Legacy Application Modernization Through Behavioral Understanding

An AI-Assisted Approach Using Finite State Machine Extraction

Bob Harwood

Enterprise Architecture | Farmers Insurance Group

December 2025

Table of Contents

Executive Summary

Farmers Insurance maintains a substantial legacy application portfolio: approximately 40 million lines of COBOL, plus significant investments in Visual Basic, .NET Framework applications, and Java systems with IBM WebSphere dependencies. This code—representing decades of accumulated business logic—processes over $20 billion in annual premiums and cannot fail. Traditional modernization approaches—whether lift-and-shift, manual rewrite, or commercial translation services—have proven inadequate for organizations of our scale and complexity.

This proposal presents a novel approach: rather than translating code syntax, we extract and preserve the behavioral semantics of legacy systems as Finite State Machines (FSMs). This behavioral representation serves as a platform-independent specification that enables true forward engineering to modern architectures—not syntactic translation that produces 'JOBOL' (Java that mimics COBOL structure) or equivalent anti-patterns in other target languages.

The approach leverages recent advances in Large Language Models (LLMs) combined with:

- Retrieval-Augmented Generation (RAG) to achieve FSM extraction accuracy exceeding 90%
- Grounding the extracted FSMs in our enterprise domain ontology—specifically our value streams, business capabilities, and domain events—so we can forward-engineer to bounded contexts that align with how our business operates

Scope Boundary

This plan focuses on reducing human effort to produce functionally correct modernized code slices. The following scope boundaries apply:

- In scope: Behavioral specification extraction (FSMs), semantic enrichment and domain mapping, forward code generation, unit-level and flow-level behavioral verification for each modernized slice
- Out of scope: Persistent data migration, historical data reconciliation, cross-system side-effects alignment (e.g., MQ delivery SLAs, checkpoint/restart semantics). These require separate workstreams with their own plans and gates.

1. The Problem: Why Direct Translation Fails at Scale

1.1 The Mathematics of Compound Error

Every code translation approach has a per-unit accuracy rate—the probability that any single translatable unit (procedure, paragraph, or statement) is correctly converted. Even impressive accuracy rates become catastrophic at enterprise scale.

Consider the compound probability equation:

$$P(\text{correct\_program}) = P(\text{correct\_unit})^n$$

Where n represents the number of translatable units. The following graph illustrates why even 99% per-unit accuracy is insufficient:

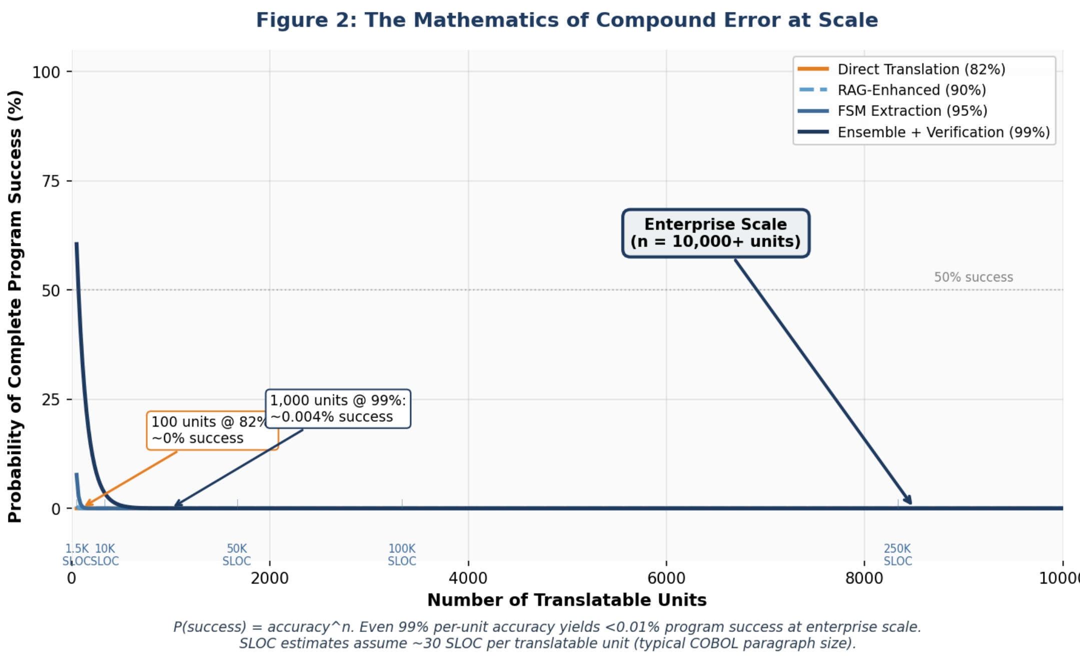

Figure 2: The Mathematics of Compound Error at Scale

For a system with 10,000 translatable units—modest by enterprise standards—even 99% per-unit accuracy yields a probability of complete correctness approaching zero. IBM's own evaluation framework documentation acknowledges this explicitly: 'if a program has 10 paragraphs and each is translated with 90% accuracy, the probability of a fully correct program is only about 34%' [1]. Our legacy estate contains millions of such units across COBOL, VB, .NET, and WebSphere Java. Direct translation is mathematically untenable.

1.2 How Direct Translation Works

Direct code translation—the approach used by most commercial modernization tools—operates by parsing source code and generating equivalent target language constructs at multiple granularities:

- Token-level translation: Keywords and operators are mapped (MOVE → assignment, PERFORM → method call)
- Statement-level translation: Individual COBOL statements become Java statements, often with runtime library support for COBOL-specific semantics
- Paragraph-level translation: COBOL paragraphs become Java methods, preserving PERFORM...THRU semantics through generated control flow
- Program-level translation: Entire COBOL programs become Java classes, with WORKING-STORAGE mapped to instance fields

Modern LLM-based approaches improve on rule-based translators by leveraging learned patterns from training corpora. However, they fundamentally operate the same way: they predict target language tokens conditioned on source language tokens, producing translations unit-by-unit.

Recent open-source efforts like Microsoft's Legacy Modernization Agents [16] demonstrate the state of agentic translation: multi-agent pipelines using Semantic Kernel that analyze COBOL, map dependencies, and generate Java or C# code. These tools provide valuable dependency visualization and structured analysis, but remain fundamentally direct-translation approaches—they lack formal behavioral specification, domain ontology alignment, and verification beyond functional testing.

Why Translation Has Lower Success Rates Than Specification Extraction

Translation requires preserving both syntax AND semantics simultaneously at every step. A single mistranslation—a wrong operator, an off-by-one loop bound, a mishandled null condition—produces a complete program that executes but produces wrong results. The compound error problem compounds these individual failures across thousands of units.

Specification extraction separates these concerns. We first extract WHAT the code does (the FSM), verify that extraction, then generate code that implements the specification. Each step can be validated independently, and errors in one step don't necessarily cascade.

Why Vendors Don't Use FSM Extraction

Several factors explain why commercial vendors optimize for direct translation rather than specification extraction:

- Training data availability: LLMs are trained predominantly on source code, not FSM-code pairs. GitHub contains billions of lines of Java but almost no paired COBOL-FSM datasets.
- Customer expectations: Enterprises often request 'Java that does what our COBOL does'—a syntactic framing. FSM extraction requires accepting an intermediate representation before seeing target code.
- Tooling maturity: Code translation tools have decades of development. FSM extraction from imperative code is an active research area with fewer production-ready tools.
- Verification complexity: Proving extracted FSMs capture source behavior, then proving generated code implements the FSM, requires formal methods expertise most vendors lack.

1.3 Commercial Service Approaches

Major vendors offer legacy modernization services. Understanding their approaches—and why they've chosen direct translation—illuminates both the state of the art and the gap we propose to fill.

Why Vendors Use Direct Translation

Commercial modernization tools predominantly use direct translation because it aligns with market expectations and available training data. Enterprises typically request 'Java that does what our COBOL does'—a syntactic framing that translation directly addresses. LLMs are trained on billions of lines of source code, making code-to-code translation a natural fit. The compound error problem (Section 1.1) is managed through human-in-the-loop validation rather than eliminated through alternative approaches. This is a rational business decision: translation ships faster, requires less novel research, and produces visible output quickly—even if that output requires substantial manual correction.

IBM watsonx Code Assistant for Z

IBM's offering uses a fine-tuned 20-billion parameter LLM for COBOL-to-Java transformation. IBM explicitly states their approach 'adopts a two-phase, semantically driven approach' rather than 'line-by-line translations' [1]. The system includes Application Discovery and Delivery Intelligence (ADDI) for understanding application structure. However, IBM's own evaluation framework acknowledges that 'manual validation of translated Java code from COBOL [is] a necessary but time-consuming and labor-intensive process' [1]. At Farmers' scale—millions of lines across interconnected systems—this human validation bottleneck remains problematic.

AWS Mainframe Modernization and AWS Transform

AWS offers multiple modernization paths. AWS Transform for Mainframe uses agentic AI to accelerate modernization 'from years to months,' handling analysis, planning, code refactoring, and migration [2]. Blu Age Analyzer supports Domain-Driven Design decomposition—AWS documentation states it 'automates domain discovery by analyzing data and call dependencies between legacy programs' to identify bounded contexts [2]. However, this decomposition operates post-translation: the domain analysis happens after code has been transformed to Java, not before. The transformation itself doesn't benefit from domain understanding. Additionally, generated Java relies on Blu Age runtime libraries for functional equivalence, creating AWS infrastructure dependencies.

The Gap We Address

Both vendors have made significant advances. Our approach differs by proposing a different sequence: we extract behavioral specifications first, map them to our domain ontology, decompose into bounded contexts based on business capability alignment, and only then generate modern code. This domain-first approach means bounded context boundaries emerge from business understanding rather than code structure analysis, and the intermediate FSM representation enables formal verification of behavioral equivalence.

Why We Propose an Alternative

Our fundamental concern is the sequence of operations. In translation-first approaches, architectural decisions are made after code generation—working backwards from translated code to identify domain boundaries. In our FSM-first approach, we understand behavior before generating any target code, allowing domain decomposition to drive the architecture rather than react to it. This matters for Farmers because our business capabilities and value streams are well-defined; we want modernization to align with our enterprise architecture, not require retrofitting.

1.4 What We Actually Need

Farmers requires an approach that:

- Extracts behavioral semantics, not just syntax
- Produces platform-independent specifications that can target any modern architecture
- Integrates with our enterprise domain model (value streams, capabilities, domain events)
- Enables decomposition into bounded contexts aligned with business organization
- Achieves sufficient accuracy to minimize human intervention at scale

2. The Foundational Approach: Behavioral Understanding Through FSMs

Our approach treats legacy code not as text to translate but as behavior to understand. We extract Finite State Machines that capture what the code does—its states, transitions, and conditions—independent of how it's implemented.

2.1 Why Finite State Machines?

FSMs are a natural representation for business processes. A policy moves through states (Quoted → Bound → Active → Renewed/Cancelled). A claim progresses (Filed → Assigned → Investigated → Approved/Denied → Paid). These state machines are implicit in COBOL code—our task is to make them explicit.

FSMs provide several critical properties:

- Platform independence: An FSM describes behavior without prescribing implementation
- Formal verifiability: FSM equivalence can be mathematically proven
- Domain alignment: Business stakeholders understand states and transitions
- Composability: Complex behaviors emerge from FSM composition

2.2 Research Foundation

This approach builds on recent advances in using LLMs for program understanding and specification extraction. Key papers include:

ProtocolGPT [4]: Demonstrates FSM extraction from protocol implementations with greater than 90% precision. The key insight: LLMs can identify state patterns even without explicit state declarations by analyzing behavioral sequences. Available at: arxiv.org/abs/2405.00393

FlowFSM [5]: Extends FSM extraction using an agentic framework with prompt chaining and chain-of-thought reasoning. Their hierarchical decomposition approach provides a tractable path through large codebases. Available at: arxiv.org/abs/2507.11222

SpecGPT [6]: Achieves 91.14% F1-score on protocol state machine extraction from 3GPP specifications using domain-informed prompting and LLM ensembles. Most relevant is their handling of complex specifications through chain-of-thought reasoning. Available at: arxiv.org/abs/2510.14348

Zhang et al. [7]: At ICSE 2024's LLM4Code Workshop demonstrated 81.99% accuracy on direct COBOL-to-Java translation using refinement-based strategies. This provides our baseline: our FSM-mediated approach must exceed this to justify the additional complexity.

2.3 The Extraction Pipeline

Our pipeline combines static analysis, RAG-enhanced LLM extraction, and formal verification:

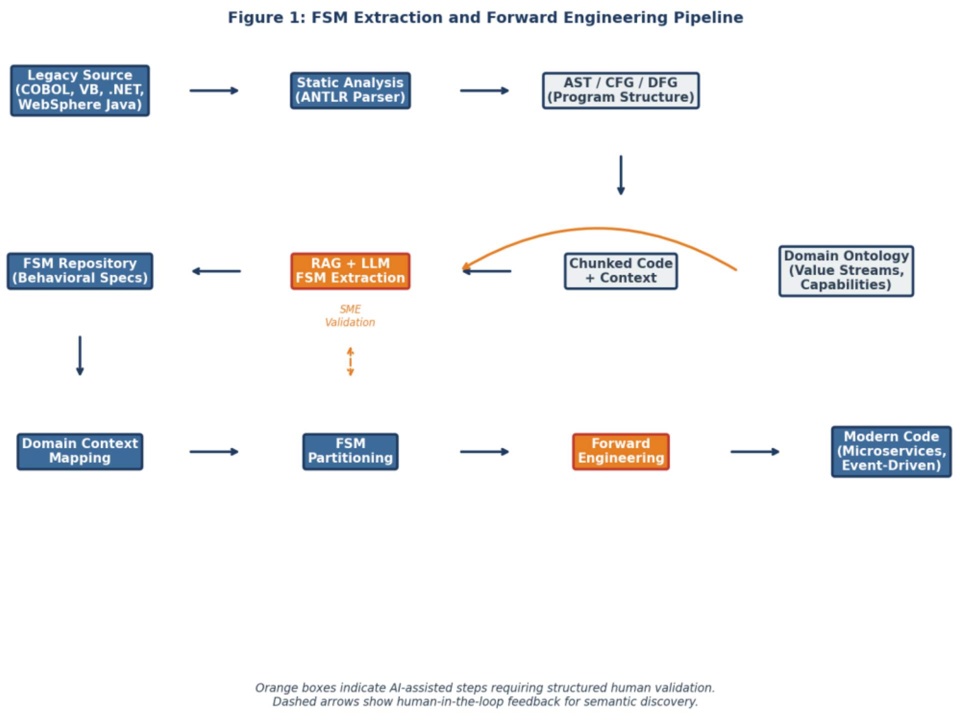
Figure 1: FSM Extraction and Forward Engineering Pipeline
Legacy Source (COBOL, VB, .NET, WebSphere Java)
Static Analysis (ANTLR Parser)
AST / CFG / DFG (Program Structure)
Domain Ontology (Value Streams, Capabilities)
Chunked Code + Context
RAG + LLM FSM Extraction
SME Validation
FSM Repository (Behavioral Specs)
Domain Context Mapping
FSM Partitioning
Forward Engineering
Modern Code (Microservices, Event-Driven)
Orange boxes indicate AI-assisted steps requiring structured human validation.
Dashed arrows show human-in-the-loop feedback for semantic discovery.

Figure 1: FSM Extraction and Forward Engineering Pipeline

2.4 RAG Architecture for Legacy Code Understanding

Standard LLMs lack deep expertise in legacy languages and enterprise-specific conventions. We augment them using Retrieval-Augmented Generation with a specialized knowledge base that must be constructed as part of the modernization effort:

```
class LegacyKnowledgeBase:
    def __init__(self):
        self.embeddings = OpenAIEmbeddings()
        self.vectorstore = FAISS.load_local('legacy_knowledge')

        # Knowledge categories - sources noted
        self.sources = {
            'language': 'Language specs (COBOL-85, VB6, etc.) - publicly available',
            'patterns': 'Platform patterns (CICS, WebSphere, etc.) - vendor docs',
            'domain': 'Insurance terminology - internal glossaries, must be curated',
            'codebase': 'Existing docs, copybooks, config - extracted from SCM',
            'tribal': 'SME interviews, retirement knowledge capture - MUST BUILD'
        }

    def retrieve_context(self, code_chunk: str, query_type: str) -> List[Document]:
        """Retrieve relevant context for FSM extraction."""
        query = f"{query_type}: {self._summarize(code_chunk)}"
        return self.vectorstore.similarity_search(query, k=5)
```

Knowledge Base Construction

The knowledge base does not exist today and must be built. Sources include: (1) Language specifications and platform documentation, publicly available; (2) Existing program documentation, copybooks, JCL, and configuration files extracted from source control; (3) Insurance domain glossaries and business process documentation from internal repositories; (4) Critically, tribal knowledge from retiring SMEs—this requires active knowledge capture before institutional memory is lost. The knowledge base construction is a prerequisite investment, not an assumed asset.

The Cryptic Identifier Challenge

Much of our legacy code was written in the 1970s when memory was expensive and identifier lengths were severely limited. The readable example below is aspirational; reality looks more like this:

```
* Realistic 1970s-era COBOL with cryptic identifiers
P1200.
    EVALUATE TRUE
        WHEN W3-S1 = '1'
            PERFORM P1210
            IF W3-F7 = 'Y'
                MOVE '2' TO W3-S1
        WHEN W3-S1 = '2'
            PERFORM P1220
            MOVE '3' TO W3-S1
        WHEN W3-S1 = '3'
            PERFORM P1230
    END-EVALUATE.
```

Without a data dictionary—and we've lost ours in more than one system—W3-S1 might be 'claim status' or 'policy state' or something else entirely. The FSM extraction can identify THAT there's a state machine with states '1', '2', '3' and specific transitions. But semantic meaning requires either: (1) surviving documentation in the knowledge base, (2) inference from usage patterns across the codebase, or (3) human-in-the-loop confirmation. This is where current vendors require substantial manual effort, and where we propose investing in systematic knowledge capture.

The LLM, augmented with whatever domain context is available, extracts the structural FSM:

```
{
  "fsm": "P1200_StateMachine",
  "states": ["1", "2", "3"],
  "semantic_guess": "Possibly claim or policy lifecycle - NEEDS VERIFICATION",
  "transitions": [
    {"from": "1", "to": "2", "trigger": "P1210_complete",
     "action": "P1210", "guard": "W3-F7 = 'Y'"},
    {"from": "2", "to": "3", "trigger": "P1220_complete", "action": "P1220"},
    {"from": "3", "trigger": "P1230_invoked", "action": "P1230"}
  ],
  "confidence": 0.72,
  "requires_sme_review": true
}
```

2.5 Handling COBOL Complexity

COBOL presents unique challenges that our pipeline specifically addresses:

Copybooks and Data Structures

COBOL copybooks define shared data structures across programs. We parse and index all copybooks, building a data dictionary that informs FSM extraction. When the LLM encounters CUSTOMER-RECORD, it retrieves the full structure definition.

```
class CopybookResolver:
    def __init__(self, copybook_dir: str):
        self.copybooks = {}
        self.parser = COBOLParser()  # ANTLR-based
        self._load_copybooks(copybook_dir)

    def resolve(self, code: str) -> str:
        """Expand COPY statements with actual copybook content."""
        pattern = r'COPY\s+(\w+)\.'
        for match in re.finditer(pattern, code):
            name = match.group(1)
            if name in self.copybooks:
                code = code.replace(match.group(0), self.copybooks[name])
        return code
```

CICS Transaction Processing

CICS programs have implicit state across pseudo-conversational interactions. We track COMMAREA contents and HANDLE conditions to reconstruct the full transaction FSM:

```
* CICS pseudo-conversational pattern
PROCEDURE DIVISION.
    EVALUATE EIBCALEN
        WHEN 0
            PERFORM INITIAL-ENTRY
        WHEN OTHER
            MOVE DFHCOMMAREA TO WS-COMMAREA
            EVALUATE WS-PROCESS-STATE
                WHEN 'MENU'
                    PERFORM PROCESS-MENU-SELECTION
                WHEN 'SEARCH'
                    PERFORM PROCESS-SEARCH-RESULTS
                WHEN 'DETAIL'
                    PERFORM PROCESS-DETAIL-ACTION
            END-EVALUATE
    END-EVALUATE.
```

This pattern is recognized and extracted as a navigation FSM with states corresponding to screen contexts.

Batch Processing Flows

Large batch programs often implement complex multi-phase processing. We use Data Flow Graph analysis to identify phase boundaries:

```
def identify_batch_phases(cfg: ControlFlowGraph, dfg: DataFlowGraph) -> List[Phase]:
    """Identify processing phases in batch COBOL programs."""
    phases = []

    # Phases typically bounded by file operations
    file_ops = dfg.find_nodes(type=['OPEN', 'CLOSE', 'READ', 'WRITE'])

    # Group operations by file and sequence
    file_sequences = group_by_file(file_ops)

    for file_id, ops in file_sequences.items():
        # Each OPEN...CLOSE span is a potential phase
        for open_op, close_op in find_pairs(ops, 'OPEN', 'CLOSE'):
            phase = extract_phase(cfg, dfg, open_op, close_op)
            phases.append(phase)

    return phases
```

Dead Code and Copy-Paste Proliferation

Decades of maintenance have left our codebase riddled with dead code paths, commented-out sections, and copy-paste variations. Before FSM extraction can be meaningful, we must identify and eliminate these artifacts:

```python
def identify_dead_code(cfg: ControlFlowGraph, call_graph: CallGraph) -> DeadCodeReport:
    """Identify unreachable code, unused variables, and orphaned modules."""
    report = DeadCodeReport()

    # Unreachable basic blocks
    reachable = cfg.compute_reachable_from_entry()
    report.unreachable_blocks = cfg.all_blocks - reachable

    # Unused paragraphs/procedures (no callers in call graph)
    for node in call_graph.nodes:
        if call_graph.in_degree(node) == 0 and not node.is_entry_point:
            report.orphan_procedures.append(node)

    # Clone detection for copy-paste analysis
    report.clones = detect_code_clones(cfg, similarity_threshold=0.85)

    return report
```

Copy-paste reuse is particularly problematic: the 'same' logic may exist in dozens of places with subtle variations. Our approach identifies these clones, extracts a canonical FSM for the pattern, then maps variations back to it—enabling consistent modernization rather than translating each copy independently.

3. Forward Engineering to Modern Architecture

Extracted FSMs are behavioral specifications, not implementation. The forward engineering phase transforms these specifications into modern architectures—avoiding the 'JOBOL' trap that plagues direct translation.

3.1 The JOBOL Problem

Direct code translation—whether manual or automated—typically produces Java (or Python, or Go) that mimics the source structure: long procedural methods, global data access, implicit state scattered across variables. This 'JOBOL' (or its VB, .NET, or WebSphere equivalents) provides no architectural benefit. It merely relocates the legacy problem to a new runtime.

True modernization requires decomposition. A 50,000-line program processing policies should become multiple bounded contexts: Quote Management, Underwriting, Policy Issuance, Billing. Each context should be independently deployable, with clear interfaces and domain events.

3.2 Domain-Driven Decomposition Using Our Ontology

Farmers has invested in defining our enterprise domain model through the Observability Ontology [8]. This ontology specifies value streams, business capabilities, and domain events. We leverage this directly for bounded context identification.

Value Stream Alignment

Our ontology defines the Insurance Product & Service Lifecycle value stream with capabilities organized in three tiers:

Tier	Capabilities
Upstream	Product Strategy, Market Research, Actuarial Analysis, Product Development
Midstream	Quote Generation, Underwriting, Policy Issuance, Endorsement Processing, Billing, Premium Collection
Downstream	FNOL, Claim Assignment, Investigation, Adjudication, Fraud Detection, Payment Processing

Extracted FSMs are automatically classified to capabilities based on their states, transitions, and data access patterns:

```
def classify_fsm_to_capability(fsm: FSM, ontology: DomainOntology) -> Capability:
    """Map extracted FSM to enterprise capability."""

    # Analyze FSM characteristics
    state_terms = extract_domain_terms(fsm.states)
    transition_terms = extract_domain_terms(fsm.transitions)
    data_entities = fsm.referenced_entities

    # Score against each capability
    scores = {}
    for capability in ontology.capabilities:
        score = (
            term_similarity(state_terms, capability.vocabulary) * 0.3 +
            term_similarity(transition_terms, capability.actions) * 0.3 +
            entity_overlap(data_entities, capability.entities) * 0.4
        )
        scores[capability] = score

    return max(scores, key=scores.get)
```

Domain Events from FSM Transitions

Our ontology defines canonical domain events: PolicyQuoteCreated, PolicyBound, ClaimFiled, ClaimApproved, etc. FSM transitions map to these events:

```
def extract_domain_events(fsm: FSM, ontology: DomainOntology) -> List[DomainEvent]:
    """Transform FSM transitions into domain events."""
    events = []

    for transition in fsm.transitions:
        # Match transition to canonical event
        event_type = ontology.match_event(
            from_state=transition.from_state,
            to_state=transition.to_state,
            action=transition.action
        )

        if event_type:
            event = DomainEvent(
                type=event_type,
                source_fsm=fsm.name,
                payload_schema=infer_payload(transition),
                capability=fsm.capability
            )
            events.append(event)

    return events
```

This produces event-driven architectures where services communicate through well-defined domain events rather than procedural calls.

Handling Cross-Context FSMs

Legacy programs routinely violate bounded context boundaries—a single COBOL program might handle quoting, underwriting, AND billing in one monolithic flow. The extracted FSM will reflect this reality: it will contain states and transitions that span what SHOULD be separate contexts.

Our decomposition strategy handles this through FSM partitioning:

```
def partition_cross_context_fsm(fsm: FSM, ontology: DomainOntology) -> List[FSMFragment]:
    """Split FSM that spans multiple capabilities into context-aligned fragments."""

    # Classify each state to its most likely capability
    state_assignments = {}
    for state in fsm.states:
        state_assignments[state] = classify_state_to_capability(state, fsm, ontology)

    # Identify context-crossing transitions (these become integration points)
    crossing_transitions = []
    for transition in fsm.transitions:
        if state_assignments[transition.from_state] !=
state_assignments[transition.to_state]:
            crossing_transitions.append(transition)

    # Partition into fragments, converting crossings to domain events
    fragments = []
    for capability in set(state_assignments.values()):
        fragment_states = [s for s, c in state_assignments.items() if c == capability]
        fragment = FSMFragment(
            capability=capability,
            states=fragment_states,
            internal_transitions=[t for t in fsm.transitions
                                  if t.from_state in fragment_states
                                  and t.to_state in fragment_states],
            inbound_events=[crossing_to_event(t) for t in crossing_transitions
                           if t.to_state in fragment_states],
            outbound_events=[crossing_to_event(t) for t in crossing_transitions
                             if t.from_state in fragment_states]
        )
        fragments.append(fragment)

    return fragments
```

The key insight: transitions that cross capability boundaries become domain events in the modernized architecture. What was a PERFORM or CALL becomes a published event that the downstream context subscribes to. This transforms tightly-coupled procedural code into loosely-coupled event-driven services—but it requires human validation that the partitioning matches business intent.

3.3 Bounded Context Identification

Bounded contexts emerge from analyzing FSM clusters, data affinity, and transaction boundaries:

```
def identify_bounded_contexts(fsms: List[FSM], ontology: DomainOntology) ->
List[BoundedContext]:
    """Group FSMs into bounded contexts based on cohesion metrics."""

    # Build affinity graph
    affinity = nx.Graph()
    for fsm in fsms:
        affinity.add_node(fsm.name, capability=fsm.capability)

    for fsm_a, fsm_b in combinations(fsms, 2):
        # Calculate cohesion score
        score = (
            data_coupling(fsm_a, fsm_b) * 0.4 +      # Shared data entities
            capability_alignment(fsm_a, fsm_b) * 0.3 + # Same capability
            transaction_coupling(fsm_a, fsm_b) * 0.3   # Same transaction boundary
        )
        if score > COHESION_THRESHOLD:
            affinity.add_edge(fsm_a.name, fsm_b.name, weight=score)

    # Detect communities (bounded contexts)
    communities = nx.community.louvain_communities(affinity)

    return [BoundedContext(
        fsms=[fsms[n] for n in community],
        capability=dominant_capability(community),
        aggregate_roots=identify_aggregates(community)
    ) for community in communities]
```

3.4 Aggregate Root Detection

Within each bounded context, we identify aggregate roots—entities that are modified together and should be persisted atomically:

```
def identify_aggregates(fsms: List[FSM]) -> List[AggregateRoot]:
    """Find aggregate roots from data modification patterns."""

    # Build entity modification graph from DFG
    mod_graph = nx.DiGraph()
    for fsm in fsms:
        for transition in fsm.transitions:
            entities_modified = transition.data_writes
            for entity in entities_modified:
                mod_graph.add_node(entity)
            # Entities modified together have edges
            for e1, e2 in combinations(entities_modified, 2):
                mod_graph.add_edge(e1, e2)

    # Find strongly connected components
    aggregates = []
    for component in nx.strongly_connected_components(mod_graph):
        root = find_root_entity(component)  # Entity with most references
        aggregates.append(AggregateRoot(
            root=root,
            members=component - {root}
        ))

    return aggregates
```

For example, this analysis might reveal that Policy, Coverage, and Endorsement form an aggregate with Policy as the root—they're always modified together in COBOL code and should remain transactionally consistent in the modernized system.

3.5 Modern Architecture Generation

With bounded contexts, aggregates, and domain events identified, we generate modern architectures:

Event Sourcing from FSMs

FSM transitions naturally map to event sourcing: each transition becomes a persisted event, and current state is derived by replaying events.

```
class PolicyAggregate:
    """Generated from extracted FSM + aggregate analysis."""

    def __init__(self, policy_id: str):
        self.policy_id = policy_id
        self.events: List[DomainEvent] = []
        self._state = PolicyState.DRAFT

    def bind(self, binding_data: BindingData) -> None:
        """Transition from FSM: QUOTED -> BOUND."""
        if self._state != PolicyState.QUOTED:
            raise InvalidTransition(self._state, 'bind')

        event = PolicyBound(
            policy_id=self.policy_id,
            effective_date=binding_data.effective_date,
            premium=binding_data.premium,
            timestamp=datetime.utcnow()
        )
        self._apply(event)
        self.events.append(event)

    def _apply(self, event: DomainEvent) -> None:
        if isinstance(event, PolicyBound):
            self._state = PolicyState.BOUND
```

Event-Sourcing Decision Table

Event sourcing is not universally appropriate. We apply the following decision criteria to determine persistence patterns per bounded context:

Criterion	→ Event Sourcing	→ State Persistence
Audit criticality (regulatory, claims trail)	✓ ES default	—
Write volume (millions/day, tight latency)	—	✓ State default
Schema volatility (frequent evolution)	✓ ES with upcasters	—
Cross-context choreography (sagas)	✓ ES preferred	—
Reporting/reconciliation needs	✓ ES + projections	State + boundary events

Default polarity: Event sourcing is the default for audit-critical Policy and Claims contexts. State persistence with boundary events is the default for high-throughput Billing and Payment operations. Deviations require documented justification in the context RFC.

ES Readiness Checklist

Before deploying ES-selected contexts: (1) versioned events and upcasters in place, (2) Decider/Aggregate patterns enforced, (3) projection rebuild strategy documented, (4) compensation semantics (idempotency, exactly-once) tested.

ES Performance & Storage Budgets

Each ES-selected context must document: (1) projected event volume (events/day, peak rate/sec), (2) snapshot frequency and rebuild time target (e.g., full rebuild < 4 hours), (3) projection lag tolerance (e.g., < 500ms for read models), (4) storage growth rate and retention policy, (5) cost estimate (storage, compute for projections). Contexts exceeding thresholds trigger architecture review before deployment.

Saga Patterns from Multi-Step Procedures

COBOL procedures that span multiple records or files become sagas—choreographed sequences of local transactions with compensating actions:

```
class NewBusinessSaga:
    """Generated from COBOL PERFORM spanning multiple files."""

    def __init__(self, event_bus: EventBus):
        self.event_bus = event_bus
        event_bus.subscribe(PolicyBound, self.on_policy_bound)
        event_bus.subscribe(PaymentReceived, self.on_payment_received)
        event_bus.subscribe(PaymentFailed, self.on_payment_failed)

    async def on_policy_bound(self, event: PolicyBound):
        await self.event_bus.publish(RequestPayment(
            policy_id=event.policy_id,
            amount=event.premium
        ))

    async def on_payment_received(self, event: PaymentReceived):
        await self.event_bus.publish(IssuePolicy(
            policy_id=event.policy_id
        ))

    async def on_payment_failed(self, event: PaymentFailed):
        # Compensating action
        await self.event_bus.publish(CancelBinding(
            policy_id=event.policy_id,
            reason='payment_failed'
        ))
```

3.6 Semantic Discovery: The HITL Reality

The examples above assume we can generate meaningful domain artifacts like PolicyAggregate and PolicyBound. But when source identifiers are cryptic—P1200, W3-S1, P1220—how do we discover that this represents policy lifecycle processing? This is the central challenge that commercial vendors address through substantial human-in-the-loop (HITL) effort, and we must be honest about it.

Prior research on business rule extraction [10, 12, 13] has established foundational techniques: control flow graph analysis, program slicing on business-relevant variables, and model-driven frameworks. Our approach builds on these foundations while adding LLM-based semantic inference and enterprise ontology alignment—capabilities unavailable when earlier tools were developed.

The Semantic Gap

FSM extraction can identify STRUCTURE: there's a state machine with states '1', '2', '3', transitions triggered by specific conditions, and actions that invoke other procedures. But MEANING—that state '2' represents 'Policy Quoted' and P1220 performs 'Underwriting Validation'—requires semantic discovery. The code itself doesn't contain this information when identifiers are coded rather than mnemonic.

Multi-Source Semantic Inference

Our approach uses multiple signals to infer semantics, ranked by reliability:

```
class SemanticInferenceEngine:
    """Infer semantic meaning from multiple sources."""

    def infer_semantics(self, fsm: FSM, identifier: str) -> SemanticAnnotation:
        candidates = []

        # Source 1: Surviving documentation (highest confidence)
        if doc_match := self.knowledge_base.lookup_documentation(identifier):
            candidates.append(SemanticCandidate(
                meaning=doc_match.description,
                confidence=0.95,
                source='documentation'
            ))

        # Source 2: Data dictionary fragments (if any survived)
        if dict_match := self.knowledge_base.lookup_data_dictionary(identifier):
            candidates.append(SemanticCandidate(
                meaning=dict_match.business_name,
                confidence=0.90,
                source='data_dictionary'
            ))

        # Source 3: Usage pattern analysis across codebase
        usage_patterns = self.analyze_usage_patterns(identifier)
        if usage_patterns.consistent:
            candidates.append(SemanticCandidate(
                meaning=usage_patterns.inferred_purpose,
                confidence=0.70,
                source='usage_inference'
            ))

        # Source 4: LLM inference from context (lowest confidence)
        llm_guess = self.llm.infer_meaning(identifier, fsm.context)
        candidates.append(SemanticCandidate(
            meaning=llm_guess,
            confidence=0.50,
            source='llm_inference'
        ))

        return self.select_best_candidate(candidates)
```

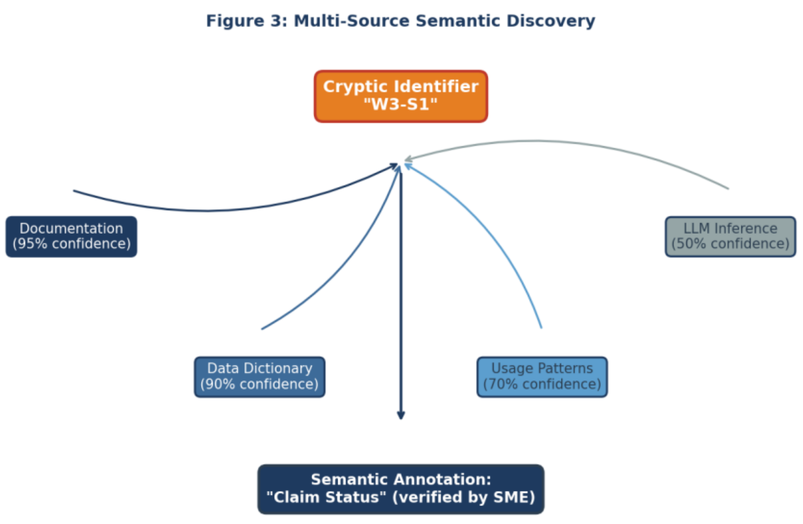
Figure 3: Multi-Source Semantic Discovery
Cryptic Identifier "W3-S1"
Documentation (95% confidence)
LLM Inference (50% confidence)
Data Dictionary (90% confidence)
Usage Patterns (70% confidence)
Semantic Annotation: "Claim Status" (verified by SME)

Figure 3: Multi-Source Semantic Discovery

Human-in-the-Loop Validation

No matter how sophisticated the inference, semantic discovery requires human validation—particularly for the 30-50% of identifiers where automated inference confidence is below 0.70. Our approach differs from current vendors not in eliminating HITL, but in HOW we structure it:

- Batch validation: Present SMEs with clusters of related identifiers for validation, not individual items. 'These 47 identifiers appear to relate to policy rating—confirm or correct.'
- Confidence-based prioritization: Focus SME time on low-confidence inferences. High-confidence matches (from documentation) proceed automatically.
- Propagation: When an SME confirms W3-S1 means 'claim status', propagate to all 147 occurrences across the codebase automatically.
- Knowledge capture: Every SME validation enriches the knowledge base, improving inference for subsequent modules.

We estimate 40-60% of semantic discovery can be automated with high confidence; the remainder requires structured SME engagement. This is a significant investment, but it's an investment that produces a permanent asset—a semantic knowledge base that doesn't exist today and would benefit any modernization approach.

3.7 Twelve-Factor Compliance

Generated code follows twelve-factor principles by design:

Factor	COBOL Pattern	Generated Pattern
Config	Hardcoded in WORKING-STORAGE	Environment variables, config service
Backing Services	CICS, MQ, DB2 direct coupling	Injected via environment URLs
Statelessness	State in COMMAREA/DFHEIBLK	Event-sourced state, external stores
Logs	DISPLAY statements to SYSOUT	Structured logging to stdout

4. Behavioral Equivalence Verification

Confidence in the modernized system requires proving behavioral equivalence—that the new code produces identical outputs for identical inputs.

4.1 FSM Bisimulation

We verify that the original (extracted) FSM and the generated code's FSM are bisimilar—they can simulate each other's transitions:

```
def verify_bisimulation(original_fsm: FSM, generated_code: Module) -> VerificationResult:
    """Verify behavioral equivalence through bisimulation checking."""

    # Extract FSM from generated code (reverse engineering)
    generated_fsm = extract_fsm_from_code(generated_code)

    # Build relation checking
    relation = set()
    worklist = [(original_fsm.initial, generated_fsm.initial)]

    while worklist:
        s1, s2 = worklist.pop()
        if (s1, s2) in relation:
            continue
        relation.add((s1, s2))

        # For each transition from s1, s2 must have matching transition
        for t1 in original_fsm.transitions_from(s1):
            matching = [t2 for t2 in generated_fsm.transitions_from(s2)
                        if t2.label == t1.label]
            if not matching:
                return VerificationResult(
                    success=False,
                    failure=f'Missing transition {t1} from {s2}'
                )
            for t2 in matching:
                worklist.append((t1.target, t2.target))

    return VerificationResult(success=True, relation=relation)
```

4.2 Property-Based Testing

Beyond structural equivalence, we verify semantic properties using property-based testing:

```
from hypothesis import given, strategies as st

class PolicyBehaviorTests:
    """Property-based tests derived from FSM specifications."""

    @given(st.builds(PolicyData))
    def test_quote_to_bind_preserves_premium(self, policy_data):
        """Property: Premium calculated at quote equals premium at bind."""
        # Run through original COBOL (via service wrapper)
        cobol_result = cobol_service.quote_and_bind(policy_data)

        # Run through generated service
        modern_result = modern_service.quote_and_bind(policy_data)

        assert cobol_result.premium == modern_result.premium
    @given(st.builds(ClaimData))
    def test_claim_state_transitions_match(self, claim_data):
        """Property: State sequences are identical."""
        cobol_states = cobol_service.process_claim_get_states(claim_data)
        modern_states = modern_service.process_claim_get_states(claim_data)

        assert cobol_states == modern_states
```

4.3 Shadow Mode Validation

Before full cutover, generated services run in shadow mode—receiving production traffic but not serving responses:

```
class ShadowModeValidator:
    def __init__(self, cobol_service, modern_service, metrics):
        self.cobol = cobol_service
        self.modern = modern_service
        self.metrics = metrics

    async def process(self, request):
        # Execute both in parallel
        cobol_future = self.cobol.process(request)
        modern_future = self.modern.process(request)

        cobol_result = await cobol_future
        modern_result = await modern_future

        # Compare and record
        match = self._compare(cobol_result, modern_result)
        self.metrics.record_comparison(match, request.type)

        # Always return COBOL result in shadow mode
        return cobol_result

    def _compare(self, cobol, modern):
        return ComparisonResult(
            outputs_match=cobol.output == modern.output,
            state_match=cobol.final_state == modern.final_state,
            timing_delta=modern.duration - cobol.duration
        )
```

4.4 Verification Instrumentation Requirements

Bisimulation checking for distributed services requires instrumented observability. Generated code must emit traces that enable equivalence verification:

Transition/Guard Traces

Every generated service emits structured logs for each state transition:

```
@dataclass
class TransitionTrace:
    trace_id: str          # Correlation across services
    service: str           # Originating service
    fsm_id: str            # Source FSM identifier
    from_state: str        # Previous state
    to_state: str          # New state
    trigger: str           # Event/command that triggered
    guard_evaluated: Dict[str, bool]  # All guards checked
    timestamp: datetime
```

```
# Emitted on every transition:
logger.info('transition', extra=asdict(TransitionTrace(...)))
```

Shadow Comparison Dimensions

Shadow mode comparison explicitly checks these dimensions:

- Outputs: Response payloads, return codes, error messages
- Persisted state deltas: Database writes (row-level), file outputs (record-level)
- Emitted domain events: Event type, payload, sequence
- Timing tolerances: Latency within defined bounds (±10% or absolute threshold)
- Ordering guarantees: Message/record sequence preservation where required

Three independent verification methods must agree before declaring equivalence: (1) bisimulation via transition traces, (2) property-based test suites, and (3) shadow mode comparison on production traffic.

Privacy & Compliance in Verification Traces

Transition traces from Policy and Claims contexts may contain PII. All emitted traces must: (1) apply field-level masking for SSN, policy numbers, claim IDs, and PHI fields per data classification tags, (2) emit classification labels in trace metadata for downstream filtering, (3) respect RBAC scopes—only authorized roles (Verification Engineer, Compliance Auditor) may access unmasked traces, (4) enforce retention limits aligned with data governance policy (default: 90 days for masked, 30 days for unmasked).

5. Implementation Plan

5.1 Technical Components

Static Analysis Layer

Built on ANTLR with the cobol85 grammar (github.com/antlr/grammars-v4/tree/master/cobol85), extended for IBM Enterprise COBOL dialects. Produces AST, Control Flow Graphs, and Data Flow Graphs for each program.

Knowledge Base

FAISS vector store containing COBOL language specifications, our copybook library, domain terminology from the Observability Ontology, and existing program documentation. Updated incrementally as extraction proceeds.

LLM Extraction Engine

Claude or GPT-4 class model with RAG augmentation. Processes code chunks with retrieved context, outputs FSM specifications in JSON format. Ensemble of multiple extractions with voting for high-confidence results.

Domain Mapper

Maps extracted FSMs to enterprise ontology: value streams, capabilities, domain events. Uses embeddings similarity and entity matching against our published ontology.

Code Generator

Template-based generation targeting multiple architectures: Spring Boot microservices, reactive streams, serverless functions. All generators enforce twelve-factor compliance.

Verification Suite

Bisimulation checker, property-based test generator, shadow mode infrastructure. Integrated with CI/CD for continuous validation.

5.2 Phased Delivery

1. Phase 1 - Foundation (Months 1-3): Static analysis pipeline, knowledge base construction, initial FSM extraction on pilot programs. Deliverable: FSM extraction achieving >85% accuracy on pilot.
2. Phase 2 - Domain Integration (Months 4-6): Ontology mapping, bounded context identification, aggregate detection. Deliverable: Automated domain decomposition for pilot programs.
3. Phase 3 - Generation (Months 7-10): Code generators for target architectures, verification suite. Deliverable: End-to-end modernization of pilot programs with verified equivalence.
4. Phase 4 - Scale (Months 11-16): Production pilot with shadow mode, expand to additional program families. Deliverable: Modernization pipeline meeting per-artifact UCR targets (§5.6).

5.3 Pilot Selection Criteria

Initial pilots should be programs that are:

- Medium complexity (5,000-20,000 lines)—large enough to validate the approach, small enough for rapid iteration
- Well-documented with clear business purpose
- Representative of larger patterns (batch processing, CICS transactions, or both)
- Non-critical path—allowing shadow mode validation without production risk
- Owned by engaged business stakeholders who can validate domain mappings

Pilot Anti-Criteria (Defer Until Instrumentation Battle-Tested)

- Payment rails and financial settlement: High regulatory sensitivity; defer until shadow comparison dimensions include ledger reconciliation and audit trail verification
- Ultra-high-volume nightly batch (>1M records): Verification harness must prove checkpoint/restart equivalence and replay performance before tackling production-scale batch

5.4 Knowledge Base Readiness Index (KBRI)

Phase progression is gated by a quantitative readiness score:

$$\mathbf{KBRI = 0.25 \cdot IC + 0.25 \cdot CC + 0.20 \cdot OA + 0.15 \cdot EL + 0.15 \cdot CRS}$$

Where:

- IC (Identifier Coverage): % of unique domain identifiers labeled at confidence ≥ 0.80
- CC (Copybook Coverage): % of copybooks resolved and normalized (no unresolved COPY statements)
- OA (Ontology Alignment): % of FSM transitions mapped to canonical domain events at confidence ≥ 0.80
- EL (Event Linkage): % of transitions producing/consuming events per enterprise ontology
- CRS (Conflict Resolution Score): 1 − (conflicts / labeled items), with conflicts adjudicated

5.5 Domain-Specific Extraction Metrics

Aggregate accuracy targets mask where extraction actually fails. We track per-artifact metrics with explicit thresholds:

Metric	Target	Gate
CICS Navigation F1 (pseudo-conversational flows)	≥ 0.85	Phase 1→2
Batch Phase Boundary F1 (OPEN..CLOSE spans)	≥ 0.80	Phase 1→2
Copybook Resolution Accuracy	≥ 0.95	Phase 1→2
Call-Graph Coverage (reachable procedures)	≥ 0.90	Phase 1→2
Guard Condition Precision/Recall	≥ 0.80	Phase 2→3
Transition Miss Rate (vs. gold truth set)	≤ 0.05	Phase 2→3

5.6 Unplanned Correction Rate (UCR)

The 'human intervention' metric is reframed to distinguish planned SME validation (budgeted) from unplanned corrections (escapes). UCR measures residual errors after the validation pipeline:

$$\mathbf{UCR = (corrections\ after\ validation)\ /\ (extracted\ FSM\ specs)}$$

Per-artifact UCR targets:

- Copybook fix rate: ≤ 2%
- State label correction: ≤ 10%
- Guard condition correction: ≤ 15%
- Data-entity mapping correction: ≤ 10%
- Transition miss correction: ≤ 5%

Note: SME semantic validation is planned work and budgeted separately. UCR measures only escapes that require rework after validation completes.

5.7 Phase GO/NO-GO Gates

Phase 1 → 2 GO:

- KBRI ≥ 0.70
- Copybook unresolved ≤ 5%
- Domain-specific extraction: CICS F1 ≥ 0.85, Batch F1 ≥ 0.80, Call-graph ≥ 0.90
- Semantic Review Board constituted; first 300 identifiers adjudicated and propagated

Phase 2 → 3 GO:

- KBRI ≥ 0.80; conflict rate ≤ 10%
- Event namespaces (pl.*, bi.*) registered; envelope/versioning rules enforced in registry
- ES decision table applied per context; ES readiness checklist complete where ES selected
- Ontology alignment disagreements documented and adjudicated

Phase 3 → 4 GO:

- UCR ≤ per-artifact targets across pilot cohort
- Bisimulation verified via transition/guard traces + property suites
- Shadow mode harness ready; replay infrastructure tested; data masking signed off

Program GREEN:

- Shadow delta thresholds (outputs/state/events) within tolerances for ≥ 30 days
- Incident rate ≤ baseline; no P1/P2 attributable to modernized components
- Governance in Git: CODEOWNERS on KB/ontology/policy paths; protected branches; semantic decision log

5.8 Gate Ownership & Funding Map

Each gate has an accountable owner and funding source:

Gate	Owner	Sign-Off	Funding Source
Phase 1→2	Enterprise Architecture	Dir. App Infrastructure	IT Modernization
Phase 2→3	Domain Owners + EA	VP Engineering	IT Modernization
Phase 3→4	Platform Engineering	VP Engineering + Compliance	IT Modernization
GREEN	Program Office	CIO + Business Sponsor	Operational Budget

6. Risks and Mitigations

Risk	Impact	Mitigation
FSM extraction accuracy below threshold	Increased human review, slower throughput	Ensemble extraction, iterative prompt refinement, expanded knowledge base
Domain mapping errors	Incorrect bounded contexts, misaligned services	Business stakeholder validation gates, iterative refinement of ontology mappings
Over-decomposition	Too many small services, operational complexity	Data affinity thresholds, minimum context size constraints, strangler fig incremental approach
Behavioral verification gaps	Undetected regressions in production	Extended shadow mode, production traffic replay, comprehensive property tests
LLM capability regression	Model updates degrade extraction quality	Version pinning, regression test suite, multi-model fallback

Over-Decomposition Numeric Guardrails

To prevent service proliferation, the following constraints are enforced:

- Minimum context size: ≥ 3 aggregates OR ≥ 5 extracted FSMs per bounded context
- Maximum services per capability per wave: 8 (exceptions require architecture review)
- Operational readiness before splitting: SLOs defined, runbooks documented, on-call rotation staffed
- Platform cost model: compute, observability, and pipeline costs estimated per service; thin-slice vs. consolidation trade-off documented

6.1 Governance Model

Configuration-as-code governance prevents drift and ensures audit trail:

CODEOWNERS

- ontology/** — Enterprise Architecture team (2 approvers required)
- kb/** — Knowledge Base curators + Domain SME rotation
- policy/** — Semantic Review Board
- generated/** — Code generation templates owned by Platform team

Protected Branches

Main branches require: (1) passing CI including extraction regression suite, (2) CODEOWNER approval, (3) semantic diff review for KB/ontology changes.

Schema Registry

All domain events registered with versioning rules from DDD model. Breaking changes require 30-day deprecation (internal) or 90-day (partner APIs). Schema evolution enforced via CI gate.

Semantic Decision Log

Every semantic adjudication (identifier meaning, context boundary, event mapping) logged with rationale, reviewer, and timestamp. Log is append-only and feeds KB propagation.

7. Conclusion

Direct code translation cannot modernize enterprise legacy systems at scale. The mathematics of compound error, combined with the architectural preservation problem ('JOBOL' and its equivalents in other language pairs), renders such approaches inadequate for organizations of our size and complexity. Commercial services, while sophisticated, optimize for translation rather than true architectural modernization.

The behavioral understanding approach—extracting FSMs as platform-independent specifications—addresses these limitations. By capturing what the code does rather than how it's written, we obtain specifications that can be forward-engineered to genuinely modern architectures. Grounding this extraction in our enterprise domain ontology ensures the resulting bounded contexts align with how our business actually operates.

We are transparent about the challenges: cryptic 1970s-era identifiers require semantic discovery that depends significantly on surviving documentation and SME knowledge capture. Cross-context programs must be decomposed into aligned bounded contexts. These are not trivial problems, and they require structured human-in-the-loop engagement—but our approach reduces and structures that engagement rather than relying on ad-hoc manual effort.

The research foundation is sound: demonstrated >90% accuracy on FSM extraction, 91% on specification extraction, and 82% on direct translation (our baseline to exceed). The technical components are mature: ANTLR parsing, RAG-enhanced LLMs, vector similarity search, and property-based testing are production-ready technologies.

The proposal requests funding for a 16-month phased implementation, beginning with pilot programs and scaling to production throughput. Success criteria are concrete and measurable: domain-specific extraction metrics (CICS F1 ≥ 0.85, batch F1 ≥ 0.80, guard precision ≥ 0.80), KBRI-gated phase progression, per-artifact Unplanned Correction Rates (copybook ≤ 2%, state labels ≤ 10%, guards ≤ 15%), and verified behavioral equivalence via three independent methods.

This investment positions Farmers to systematically retire legacy technical debt—COBOL, VB, .NET Framework, WebSphere-dependent Java—while simultaneously decomposing monolithic systems into domain-aligned microservices. Not merely relocating legacy code to new platforms, but genuinely modernizing our architecture.

References

[1] IBM Research. "Quality Evaluation of COBOL to Java Code Transformation." arXiv:2507.23356, July 2025. arxiv.org/abs/2507.23356

[2] AWS. "How to Migrate Mainframe Batch to Cloud Microservices with AWS Blu Age." AWS Partner Network Blog, April 2022. aws.amazon.com/blogs/apn/...

[3] ISO/IEC 19506:2012. "Knowledge Discovery Metamodel (KDM)." OMG specification v1.3.

[4] Wei, H. et al. "Unleashing the Power of LLM to Infer State Machine from the Protocol Implementation." arXiv:2405.00393, May 2024. arxiv.org/abs/2405.00393

[5] Wael, F. et al. "An Agentic Flow for Finite State Machine Extraction using Prompt Chaining." arXiv:2507.11222, July 2025. arxiv.org/abs/2507.11222

[6] Zhang, M. et al. "Automated Extraction of Protocol State Machines from 3GPP Specifications with Domain-Informed Prompts and LLM Ensembles." arXiv:2510.14348, October 2025. arxiv.org/abs/2510.14348

[7] Zhang, Y. et al. "Translation of Low-Resource COBOL to Logically Correct and Readable Java leveraging High-Resource Java Refinement." ICSE 2024 LLM4Code Workshop. ICSE 2024

[8] Farmers Insurance Group. "Observability Ontology Guidance." Internal document. Value streams, capabilities, and domain events for P&C insurance.

[9] ANTLR. "COBOL85 Grammar." github.com/antlr/grammars-v4/tree/master/cobol85

[10] Cosentino, V. et al. "Extracting Business Rules from COBOL: A Model-Based Framework." 20th Working Conference on Reverse Engineering (WCRE), 2013. researchgate.net/publication/334946627

[11] CloseLoop. "LLMs in Legacy System Migration." Technical blog, 2024. closeloop.com/blog/llms-in-legacy-system-migration

[12] Sneed, H. M. "Extracting Business Logic from Existing COBOL Programs as a Basis for Redevelopment." IEEE International Workshop on Program Comprehension (IWPC), 2001. dl.acm.org/doi/10.1145/2383276.2383286

[13] Ali, S. et al. "COBREX: A Tool for Extracting Business Rules from COBOL." IEEE Conference on Software Maintenance and Evolution (ICSME), 2022. CFG-based business rule extraction approach.

[14] Xu, F. et al. "Large Language Models for Code Analysis: Do LLMs Really Do Their Job?" arXiv:2411.14971, November 2024. arxiv.org/abs/2411.14971

[15] Addepto. "How LLMs Could Help Migrate Legacy Systems." Technical analysis, 2024. addepto.com/blog/how-llms-could-help-migrate-legacy-systems

[16] Microsoft Azure Samples. "Legacy Modernization Agents." AI-powered COBOL to Java/C# agents using Semantic Kernel. Comparison baseline for direct translation approaches. github.com/Azure-Samples/Legacy-Modernization-Agents

Appendix A: Architecture Diagrams

The following C4 model diagrams illustrate the proposed modernization platform architecture at multiple levels of abstraction.

A.1 System Landscape View

The complete modernization ecosystem showing all systems, personas, and their relationships.

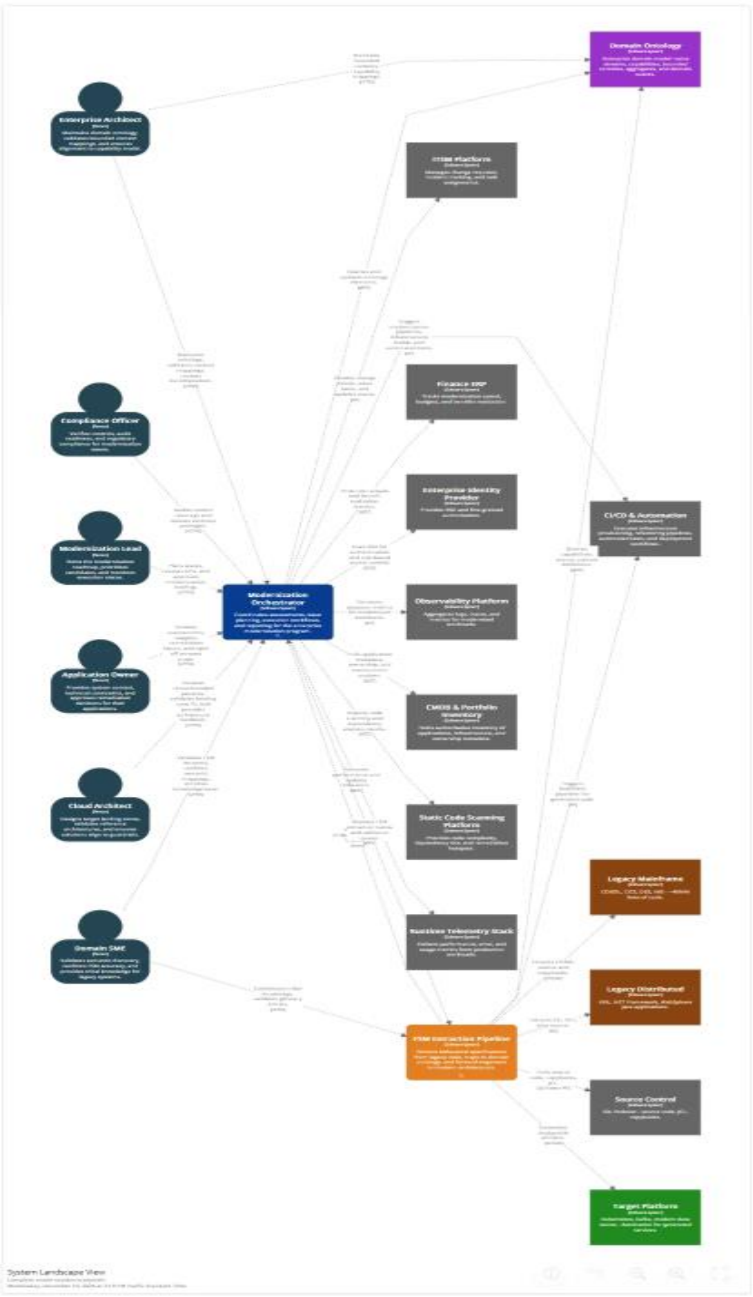

Figure A-1: System Landscape - Complete Modernization Ecosystem

A.2 FSM Extraction Pipeline - System Context

The FSM Extraction Pipeline in context with its surrounding systems and personas.

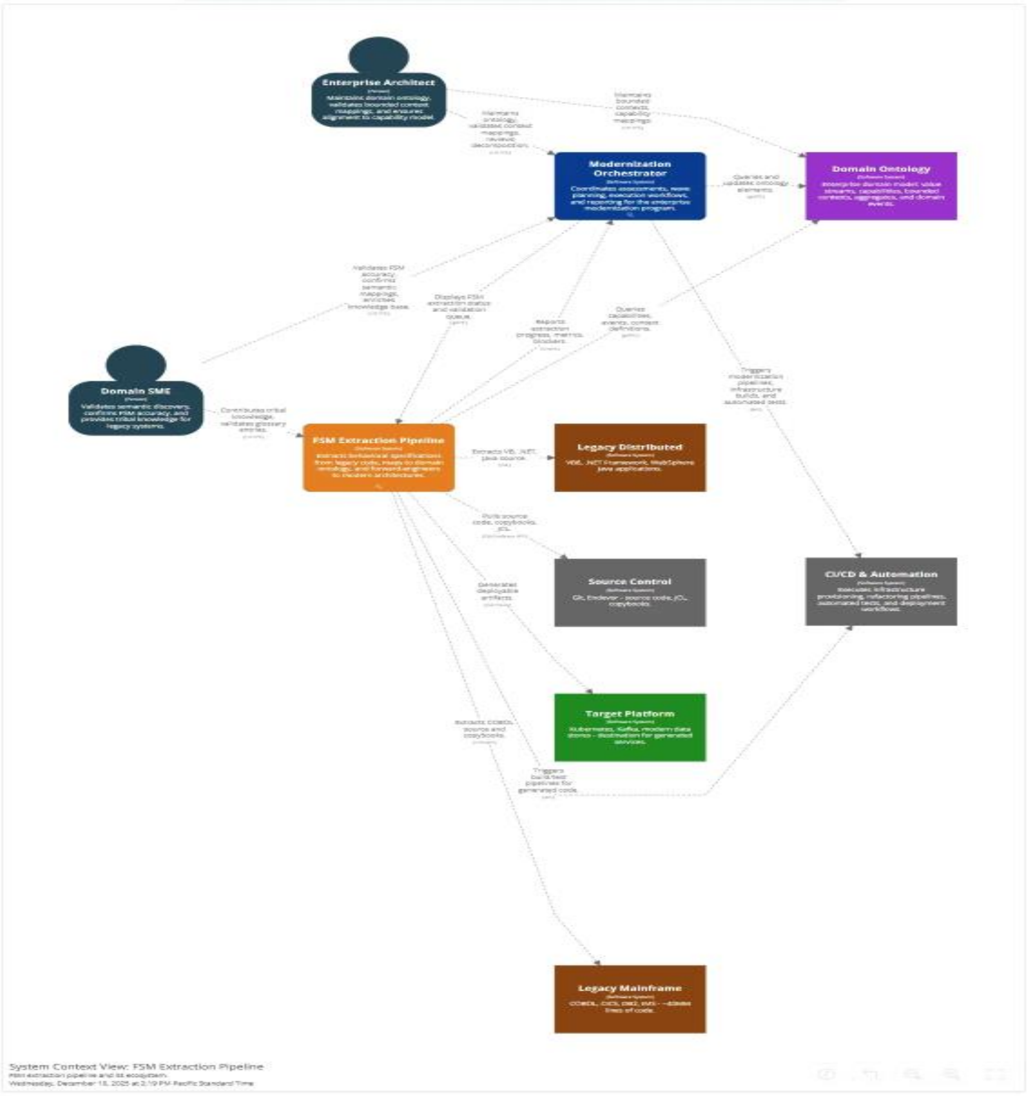
Enterprise Architect
Maintains domain ontology, validates bounded context mappings, and ensures alignment to capability model.
Maintains ontology, validates context mappings, reviews decomposition.
Maintains bounded contexts, capability mappings
Modernization Orchestrator
Coordinates assessments, wave planning, execution workflows, and reporting for the enterprise modernization program.
Queries and validates ontology elements
Domain Ontology
Enterprise domain model: value streams, capabilities, bounded contexts, aggregates, and domain events.
Validates FSM accuracy, confirms semantic mappings, enriches knowledge base.
Displays FSM extraction status and validation queue.
Reports extraction progress, metrics, blockers.
Queries capabilities, events, context definitions.
Triggers modernization pipelines, infrastructure builds, and automated tests.
Domain SME
Validates semantic discovery, confirms FSM accuracy, and provides tribal knowledge for legacy systems.
Contributes tribal knowledge, validates glossary entries.
FSM Extraction Pipeline
Extracts behavioral specifications from legacy code, maps to domain ontology, and forward-engineers to modern architectures.
Extracts VB, .NET, Java source
Legacy Distributed
VB6, .NET Framework, WebSphere Java applications.
Pulls source code, copybooks, JCL
Source Control
Git, Endevor - source code, JCL, copybooks.
CI/CD & Automation
Executes infrastructure provisioning, refactoring pipelines, automated tests, and deployment workflows.
Generates deployable artifacts.
Target Platform
Kubernetes, Kafka, modern data stores - destination for generated services.
Extracts COBOL source and copybooks.
Triggers build/test pipelines for generated code.
Legacy Mainframe
COBOL, CICS, DB2, IMS - lines of code.
System Context View: FSM Extraction Pipeline
FSM extraction pipeline and its ecosystem.
Wednesday, December 10, 2025 at 2:19 PM Pacific Standard Time

Figure A-2: FSM Extraction Pipeline - System Context View

A.3 FSM Extraction Pipeline - Container View

Internal containers within the FSM Extraction Pipeline showing the major processing stages.

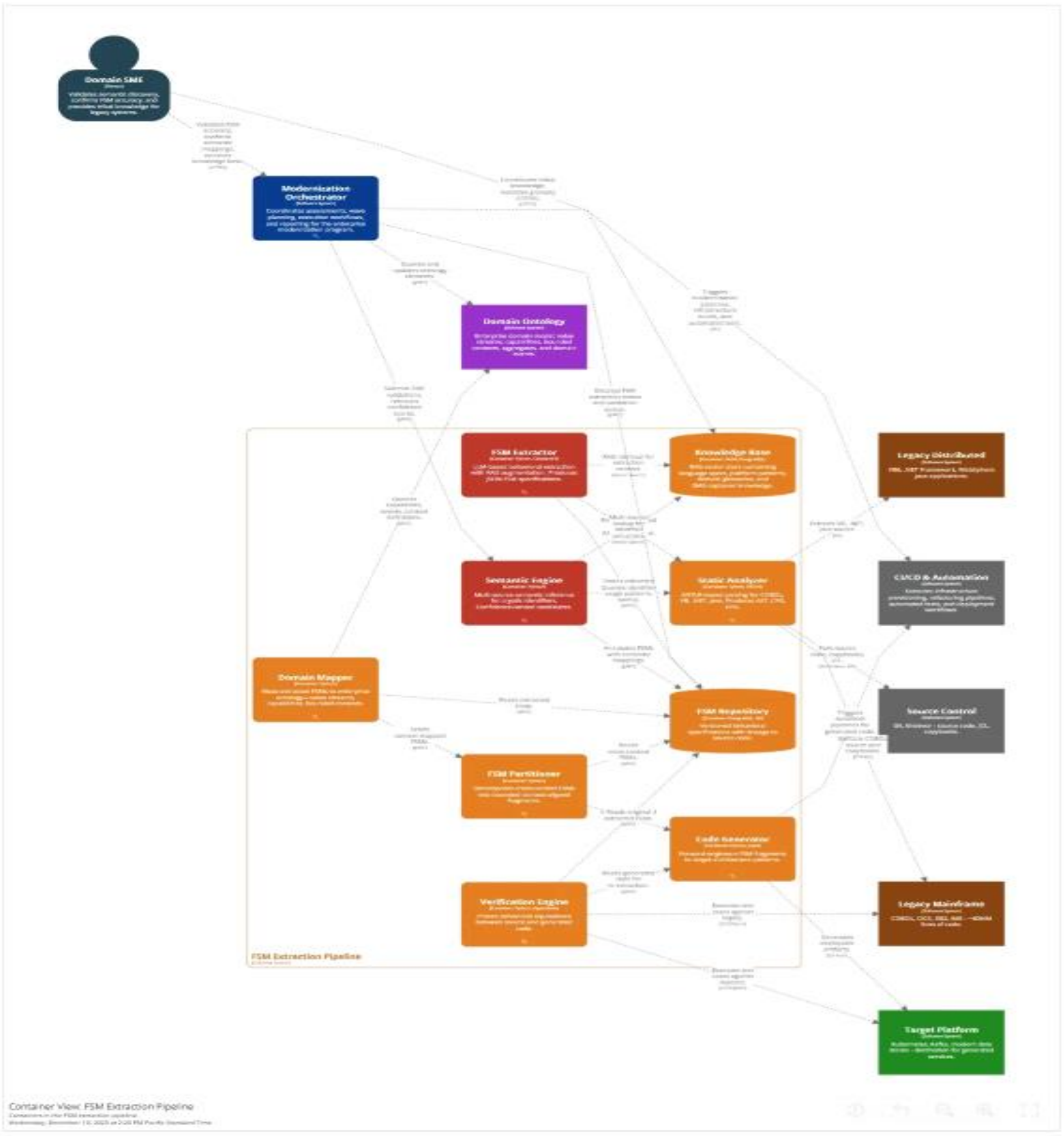

Figure A-3: FSM Extraction Pipeline - Container View

A.4 Component Views

Detailed component views for each major container in the FSM Extraction Pipeline.

Static Analyzer Components

ANTLR-based parsing for COBOL, VB, .NET, and Java with CFG/DFG construction and dead code detection.

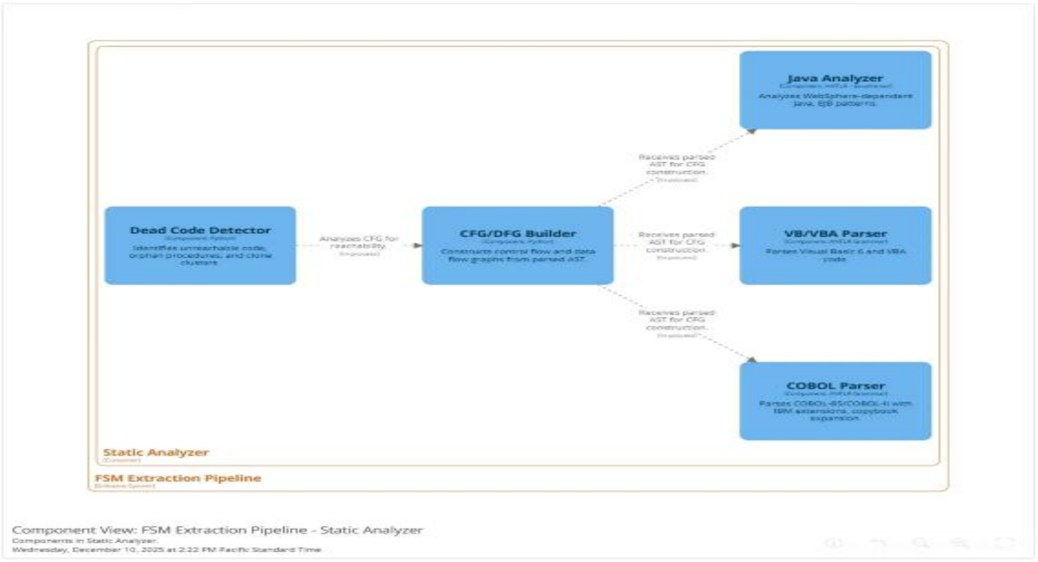
Java Analyzer
Analyzes WebSphere-dependent Java, EJB patterns.
Receives parsed AST for CFG construction.
Dead Code Detector
Identifies unreachable code, orphan procedures, and clone clusters.
Analyzes CFG for reachability
CFG/DFG Builder
Constructs control flow and data flow graphs from parsed AST.
Receives parsed AST for CFG construction.
VB/VBA Parser
Parses Visual Basic 6 and VBA code.
Receives parsed AST for CFG construction.
COBOL Parser
Parses COBOL-85/COBOL-II with IBM extensions, copybook expansion.
Static Analyzer
FSM Extraction Pipeline
Component View: FSM Extraction Pipeline - Static Analyzer
Components in Static Analyzer.
Wednesday, December 10, 2025 at 2:22 PM Pacific Standard Time

Figure A-4: Static Analyzer Components

FSM Extractor Components

LLM-based behavioral extraction with RAG augmentation and ensemble voting for confidence scoring.

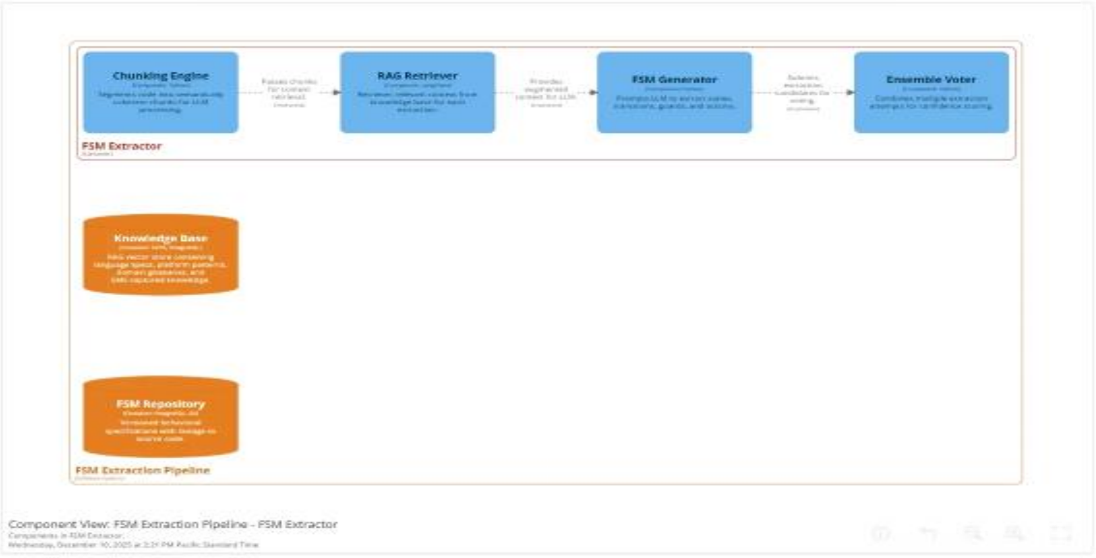

Figure A-5: FSM Extractor Components

Semantic Engine Components

Multi-source semantic inference for cryptic identifiers with confidence-ranked candidates.

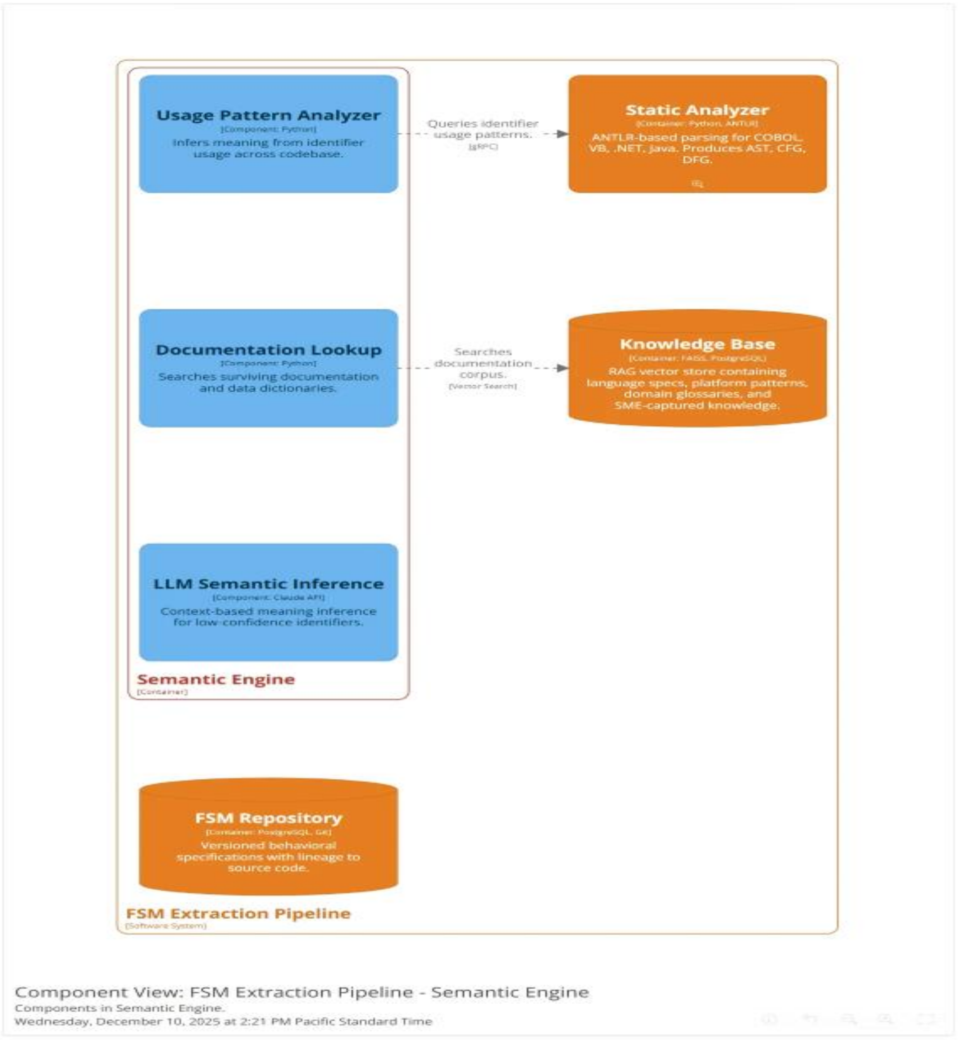
Usage Pattern Analyzer
[Component: Python]
Infers meaning from identifier usage across codebase.
Queries identifier usage patterns.
[gRPC]
Static Analyzer
[Container: Python, ANTLR]
ANTLR-based parsing for COBOL, VB, .NET, Java. Produces AST, CFG, DFG.
Documentation Lookup
[Component: Python]
Searches surviving documentation and data dictionaries.
Searches documentation corpus.
[Vector Search]
Knowledge Base
[Container: FAISS, PostgreSQL]
RAG vector store containing language specs, platform patterns, domain glossaries, and SME-captured knowledge.
LLM Semantic Inference
[Component: Claude API]
Context-based meaning inference for low-confidence identifiers.
Semantic Engine
[Container]
FSM Repository
[Container: PostgreSQL, Git]
Versioned behavioral specifications with lineage to source code.
FSM Extraction Pipeline
[Software System]
Component View: FSM Extraction Pipeline - Semantic Engine
Components in Semantic Engine.
Wednesday, December 10, 2025 at 2:21 PM Pacific Standard Time

Figure A-6: Semantic Engine Components

Domain Mapper Components

Maps extracted FSMs to enterprise ontology—value streams, capabilities, and bounded contexts.

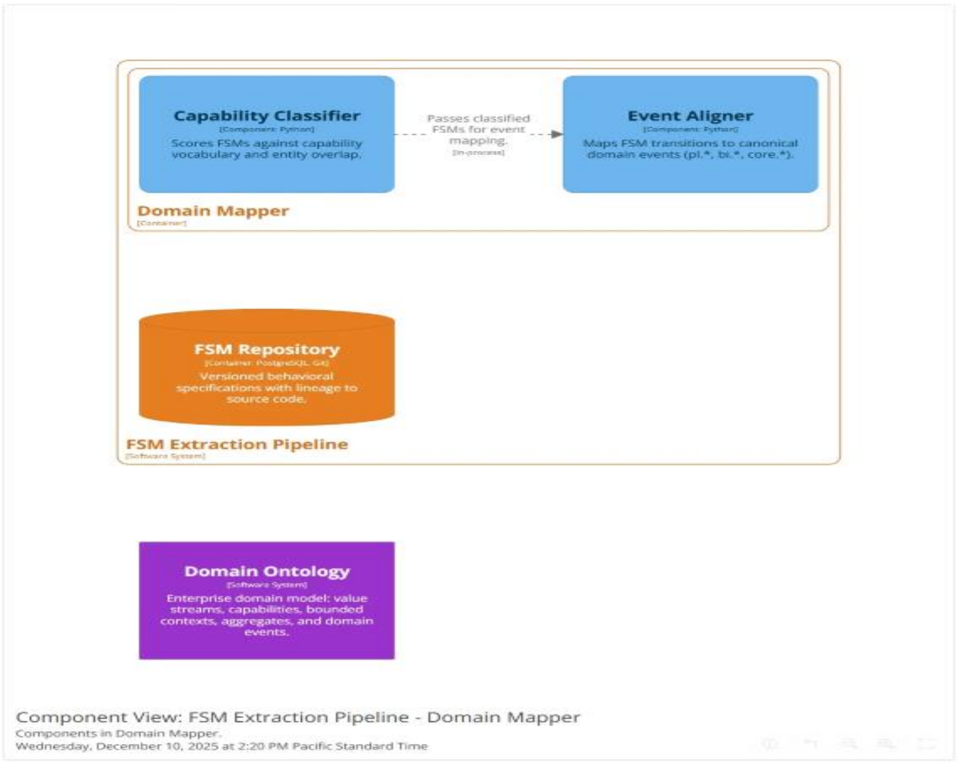
Capability Classifier
[Component: Python]
Scores FSMs against capability vocabulary and entity overlap.
Passes classified FSMs for event mapping.
[In-process]
Event Aligner
[Component: Python]
Maps FSM transitions to canonical domain events (pl.*, bi.*, core.*).
Domain Mapper
[Container]
FSM Repository
[Container: PostgreSQL, Git]
Versioned behavioral specifications with lineage to source code.
FSM Extraction Pipeline
[Software System]
Domain Ontology
[Software System]
Enterprise domain model: value streams, capabilities, bounded contexts, aggregates, and domain events.
Component View: FSM Extraction Pipeline - Domain Mapper
Components in Domain Mapper.
Wednesday, December 10, 2025 at 2:20 PM Pacific Standard Time

Figure A-7: Domain Mapper Components

FSM Partitioner Components

Decomposes cross-context FSMs into bounded-context-aligned fragments with domain event conversion.

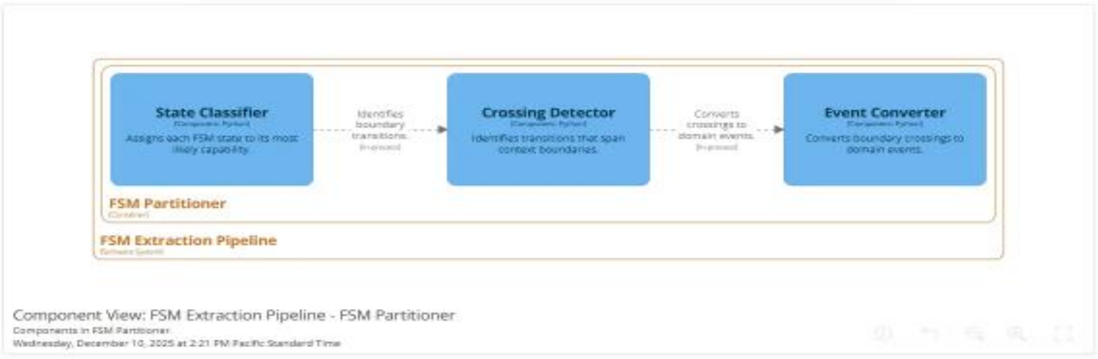
State Classifier
[Component: Python]
Assigns each FSM state to its most likely capability.
Identifies boundary transitions.
[In-process]
Crossing Detector
[Component: Python]
Identifies transitions that span context boundaries.
Converts crossings to domain events.
[In-process]
Event Converter
[Component: Python]
Converts boundary crossings to domain events.
FSM Partitioner
[Container]
FSM Extraction Pipeline
[Software System]
Component View: FSM Extraction Pipeline - FSM Partitioner
Components in FSM Partitioner
Wednesday, December 10, 2025 at 2:21 PM Pacific Standard Time

Figure A-8: FSM Partitioner Components

Code Generator Components

Forward-engineers FSM fragments to DDD aggregates, sagas, and API contracts.

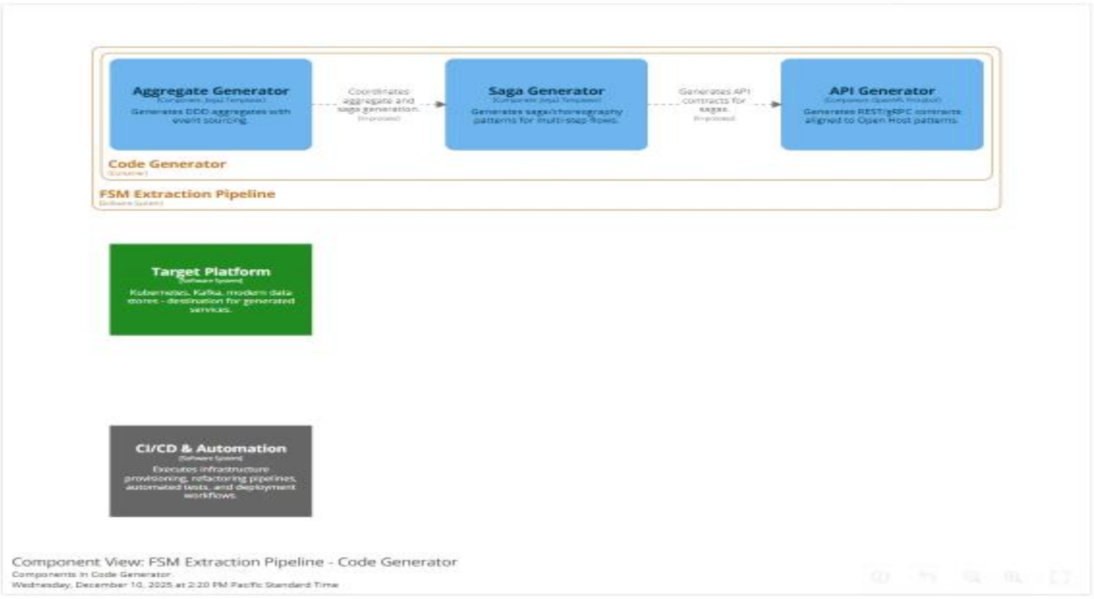
Aggregate Generator
Generates DDD aggregates with event sourcing.
Coordinates aggregate and saga generation.
Saga Generator
Generates saga/choreography patterns for multi-step flows.
Generates API contracts for sagas.
API Generator
Generates REST/gRPC contracts aligned to Open Host patterns.
Code Generator
FSM Extraction Pipeline
Target Platform
[Software System]
Kubernetes, Kafka, modern data stores - destination for generated services.
CI/CD & Automation
[Software System]
Executes infrastructure provisioning, refactoring pipelines, automated tests, and deployment workflows.
Component View: FSM Extraction Pipeline - Code Generator
Components in Code Generator
Wednesday, December 10, 2025 at 2:20 PM Pacific Standard Time

Figure A-9: Code Generator Components

Verification Engine Components

Proves behavioral equivalence through bisimulation checking and shadow execution.

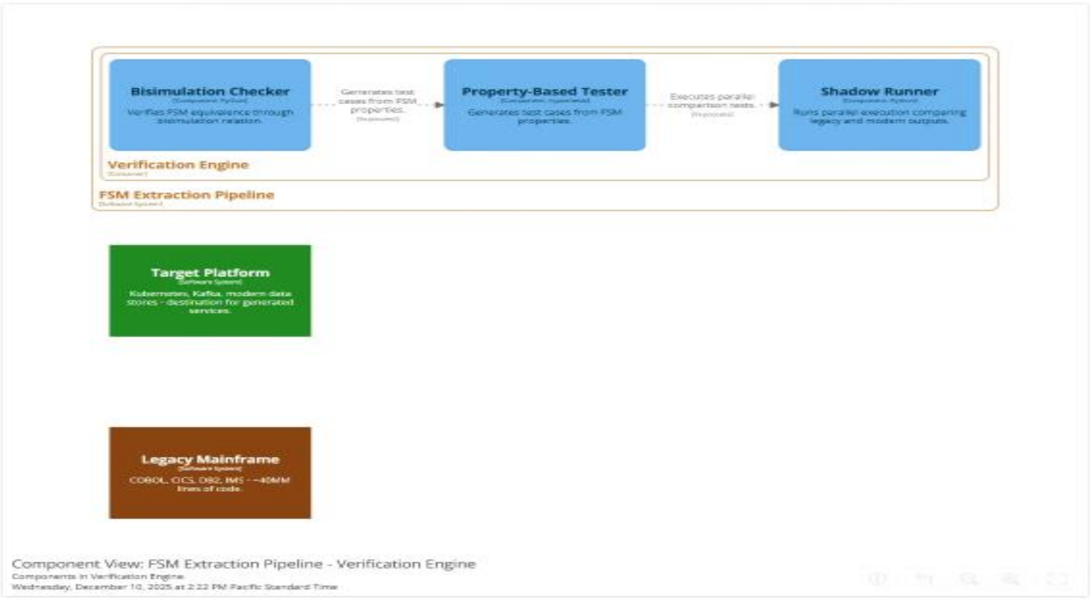
Bisimulation Checker
Verifies FSM equivalence through bisimulation relation.
Generates test cases from FSM properties.
Property-Based Tester
Generates test cases from FSM properties.
Executes parallel comparison tests.
Shadow Runner
Runs parallel execution comparing legacy and modern outputs.
Verification Engine
FSM Extraction Pipeline
Target Platform
[Software System]
Kubernetes, Kafka, modern data stores - destination for generated services.
Legacy Mainframe
[Software System]
COBOL, CICS, DB2, IMS - ~40MM lines of code.
Component View: FSM Extraction Pipeline - Verification Engine
Components in Verification Engine
Wednesday, December 10, 2025 at 2:22 PM Pacific Standard Time

Figure A-10: Verification Engine Components

A.5 Dynamic View: FSM Extraction Workflow

The end-to-end sequence for extracting behavioral specifications from a legacy module, showing the numbered steps through the pipeline.

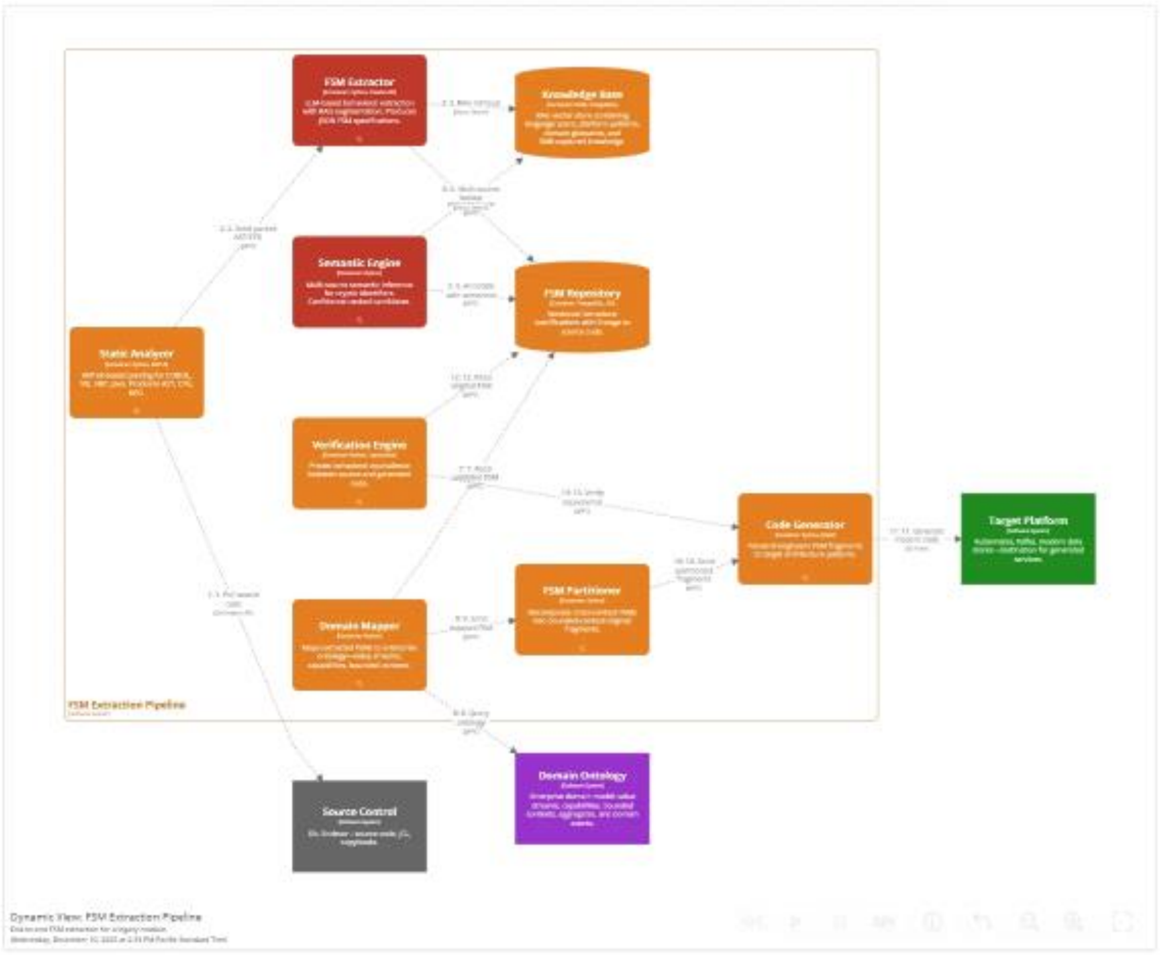

Figure A-11: Dynamic View - End-to-End FSM Extraction Workflow

Appendix B: Modernization Orchestrator Architecture

The Modernization Orchestrator coordinates the "outer loop" of the modernization program—portfolio assessment, wave planning, pattern recommendation, and execution tracking. It integrates with the FSM Extraction Pipeline for applications selected for the "refactor" pattern.

B.1 Modernization Orchestrator - System Context

The orchestrator in context with enterprise systems, personas, and integration points.

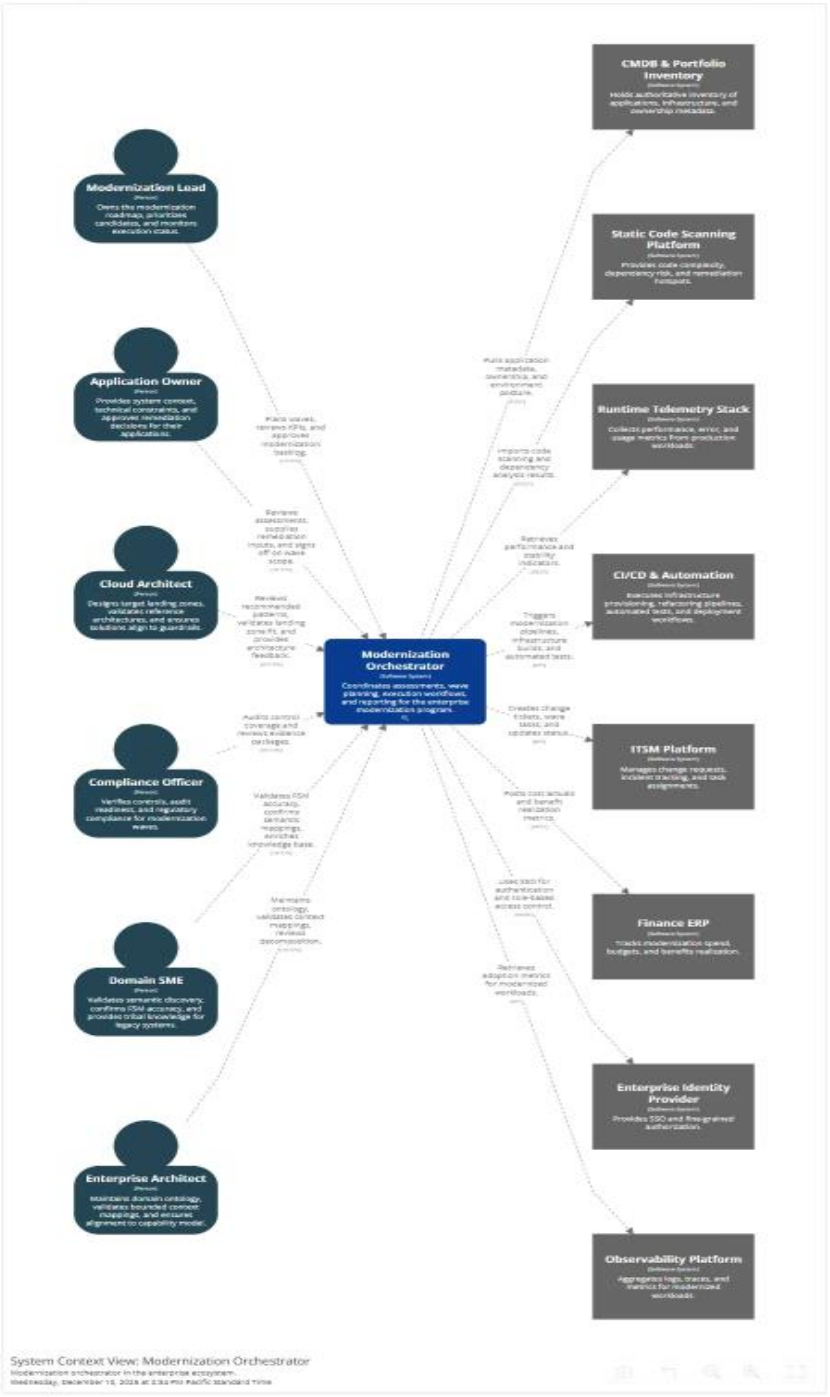
Modernization Lead
[Person]
Owns the modernization roadmap, prioritizes candidates, and monitors execution status.
Application Owner
[Person]
Provides system context, technical constraints, and approves remediation decisions for their applications.
Cloud Architect
[Person]
Designs target landing zones, validates reference architectures, and ensures solutions align to guardrails.
Compliance Officer
[Person]
Verifies controls, audit readiness, and regulatory compliance for modernization waves.
Domain SME
[Person]
Validates semantic discovery, confirms FSM accuracy, and provides tribal knowledge for legacy systems.
Enterprise Architect
[Person]
Maintains domain ontology, validates bounded context mappings, and ensures alignment to capability model.
Modernization Orchestrator
[Software System]
Coordinates assessments, wave planning, execution workflows, and reporting for the enterprise modernization program.
CMDB & Portfolio Inventory
[Software System]
Holds authoritative inventory of applications, infrastructure, and ownership metadata.
Static Code Scanning Platform
[Software System]
Provides code complexity, dependency risk, and remediation hotspots.
Runtime Telemetry Stack
[Software System]
Collects performance, error, and usage metrics from production workloads.
CI/CD & Automation
[Software System]
Executes infrastructure provisioning, refactoring pipelines, automated tests, and deployment workflows.
ITSM Platform
[Software System]
Manages change requests, incident tracking, and task assignments.
Finance ERP
[Software System]
Tracks modernization spend, budgets, and benefits realization.
Enterprise Identity Provider
[Software System]
Provides SSO and fine-grained authorization.
Observability Platform
[Software System]
Aggregates logs, traces, and metrics for modernized workloads.
Plans waves, reviews KPIs, and approves modernization backlog.
Reviews assessments, supplies remediation inputs, and signs off on wave scope.
Reviews recommended patterns, validates landing zone fit, and provides architecture feedback.
Audits control coverage and reviews evidence packages.
Validates FSM accuracy, confirms semantic mappings, enriches knowledge base.
Maintains ontology, validates context mappings, reviews decomposition.
Pulls application metadata, ownership, and environment posture.
Imports code scanning and dependency analysis results.
Retrieves performance and stability indicators.
Triggers modernization pipelines, infrastructure builds, and automated tests.
Creates change tickets, wave tasks, and updates status.
Posts cost actuals and benefit realization metrics.
Uses SSO for authentication and role-based access control.
Retrieves adoption metrics for modernized workloads.
System Context View: Modernization Orchestrator
Modernization orchestrator in the enterprise ecosystem.

Figure B-1: Modernization Orchestrator - System Context View

B.2 Modernization Orchestrator - Container View

Internal containers showing the Portal, Assessment Service, Workflow Orchestrator, Data Hub, Integration Hub, and Reporting Service. Note the connection to the FSM Extraction Pipeline for refactor candidates.

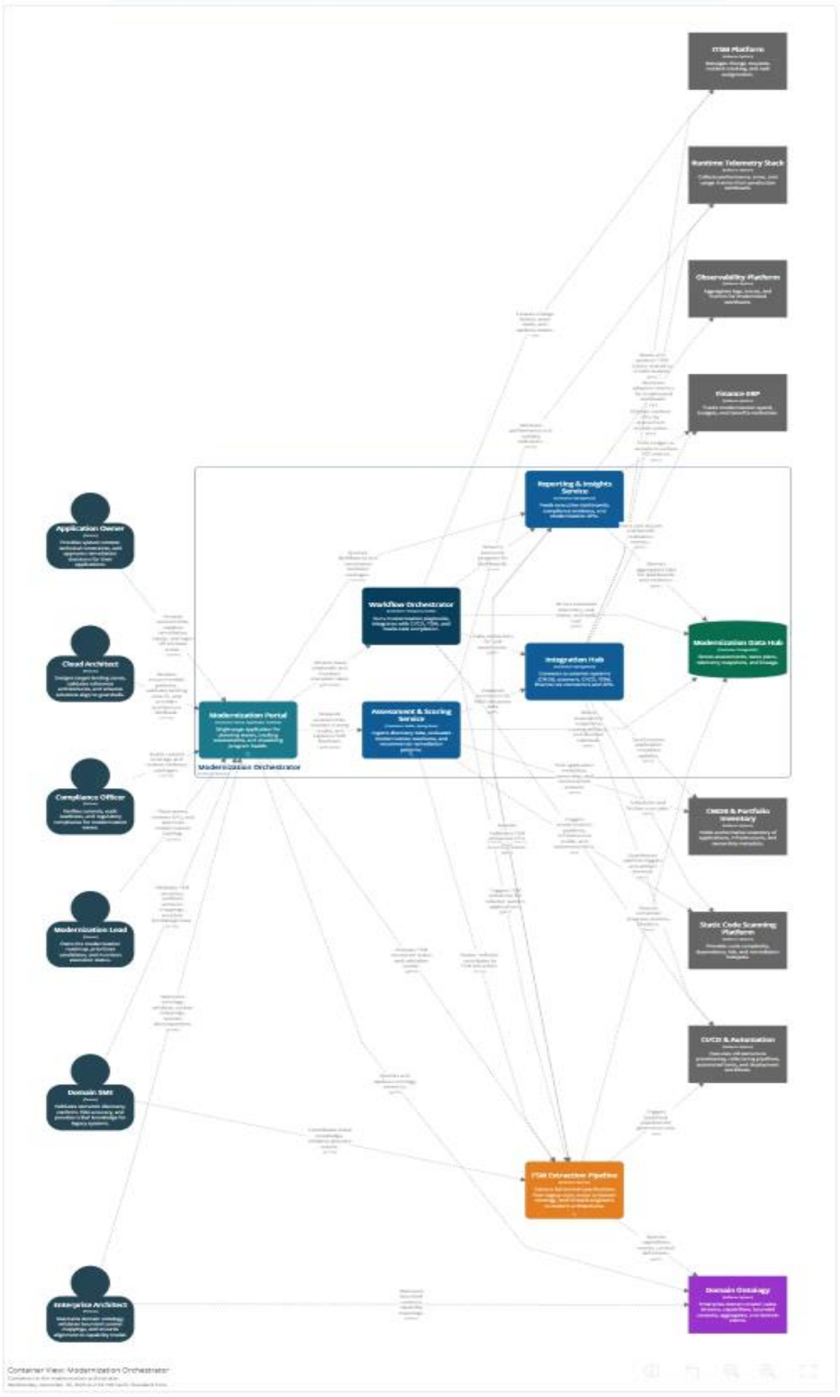

Figure B-2: Modernization Orchestrator - Container View

B.3 Component Views

Assessment & Scoring Service Components

The Discovery Collector, Scoring Engine, and Pattern Recommender. Note how the Pattern Recommender routes "refactor" candidates to the FSM Extraction Pipeline.

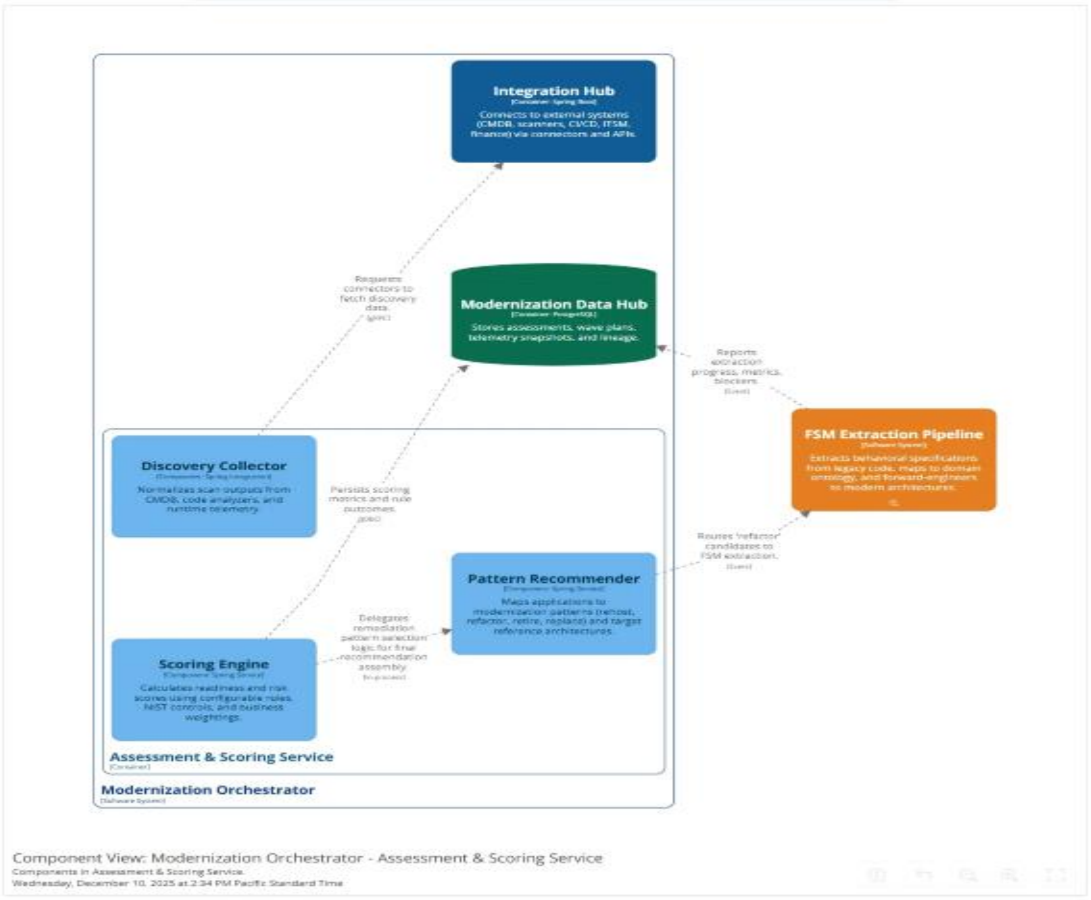
Integration Hub
[Container: Spring Boot]
Connects to external systems (CMDB, scanners, CI/CD, ITSM, finance) via connectors and APIs.
Modernization Data Hub
[Container: PostgreSQL]
Stores assessments, wave plans, telemetry snapshots, and lineage.
Requests connectors to fetch discovery data.
[gRPC]
Reports extraction progress, metrics, blockers
[Event]
FSM Extraction Pipeline
[Software System]
Extracts behavioral specifications from legacy code, maps to domain ontology, and forward-engineers to modern architectures.
Discovery Collector
[Component: Spring Integration]
Normalizes scan outputs from CMDB, code analyzers, and runtime telemetry.
Persists scoring metrics and rule outcomes.
[JDBC]
Routes 'refactor' candidates to FSM extraction.
[Event]
Pattern Recommender
[Component: Spring Service]
Maps applications to modernization patterns (rehost, refactor, retire, replace) and target reference architectures.
Delegates remediation pattern selection logic for final recommendation assembly.
[In-process]
Scoring Engine
[Component: Spring Service]
Calculates readiness and risk scores using configurable rules, NIST controls, and business weightings.
Assessment & Scoring Service
[Container]
Modernization Orchestrator
[Software System]
Component View: Modernization Orchestrator - Assessment & Scoring Service
Components in Assessment & Scoring Service.
Wednesday, December 10, 2025 at 2:34 PM Pacific Standard Time

Figure B-3: Assessment & Scoring Service Components

Modernization Portal Components

The user interface components including the Wave Dashboard, Assessment Workspace, Decision Center, FSM Validation UI (for SME review), and Ontology Browser (for Enterprise Architect use).

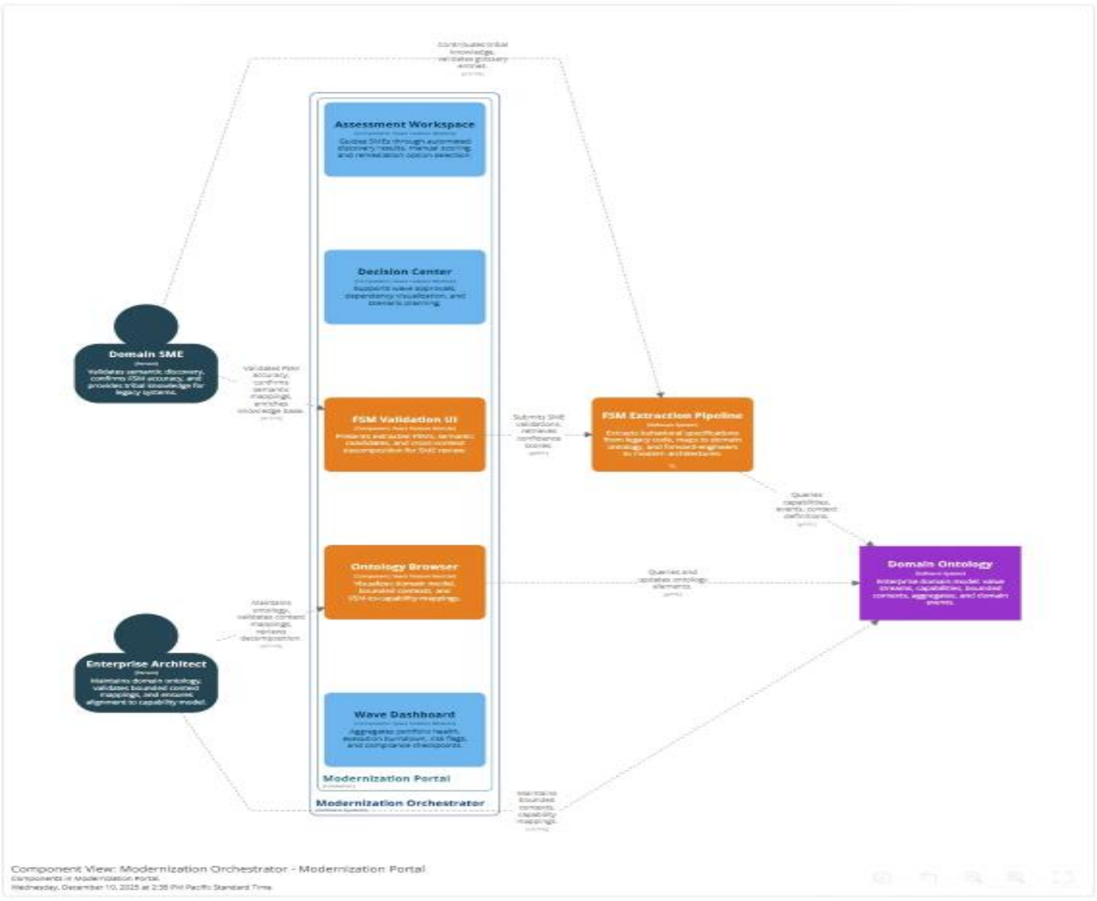
Contributes tribal knowledge, validates glossary entries
Assessment Workspace
Guides SMEs through automated discovery results, manual scoring and remediation option selection.
Decision Center
Supports wave approvals, dependency visualization, and scenario planning.
Domain SME
Validates semantic discovery, confirms FSM accuracy, and provides tribal knowledge for legacy systems.
Validates FSM accuracy, confirms semantic mappings, enriches knowledge base
FSM Validation UI
Presents extracted FSMs, semantic candidates, and cross-context decomposition for SME review.
Submits SME validations, retrieves confidence scores
FSM Extraction Pipeline
Extracts behavioral specifications from legacy code, maps to domain ontology, and forward-engineers to modern architectures.
Queries capabilities, events, context definitions
Ontology Browser
Visualizes domain model, bounded contexts, and FSM-to-capability mappings.
Queries and updates ontology elements
Domain Ontology
Enterprise domain model: value streams, capabilities, bounded contexts, aggregates, and domain events.
Maintains ontology, validates context mappings, reviews decomposition
Enterprise Architect
Maintains domain ontology, validates bounded context mappings, and ensures alignment to capability model.
Wave Dashboard
Aggregates portfolio health, execution burndown, risk flags, and compliance checkpoints.
Modernization Portal
Modernization Orchestrator
Maintains bounded contexts, capability mappings
Component View: Modernization Orchestrator - Modernization Portal
Components in Modernization Portal
Wednesday, December 10, 2025 at 2:38 PM Pacific Standard Time

Figure B-4: Modernization Portal Components