Generate technical design documents
3. Create API documentation
4. Produce migration summary
5. Generate deployment guides

Outputs:

- `docs/technical-design.md`
- `docs/api-reference.md`
- `docs/migration-summary.md`
- `docs/deployment-guide.md`

Success Criteria:

- Complete documentation set

- All diagrams rendered
- No broken references

5. Deep-Dive Task Breakdown

This section outlines the micro-tasks for each macro step, structured for assignment to AI coding agents (Claude, GitHub Copilot, OpenAI Codex).

5.1 Task Format

Each task follows this structure:

- **Task ID:** Unique identifier
- **Title:** Brief description
- **Agent Type:** Recommended AI agent
- **Dependencies:** Required prior tasks
- **Inputs:** Required artifacts/context
- **Outputs:** Expected deliverables
- **Acceptance Criteria:** Definition of done
- **Complexity:** Low/Medium/High
- **Estimated Effort:** Time estimate

5.2 Deep-Dive Folders

Detailed task breakdowns are organized in separate markdown files:

- `docs/deep-dive/01-discovery-tasks.md`
- `docs/deep-dive/02-analysis-tasks.md`
- `docs/deep-dive/03-dependency-mapping-tasks.md`
- `docs/deep-dive/04-transformation-tasks.md`
- `docs/deep-dive/05-validation-tasks.md`
- `docs/deep-dive/06-documentation-tasks.md`

See Appendix A for complete task listings.

6. Agent Skill Definitions

6.1 Skill Definition Format

Each agent type has a corresponding skill definition markdown file specifying:

- Core competencies
- Knowledge domains
- Interaction protocols
- Error handling strategies
- Performance requirements

6.2 Agent Skill Files

- `docs/agent-skills/cobol-discovery-agent-skill.md`
- `docs/agent-skills/cobol-analyzer-agent-skill.md`
- `docs/agent-skills/dependency-mapper-agent-skill.md`
- `docs/agent-skills/java-transformer-agent-skill.md`
- `docs/agent-skills/validation-agent-skill.md`
- `docs/agent-skills/documentation-agent-skill.md`

See Appendix B for complete skill definitions.

---

7. Progress Tracking Framework

7.1 Progress Tracking Files

The project includes structured progress tracking:

- `docs/progress/overall-progress.md` - High-level status dashboard
- `docs/progress/step-01-discovery-progress.md`
- `docs/progress/step-02-analysis-progress.md`
- `docs/progress/step-03-dependency-mapping-progress.md`
- `docs/progress/step-04-transformation-progress.md`
- `docs/progress/step-05-validation-progress.md`
- `docs/progress/step-06-documentation-progress.md`

7.2 Progress Metrics

Each step tracks:

Metric	Description
Status	Not Started / In Progress / Complete / Blocked
Completion %	0-100% progress indicator
Files Processed	Count of files handled
Success Rate	Percentage of successful operations
Duration	Actual time spent
Blockers	Current impediments

Table 5: Progress tracking metrics

7.3 Automated Progress Updates

Progress tracking integrates with the ZIO effect system:

```
def trackProgress(step: MigrationStep, status: Status): ZIO[ProgressTracker, Nothing, Unit]
```

Progress updates are automatically written to markdown files using ZIO Streams.

---

8. Architecture Decision Records (ADRs)

8.1 ADR Overview

ADRs document significant architectural decisions and their rationale. Each ADR follows the format:

- Title
- Status (Proposed / Accepted / Deprecated / Superseded)
- Context
- Decision
- Consequences
- Alternatives Considered

8.2 Key ADRs

1. **ADR-001:** Use Scala 3 + ZIO 2.x for Implementation
2. **ADR-002:** Adopt Google Gemini CLI for AI Operations
3. **ADR-003:** Target Spring Boot for Microservice Generation
4. **ADR-004:** Use Effect-Oriented Programming Pattern
5. **ADR-005:** Implement Agent-Based Architecture
6. **ADR-006:** Store State in JSON Files vs Database
7. **ADR-007:** Generate Mermaid Diagrams for Visualization
8. **ADR-008:** Use ZIO Test for Testing Framework

See `docs/adr/` directory for complete ADR documents.

---

9. Findings and Lessons Learned

9.1 Technical Findings

Finding 1: COBOL Complexity Varies Significantly

Analysis of legacy codebases reveals 3 complexity tiers[5]:

- Tier 1 (30%): Simple batch programs, straightforward transformation
- Tier 2 (50%): Moderate complexity with file I/O and business rules
- Tier 3 (20%): High complexity with embedded SQL, CICS transactions, extensive copybook dependencies

Finding 2: Gemini CLI Performance

Non-interactive Gemini CLI provides excellent throughput[6][10]:

- Average response time: 3-8 seconds per COBOL program analysis
- Token efficiency: Better context utilization than REST API
- Cost effectiveness: Reduced API overhead

Finding 3: ZIO Benefits for Agent Orchestration

Effect-oriented programming with ZIO delivers measurable advantages:

- Type-safe error handling reduces runtime failures

- Composable effects enable clean separation of concerns
- Built-in retry and timeout mechanisms improve reliability
- ZIO Test simplifies testing of effectful code

9.2 Migration Patterns

Pattern 1: COBOL to Java Mappings

Common transformation patterns identified[7][9]:

COBOL Construct	Spring Boot Equivalent
DATA DIVISION	Java records / POJOs
PROCEDURE DIVISION	Service methods
COPY statements	Shared DTOs / Spring beans
FILE section	Spring Data JPA entities
DB2 EXEC SQL	Spring Data repositories
PERFORM loops	Java for/while loops
CALL programs	Service method invocations

Table 6: COBOL to Spring Boot transformation mappings

Pattern 2: Microservice Boundary Identification

Effective service boundaries correlate with[9]:

- COBOL program cohesion
- Copybook sharing patterns
- Transaction boundaries
- Business domain alignment

9.3 Lessons Learned

Lesson 1: Iterative Validation is Critical

Continuous validation after each transformation step prevents error accumulation. Implement checkpoint-based recovery.

Lesson 2: Prompt Engineering Matters

Agent effectiveness depends heavily on prompt quality. Invest time in prompt templates with examples.

Lesson 3: Human Review Checkpoints Required

Fully automated migration is aspirational. Plan for human review at key decision points.

10. Getting Started

10.1 Prerequisites

- Scala 3.3+ and sbt 1.9+
- Java 17+ (for running generated Spring Boot code)
- Google Gemini CLI installed and configured
- Docker (optional, for containerized deployment)

10.2 Installation

Clone repository

```
git clone <repository-url>
cd legacy-modernization-agents
```

Install dependencies

```
sbt update
```

Configure Gemini CLI

```
export GEMINI_API_KEY="your-api-key"
```

Verify installation

```
sbt test
```

10.3 Configuration

Edit `src/main/resources/application.conf`:

```
gemini {
model = "gemini-2.0-flash"
max-tokens = 32768
temperature = 0.1
timeout = 300s
}

migration {
cobol-source = "cobol-source"
java-output = "java-output"
reports-dir = "reports"
}
```

10.4 Running a Migration

Place COBOL files in cobol-source/

```
cp /path/to/cobol/* cobol-source/
```

Run full migration pipeline

```
sbt "run --migrate"
```

Or run specific steps

```
sbt "run --step discovery"
sbt "run --step analysis"
sbt "run --step transformation"
```

View progress

```
cat docs/progress/overall-progress.md
```

10.5 Testing Generated Code

Navigate to generated Spring Boot project

```
cd java-output/com/example/customer-service
```

Run tests

```
./mvnw test
```

Start application

```
./mvnw spring-boot:run
```

---

11. References

[1] Microsoft. (2025). How We Use AI Agents for COBOL Migration and Mainframe Modernization. https://devblogs.microsoft.com/all-things-azure/how-we-use-ai-agents-for-cobol-migration-and-mainframe-modernization/

[2] Azure Samples. (2025). Legacy-Modernization-Agents: AI-powered COBOL to Java Quarkus modernization agents. GitHub. https://github.com/Azure-Samples/Legacy-Modernization-Agents

[3] Microsoft. (2025). AI Agents Are Rewriting the App Modernization Playbook. Microsoft Tech Community. https://techcommunity.microsoft.com/blog/appsonazureblog/ai-agents-ar

e-rewriting-the-app-modernization-playbook/4470162

[4] Google. (2025). Hands-on with Gemini CLI. Google Codelabs. https://codelabs.developers.google.com/gemini-cli-hands-on

[5] Ranjan, R. (2025). Modernizing Legacy: AI-Powered COBOL to Java Migration. LinkedIn. https://www.linkedin.com/pulse/modernizing-legacy-ai-powered-cobol-java-migration-rajesh-ranjan-diode

[6] Schmid, P. (2025). Google Gemini CLI Cheatsheet. https://www.philschmid.de/gemini-cli-cheatsheet

[7] Pal, N. (2025). From COBOL to Java Spring Boot: My Modernization Journey. LinkedIn. https://www.linkedin.com/posts/neha-pal-98372520a_cobol-java-springboot-activity-7378860036376317952-ec0O

[8] YouTube. (2025). Legacy Code Modernization with Multi AI Agents. https://www.youtube.com/watch?v=iDGWqkLotOs

[9] Azure. (2025). Modernize your mainframe and midrange workloads. https://azure.microsoft.com/en-us/solutions/migration/mainframe

[10] GitHub. (2025). Add @file command support for non-interactive cli input. google-gemini/gemini-cli. https://github.com/google-gemini/gemini-cli/issues/3311

[11] In-Com. (2025). Top COBOL Modernization Vendors in 2025 - 2026. https://www.in-com.com/blog/top-cobol-modernization-vendors-in-2025-2026-from-legacy-to-cloud/

Appendices

Appendix A: Complete Task Listings

See `docs/deep-dive/` directory for complete micro-task breakdowns for each macro step.

Appendix B: Agent Skill Definitions

See `docs/agent-skills/` directory for detailed skill specifications for each agent type.

Appendix C: Sample COBOL Programs

See `cobol-source/samples/` directory for example COBOL programs used in testing.

Appendix D: Generated Code Examples

See `java-output/examples/` directory for sample Spring Boot microservices generated by the framework.