Contents

PATENT APPLICATION

System and Method for Multi-Domain Knowledge Coordination in Large Language Model-Assisted Software Development Using Canonical Domain Models with Explicit Grounding Relationships

---

APPLICATION TYPE: Nonprovisional Utility Patent Application

TECHNOLOGY CENTER: TC 2100 (Computer Architecture, Software, and Information Security)

CLASSIFICATION: - G06N 20/00 (Machine learning) - G06F 8/10 (Software development) - G06F 40/30 (Semantic analysis)

CROSS-REFERENCE TO RELATED APPLICATIONS

Not Applicable.

BACKGROUND OF THE INVENTION

Field of the Invention

The present invention relates generally to computer-implemented systems and methods for software development assistance using artificial intelligence, and more particularly to systems and methods for coordinating multiple knowledge domains using formal canonical domain models with explicit grounding relationships to constrain and validate large language model (LLM) generation of software artifacts, thereby ensuring cross-domain consistency and reducing integration defects in complex software systems.

The invention further relates to automated validation frameworks, data structures for representing domain knowledge with typed dependencies, and systematic workflows for human-in-the-loop software development from product vision through implementation using LLM-assisted generation with formal constraints.

Description of Related Art

The development of complex software systems increasingly requires coordination across multiple specialized knowledge domains, including domain-driven design (DDD), user experience (UX) design, quality engineering (QE), data engineering, and agile project management. Each domain has evolved its own vocabulary, patterns, constraints, and best practices. However, existing systems and methods lack formal mechanisms for representing and validating dependencies between these domains, leading to integration errors, inconsistencies, and rework.

Recent advances in large language models (LLMs) such as GPT-4, Claude, and others have demonstrated the capability to assist with software development tasks including code generation, design suggestion, and documentation creation. However, these LLM systems generate outputs that, while plausible, often violate domain-specific constraints and cross-domain consistency requirements in multi-domain systems.

Several approaches have been proposed to address aspects of these problems, but each has significant limitations:

Domain-Driven Design and Bounded Contexts Evans, E. (2003) in "Domain-Driven Design: Tackling Complexity in the Heart of Software" introduced the concept of bounded contexts as explicit boundaries within which a domain model applies. Vernon, V. (2013) in "Implementing Domain-Driven Design" expanded on tactical and strategic DDD patterns. These works established the importance of explicit domain boundaries and ubiquitous language within contexts.

However, DDD provides mechanisms for intra-context consistency but no formal framework for representing or validating cross-context dependencies. Dependencies between bounded contexts remain implicit and undocumented in the methodology. There is no standardized data structure for capturing these relationships, no automated validation of cross-context consistency, and no integration with modern LLM-based development tools. When multiple bounded contexts must coordinate (as is typical in enterprise systems), developers must manually ensure consistency without formal support.

Knowledge Representation and Ontologies The field of knowledge representation has developed sophisticated formalisms for representing domain knowledge, including Web Ontology Language (OWL), Resource Description Framework (RDF), and description logics. Upper ontologies such as SUMO (Suggested Upper Merged Ontology) and DOLCE (Descriptive Ontology for Linguistic and Cognitive Engineering) provide frameworks for universal concepts. Domain-specific ontologies have been developed for healthcare (SNOMED CT), legal reasoning, and scientific domains.

While ontologies provide formal semantic reasoning capabilities, they have significant limitations for practical software engineering. Ontologies are heavyweight formalisms requiring specialized expertise in description logic and semantic reasoning. The learning curve is steep, adoption in software engineering practice is limited, and tool support is primarily research-oriented rather than production-ready. Most critically, existing ontology systems have no integration with LLM generation systems and provide no mechanisms for constraining or validating LLM outputs. The focus of ontologies is on semantic reasoning (e.g., class subsumption, consistency checking in logic) rather than practical software development workflows.

Large Language Model Systems for Code Generation Foundation models such as GPT-4 (OpenAI), Claude (Anthropic), and Llama (Meta) have demonstrated impressive capabilities for code generation and software development assistance. Commercial systems including GitHub Copilot, AWS CodeWhisperer, and others integrate LLMs into development environments for code completion and generation.

However, these systems generate plausible code without formal validation of domain-specific constraints or cross-domain consistency. The LLMs have no awareness of enterprise-specific domain models, architectural constraints, or cross-component contracts. Generation is unconstrained except by the training data and generic prompts. This leads to several problems:

1. **Consistency violations**: Generated UX components may reference non-existent domain concepts
2. **Constraint violations**: Generated aggregates may violate domain invariants or transactional boundaries
3. **Integration failures**: Generated components from different domains may use incompatible assumptions
4. **Validation gaps**: No automated way to detect these issues before code review or testing

While developers can manually review and correct these issues, this is time-consuming and error-prone, especially in large systems with multiple domains.

Schema-Guided Generation Recent research has explored using schemas to constrain LLM generation. Xu et al. (2024) demonstrated that providing JSON Schema constraints in the LLM context improves generation accuracy for structured outputs in single-domain tasks. Various constrained decoding approaches use context-free grammars (CFGs) or other formal grammars to ensure syntactic correctness of LLM outputs.

However, existing schema-guided approaches have critical limitations:

1. **Single-domain focus**: Schemas apply to one domain at a time with no cross-domain coordination
2. **No relationship formalization**: When multiple schemas are used, their relationships are implicit
3. **No consistency validation**: Systems can validate conformance to individual schemas but not cross-schema consistency
4. **No dependency management**: If Schema A references concepts from Schema B, there is no formal mechanism to validate this dependency or ensure Schema B is available

For example, if an LLM generates a UX workflow that references a DDD aggregate, existing schema-guided systems cannot validate that: (a) the referenced aggregate actually exists in the DDD model, (b) the workflow respects the aggregate's transactional boundaries, or (c) changes to the aggregate schema are propagated to the UX workflow specification. These cross-domain consistency requirements are critical in enterprise systems but are not addressed by prior art.

Retrieval-Augmented Generation (RAG) RAG systems augment LLM generation by retrieving relevant documents or code snippets from vector databases based on semantic similarity. Systems like Pinecone, Weaviate, and others provide embedding-based retrieval of relevant context.

RAG addresses the problem of incorporating existing knowledge into LLM generation, but has significant limitations:

1. **No formal validation**: Retrieved documents are selected by similarity, not formal correctness
2. **No consistency guarantees**: Multiple retrieved documents may contain contradictory information
3. **No structure awareness**: RAG treats all content as unstructured text, losing domain structure
4. **No dependency tracking**: Cannot ensure retrieved information about Domain A is consistent with Domain B

RAG systems cannot ensure that retrieved information is mutually consistent across domains or that generated artifacts respect the dependencies between domains. The similarity-based retrieval provides relevant context but no formal validation.

Model-Driven Development (MDD) Model-driven development approaches use models (typically UML) to generate code skeletons and documentation. Domain-specific languages (DSLs) provide specialized syntax for particular domains. Tools like Enterprise Architect, MagicDraw, and others support model-to-code transformation.

MDD limitations for multi-domain systems include:

1. **Single-domain focus**: UML models typically represent one domain (e.g., class structure) without formal cross-domain coordination
2. **No LLM integration**: MDD tools predate modern LLMs and provide no integration or constraint mechanisms
3. **Manual model creation**: Requires significant manual effort to create and maintain models
4. **Static generation**: Code generation is deterministic template-based, not adaptive like LLM generation

MDD cannot leverage the flexibility and natural language understanding of LLMs, and provides no mechanisms to constrain LLM generation based on formal models.

Enterprise Architecture Frameworks Enterprise architecture frameworks such as ArchiMate, TOGAF, and the C4 model provide metamodels and notations for documenting system architecture. ArchiMate 3.1, for example, defines relationships between business, application, and technology layers.

However, these frameworks are documentation-focused rather than validation-focused:

1. **Static documentation**: Diagrams and specifications document intended architecture but are not executable

2. **No automated validation**: Cannot automatically check if implementation conforms to architecture
3. **No LLM integration**: No mechanisms to use architecture specifications to constrain code generation
4. **No consistency checking**: Relationships between architectural elements are documented but not formally validated

Architecture documentation becomes stale as systems evolve, and there is no automated way to ensure generated code respects documented architectural constraints.

Summary of Prior Art Limitations The prior art fails to provide:

1. **Formal cross-domain dependency representation**: No standard data structure for explicit, typed dependencies between domain models with validation rules
2. **Automated multi-domain consistency validation**: No system for checking that artifacts spanning multiple domains satisfy consistency requirements
3. **LLM constraint mechanisms for multi-domain generation**: No method for simultaneously constraining LLM generation using multiple coordinated domain models
4. **Completeness metrics**: No quality metrics indicating whether domain models are sufficiently complete for production use
5. **Systematic multi-domain workflow**: No end-to-end process from requirements through implementation with formal validation at each step
6. **Automated impact analysis**: No automated method for determining which domains are affected by changes to a domain model and coordinating updates

These limitations result in: - Integration defects when domains use inconsistent assumptions - Rework when LLM-generated artifacts violate cross-domain constraints - Manual validation effort to ensure multi-domain consistency - Lack of quality metrics for domain model readiness - Difficulty coordinating changes across multiple domains

Need for the Invention

There is a need for a computer-implemented system and method that provides:

1. **Formal data structures** for representing domain knowledge with explicit, typed cross-domain dependencies
2. **Automated validation framework** for checking multi-domain consistency with quantitative completeness metrics
3. **LLM constraint mechanism** that simultaneously applies constraints from multiple coordinated domain models during generation
4. **Systematic development workflow** from product vision through implementation with formal validation gates

5. **Automated impact analysis and ripple effect management** when domain models change
6. **Quality metrics** such as closure percentage indicating domain model completeness
7. **Human-in-the-loop refinement** integrating subject matter expert critique with automated LLM regeneration

The present invention addresses these needs by providing canonical domain models with explicit grounding relationships, automated closure validation, LLM-constrained generation with cross-domain consistency checking, and systematic workflows for multi-domain software development.

BRIEF SUMMARY OF THE INVENTION

The present invention provides a computer-implemented system and method for multi-domain knowledge coordination in large language model (LLM) assisted software development. The invention addresses the critical problem of ensuring cross-domain consistency when LLMs generate software artifacts spanning multiple specialized knowledge domains such as domain-driven design (DDD), user experience (UX) design, quality engineering (QE), data engineering, and agile project management.

Core Innovations

The invention introduces several novel and non-obvious innovations:

1. Canonical Domain Model Data Structure A formal data structure for representing domain knowledge, comprising:

- **Unique identifier and semantic version**: Enables versioning and dependency management
- **Set of concepts**: Core abstractions within the domain (e.g., “Aggregate”, “BoundedContext” in DDD)
- **Set of patterns**: Reusable structural templates (e.g., “Repository pattern”, “Event Sourcing”)
- **Set of constraints**: Validation rules with severity levels (error, warning, info) and executable validators
- **Set of grounding relationships**: Explicit, typed dependencies to other canonical domain models
- **Ubiquitous language**: Canonical vocabulary with definitions
- **Layer designation**: Classification as foundation, derived, or meta-level model
- **Evolution history**: Tracking changes and migrations across versions

This data structure is novel in providing a complete, self-contained representation of domain knowledge that explicitly declares dependencies on other domains through grounding relationships, enabling automated consistency validation across domains.

2. Grounding Relationship Data Structure A formal data structure for representing explicit dependencies between canonical domain models, comprising:

- **Source and target model identifiers**: Defines directed dependency
- **Grounding type**: One of four types:
  - **Structural**: Target provides foundational concepts that source builds upon
  - **Semantic**: Target provides meaning or interpretation for source concepts
  - **Procedural**: Target defines processes that source follows or validates
  - **Epistemic**: Target provides assumptions or justifications for source
- **Concept mapping set**: Explicit pairings of source concepts to target concepts with cardinality constraints
- **Translation map**: Terminology mappings between models with semantic distance metrics
- **Strength designation**: Strong (hard constraint), weak (soft guidance), or optional (informational)
- **Validation rules**: Executable functions checking consistency between source and target

This grounding structure is novel in providing typed, explicit cross-domain dependencies with automated validation, enabling formal reasoning about multi-domain consistency.

3. Closure Property and Validation Method A method for calculating and validating the completeness of a canonical domain model, comprising:

- Identifying all concept references within the model (internal and external)
- For each external reference, verifying existence of a grounding relationship to the target model
- For each grounding, verifying the specific referenced concept is included in the mapping
- Calculating closure percentage: (internal references + grounded external references) / total references $\times$ 100%
- Comparing closure percentage to quality thresholds (95% for production-ready, 80% minimum)
- Generating validation report with ungrounded references identified

This closure property provides a novel quality metric for domain model completeness that is algorithmically computable and predictive of downstream integration defects. Empirical validation shows strong negative correlation ($r = -0.96$) between closure percentage and defect rates.

4. LLM-Constrained Generation Method A computer-implemented method for generating software artifacts using LLM with multi-domain constraints, comprising four phases:

Phase 1 - Schema Loading: - Identify required canonical domain models from task description - Load primary model schemas - Resolve transitive grounding dependencies (if Model A grounds in Model B, and Model B grounds in Model C, load all three) - Construct unified schema context merging concepts, patterns, constraints, and validation rules - Validate schema context for acyclicity (no circular dependencies) and constraint consistency (no contradictions)

Phase 2 - Constrained Generation: - Inject unified schema context into LLM prompt - Generate k candidate artifact continuations using beam search - For each candidate, validate against all active constraints in real-time - Prune candidates that violate constraints - Select highest-probability valid candidate - Iterate until artifact is complete

Phase 3 - Validation: - Syntactic validation: Parse artifact and verify data types - Semantic validation: Resolve all concept references and check constraint rules - Cross-domain consistency validation: For each external reference, verify supporting grounding relationship exists and concept mapping is valid - Generate validation report with errors (hard failures) and warnings (soft failures)

Phase 4 - Explanation Generation: - Build justification trace linking each design decision to schema elements - Generate hierarchical rationale with schema citations - Document rejected alternatives and reasons for rejection - Provide validation evidence showing constraint satisfaction

This method is novel in simultaneously constraining LLM generation using multiple formally related domain models with automated cross-domain consistency validation during generation, not just post-generation. Empirical results demonstrate 25-50% accuracy improvement over unconstrained baselines.

5. Automated Ripple Effect Management A method for coordinated update of multiple canonical domain models when changes occur, comprising:

- Detecting change to a canonical domain model (concept added/modified/removed)
- Traversing grounding graph following all incoming grounding relationships (models that depend on the changed model)
- Identifying affected concepts and validation failures in dependent models
- Generating impact report showing all affected models and specific failures
- Receiving user approval to proceed with ripple effect propagation
- For each affected model, using LLM to regenerate affected artifacts with updated constraints reflecting the change
- Validating all regenerated artifacts
- Updating model versions with semantic versioning (major/minor/patch bumps)
- Recalculating system-wide closure percentage

This automated ripple effect management is novel in using the grounding graph structure to identify impacts and coordinate LLM-based regeneration across multiple models, maintaining system-wide consistency when individual models evolve.

6. Greenfield Development Workflow System A systematic computer-implemented workflow orchestrating LLM generation with human oversight across nine phases:

1. **Vision Definition**: Validate product vision using Agile canonical model (completeness check for problem, users, value, metrics, constraints, assumptions)
2. **Strategic DDD Model**: Generate bounded contexts, context maps, ubiquitous language with domain expert review
3. **Epic/Feature Decomposition**: Decompose vision into epics and features with grounding validation (Epic must reference BoundedContext)
4. **User Story Generation**: Generate stories with acceptance criteria and workflows, grounded in DDD and UX models
5. **QE Model Refinement**: Generate test strategy and test cases validating domain invariants and workflows
6. **UX Model Refinement**: Generate information architecture, pages, components, and workflows grounded in bounded contexts
7. **Data-Eng Model Definition**: Generate datasets, schemas, pipelines with semantic alignment to DDD aggregates (70% threshold)
8. **Bounded Code Generation**: Generate implementation code constrained by all domain models with automated test execution
9. **Continuous Evolution**: Handle changes with impact analysis and coordinated multi-model updates

This end-to-end workflow is novel in providing systematic progression from high-level vision to implementation with formal validation at each step, automated cross-domain consistency checking, and human-in-the-loop review gates.

Advantages Over Prior Art

The invention provides significant advantages demonstrated through empirical validation:

1. **Accuracy**: 25-50% improvement in LLM generation accuracy for multi-domain tasks (average 41% across 75 experiments)
2. **Consistency**: 92% cross-domain consistency vs. 44% for unconstrained baselines
3. **Speed**: 4.7x faster solution synthesis time (9 minutes vs. 42 minutes average)
4. **Explanation Quality**: 100% improvement in justification quality (4.6/5 vs. 2.3/5 expert rating)

5. **Defect Reduction**: 3x fewer integration defects when closure >95% vs. <80%
6. **Entropy Reduction**: 50% reduction in concept distribution entropy (2.1 bits vs. 4.2 bits), indicating more constrained and consistent outputs
7. **ROI**: Break-even return on investment after 4-5 features (200 hour upfront investment amortized by 32 hour per-feature savings)

These improvements are achieved through the combination of formal domain models, explicit typed grounding relationships, automated closure validation, and LLM-constrained generation with real-time cross-domain consistency checking—a combination not taught or suggested by any prior art reference.

Scope of the Invention

The invention is applicable to complex software systems requiring coordination across multiple knowledge domains, including enterprise applications, healthcare systems, financial platforms, and government systems. The invention is particularly valuable in regulated industries where consistency, traceability, and validation are critical.

The invention works with any large language model API (GPT-4, Claude, Llama, etc.) and can be integrated into existing development workflows through IDE plugins, command-line tools, or orchestration systems such as LangGraph.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a system architecture diagram showing the major components of the canonical grounding system including model repository, validation engine, LLM interface, schema context builder, generation controller, ripple effect analyzer, and workflow orchestrator.

Figure 2 is a grounding graph diagram showing five canonical domain models (DDD, Data Engineering, UX, QE, Agile) with 19 explicit grounding relationships represented as directed edges, forming a directed acyclic graph with layered structure.

Figure 3 is a flowchart illustrating the closure percentage calculation method, showing steps for identifying concept references, classifying as internal or external, verifying grounding relationships, and calculating completeness percentage.

Figure 4 is a flowchart illustrating the four-phase LLM-constrained generation method: schema loading, constrained generation with beam search, multi-level validation, and explanation generation.

Figure 5 is a flowchart illustrating the ripple effect management process: change detection, grounding graph traversal, impact analysis, user approval, coordinated regeneration, and validation.

Figure 6 is a workflow diagram showing the nine-phase greenfield development process from product vision through strategic domain modeling, epic decomposition, user story generation, QE refinement, UX refinement, data engineering modeling, implementation, and continuous evolution.

Figure 7 is a data structure diagram showing the canonical domain model 7-tuple representation with components: identifier, domain, concepts, patterns, constraints, grounding relationships, and version.

Figure 8 is a data structure diagram showing the grounding relationship 6-tuple representation with components: source, target, type, concept mappings, strength, and validation rules.

Figure 9 is an example canonical domain model instance for Domain-Driven Design showing 13 concepts including BoundedContext, Aggregate, Entity, ValueObject, DomainEvent, Repository, and others with their properties and relationships.

Figure 10 is an example grounding relationship instance showing UX model structurally grounded in DDD model with specific concept mappings: UX.Page $\rightarrow$ DDD.BoundedContext (1:1) and UX.Workflow $\rightarrow$ DDD.DomainService (1:*).

Figure 11 is a deployment diagram showing system components deployed in cloud environment with containers for validation engine, LLM gateway, model repository, and workflow orchestrator, with external connections to LLM APIs and version control systems.

Figure 12 is a screenshot of a visual model navigator user interface showing a canonical domain model as a series of interconnected pages with concepts, patterns, constraints, and grounding relationships navigable through hyperlinks.

DETAILED DESCRIPTION OF THE INVENTION

Overview

The present invention provides a comprehensive system and method for coordinating multi-domain knowledge in LLM-assisted software development. The invention comprises novel data structures, algorithms, and workflows that enable formal representation of domain knowledge, explicit cross-domain dependencies, automated consistency validation, and constrained LLM generation with cross-domain awareness.

The following detailed description explains the invention's components, their interactions, implementation details, and operational procedures. The description includes specific algorithms, data structures, and examples that enable a person of ordinary skill in the art (POSITA) to make and use the invention without undue experimentation.

System Architecture

Referring to Figure 1, the canonical grounding system comprises the following interconnected components:

Model Repository (Component 110) A data storage system for managing versioned canonical domain model schemas. The repository stores each canonical domain model as a structured document (YAML or JSON format) conforming to a meta-schema specification (JSON Schema 2020-12).

The repository provides: - Version control integration with Git or similar systems - Semantic versioning (major.minor.patch) for each model - Concurrent access with optimistic locking - Query capabilities for finding models by ID, domain, or layer - Validation on commit ensuring meta-schema conformance

Grounding Map (Component 120) A directed acyclic graph (DAG) data structure representing all grounding relationships between canonical domain models. The graph is maintained as a separate structured document (YAML/JSON) and synchronized with the model repository.

The grounding map enables: - Efficient traversal for transitive dependency resolution - Acyclicity validation using topological sort - Impact analysis by following incoming/outgoing edges - Visualization generation (Graphviz DOT format) - Metrics calculation (betweenness centrality, clustering coefficient)

Validation Engine (Component 130) A computational component that performs automated validation operations:

Closure Calculation: Implements the closure percentage algorithm (detailed below) to determine model completeness. Executes in O(n*m) time where n = number of concepts and m = average references per concept.

Acyclicity Checking: Uses Tarjan's strongly connected components algorithm to detect cycles in the grounding graph. Executes in O(V+E) time where V = number of models (vertices) and E = number of groundings (edges).

Constraint Validation: Evaluates all constraint predicates in applicable canonical domain models against candidate artifacts. Maintains a constraint satisfaction engine with support for boolean expressions, cardinality checking, and custom validators.

Grounding Validation: For each external concept reference in an artifact, verifies: 1. Source model declares grounding to target model 2. Grounding includes concept mapping for specific referenced concept 3. Cardinality constraints are satisfied 4. Grounding-specific validation rules pass

LLM Interface (Component 140) An adapter component providing unified interface to multiple LLM APIs:

- OpenAI API (GPT-4, GPT-4 Turbo)
- Anthropic API (Claude 3.5 Sonnet, Claude 3 Opus)
- Local inference engines (vLLM, Ollama)
- Custom fine-tuned models

The interface handles: - API authentication and rate limiting - Prompt construction with schema context injection - Streaming response handling - Token counting and cost tracking - Retry logic with exponential backoff - Response parsing and validation

Schema Context Builder (Component 150) A component that constructs unified schema context for LLM consumption:

Input: Set of required canonical domain model identifiers from task analysis

Process: 1. Load primary models from repository 2. Extract grounding relationships from each model 3. Recursively load transitively referenced models (if A grounds in B, and B grounds in C, load C) 4. Detect and halt if circular dependency found 5. Merge concepts into unified vocabulary with qualified names (model_id.concept_id) 6. Merge patterns with applicability rules 7. Merge constraints with severity levels 8. Build grounding relationship graph for context 9. Validate for consistency (no contradictory constraints)

Output: Unified schema context data structure containing: - Set of all loaded canonical domain models - Combined concept vocabulary (10,000-50,000 tokens typical) - Combined pattern library (5,000-15,000 tokens typical) - Combined constraint set (5,000-20,000 tokens typical) - Grounding graph for reference resolution - Total size: 20,000-100,000 tokens depending on number of models

The context builder implements caching to avoid redundant loading and supports incremental updates when models change.

Generation Controller (Component 160) An orchestration component managing LLM-constrained generation:

Input: Task description and unified schema context

Process (detailed algorithm provided below): 1. Construct prompt by combining task description + schema context 2. Initialize beam search with width k=5 (configurable) 3. Iteratively generate candidate continuations 4. Validate each candidate using validation engine 5. Prune invalid candidates 6. Select highest-probability valid candidate 7. Append to artifact and repeat until complete 8. Perform final multi-level validation 9. Generate explanation and justification trace

Output: Validated artifact with provenance metadata

The controller implements timeout handling (configurable, default 300 seconds), error recovery with retry, and logging of all generation attempts for debugging.

Ripple Effect Analyzer (Component 170) A component for impact analysis and coordinated updates:

Input: Model change specification (model ID, changed concept, change type)

Process (detailed algorithm provided below): 1. Build reverse grounding graph (incoming edges) 2. Identify models with groundings to changed model 3. For each dependent model, check if it references changed concept 4. Run validation on affected models with proposed change 5. Collect validation failures 6. Generate impact report 7. Await user approval 8. If approved, orchestrate regeneration for each affected model 9. Validate regenerated artifacts 10. Update model versions 11. Commit to repository

Output: Updated models with version bumps, impact report, audit trail

Workflow Orchestrator (Component 180) A state machine managing the nine-phase greenfield development workflow:

Implementation: LangGraph or equivalent workflow engine

State Transitions: Each phase has entry conditions, execution logic, exit criteria, and failure handling

Persistence: Workflow state persisted to enable pause/resume

Audit Trail: All phase transitions, approvals, and artifacts logged

Visualization Renderer (Component 190) A component generating human-readable representations:

Graphviz Generation: Converts grounding graph to DOT format with clustering, edge labels, and color coding by grounding type

Markdown Export: Transforms canonical domain models to structured markdown with cross-references, tables for concepts, and examples

Interactive UI: Web application (React-based) displaying models as navigable pages with hyperlinked concepts, embedded examples, and validation status indicators

Computing Environment

The system is designed for deployment in cloud environments (AWS, Azure, GCP) or on-premises infrastructure:

Hardware Requirements: - Multi-core CPU: 16+ cores recommended for parallel validation - Memory: 64GB+ RAM for holding multiple models and LLM context - Storage: SSD-based storage for model repository (100GB+ for large deployments) - Network: High-bandwidth connection for LLM API calls (1Gbps+) - Optional: GPU for local LLM inference (NVIDIA A100 or equivalent)

Software Stack: - Operating System: Linux (Ubuntu 22.04+), macOS, or Windows Server - Runtime: Python 3.10+ for validation engine and orchestration - Database: PostgreSQL or similar for workflow state persistence - Cache: Redis for schema context caching - Version Control: Git with LFS for large schemas - Containerization: Docker with Kubernetes orchestration

Scalability: - Model repository scales to 100+ canonical domain models - Grounding graph efficiently handles 1000+ grounding relationships using graph database (Neo4j optional) - Validation parallelizable across worker nodes - LLM calls rate-limited and cached for repeated queries - Horizontal scaling via Kubernetes for high-availability deployment

Canonical Domain Model Data Structure

Referring to Figure 7, the canonical domain model is represented as a data structure comprising seven primary components (7-tuple):

Formal Definition

```
CanonicalDomainModel := {
  canonical_model_id: String
  domain: String
  layer: Enum{foundation, derived, meta}
  concepts: Set<Concept>
  patterns: Set<Pattern>
  constraints: Set<Constraint>
  grounding: Set<GroundingReference>
  version: SemanticVersion
  metadata: Metadata
}
```

Component Specifications **canonical_model_id** (String, required, unique): - Format: `model_` + domain abbreviation (e.g., "model_ddd", "model_ux") - Constraints: Must be unique across system, lowercase, alphanumeric plus underscore only - Purpose: Unambiguous identifier for referencing this model in grounding relationships

domain (String, required): - Human-readable domain name (e.g., "Domain-Driven Design", "User Experience") - Purpose: Documentation and display

layer (Enum, required): - Values: `foundation`, `derived`, `meta` - **foundation**: Models with no grounding dependencies (e.g., DDD, Data Engineering) - **derived**: Models grounding in foundation models (e.g., UX, QE) - **meta**: Models coordinating derived models (e.g., Agile) - Purpose: Architectural layering, validation of acyclicity

concepts (Set, required, non-empty):

Each concept represents a core abstraction in the domain:

```
Concept := {
  concept_id: String
  name: String
  description: String
  properties: Set<Property>
  relationships: Set<Relationship>
  examples: Set<String>
}

Property := {
  property_id: String
  name: String
  type: DataType
  cardinality: String
  constraints: Set<Constraint>
  description: String
}

Relationship := {
  relationship_type: Enum{references, contains, implements, validates}
  target_concept: ConceptReference
  cardinality: String
  description: String
}

ConceptReference := {
  model_id: String
  concept_id: String
}
```

Example Concept (DDD BoundedContext):

```
concepts:
  - concept_id: BoundedContext
    name: Bounded Context
    description: "Explicit boundary within which a domain model applies. Defines the applica
    properties:
      - property_id: context_id
        name: Context Identifier
        type: string
        cardinality: "1..1"
        constraints:
          - description: "Must be unique within system"
            rule: "unique(context_id)"
      - property_id: name
        name: Context Name
        type: string
        cardinality: "1..1"
      - property_id: responsibility
        name: Responsibility Statement
        type: string
        cardinality: "1..1"
        description: "Clear statement of what this context is responsible for"
      - property_id: core_concepts
        name: Core Concepts
        type: array<string>
        cardinality: "1..*"
        description: "Primary domain concepts managed by this context"
    relationships:
      - relationship_type: contains
        target_concept:
          model_id: model_ddd
          concept_id: Aggregate
        cardinality: "1..*"
        description: "Bounded context contains one or more aggregates"
      - relationship_type: contains
        target_concept:
          model_id: model_ddd
          concept_id: UbiquitousLanguage
        cardinality: "1..1"
        description: "Each context has its own ubiquitous language"
    examples:
      - "OrderManagement context responsible for order lifecycle"
      - "Inventory context responsible for stock tracking"
```

patterns (Set, optional):

Patterns are reusable structural templates:

```
Pattern := {
  pattern_id: String
  name: String
  intent: String
  structure: String
  participants: Set<ConceptReference>
  consequences: String
  examples: Set<String>
  applicability: BooleanExpression
}
```

Example Pattern (Repository):

```
patterns:
  - pattern_id: Repository
    name: Repository Pattern
    intent: "Mediate between the domain and data mapping layers using a collection-like inte
    structure: "Interface defining collection-like operations (add, remove, find) for aggreg
    participants:
      - model_id: model_ddd
        concept_id: Aggregate
      - model_id: model_ddd
        concept_id: Entity
    consequences: "Provides clean separation between domain and persistence. Enables testing
    examples:
      - "OrderRepository provides findById, findByCustomer, save methods"
    applicability: "aggregate.persistence_strategy == 'repository'"
```

constraints (Set, optional but recommended):

Constraints define validation rules:

```
Constraint := {
  constraint_id: String
  description: String
  rule: BooleanExpression
  severity: Enum{error, warning, info}
  validator: ValidatorFunction
  message: String
}

ValidatorFunction := Artifact → Boolean
```

Example Constraints:

```
constraints:
  - constraint_id: AGGREGATE_HAS_ROOT
    description: "Every aggregate must have exactly one root entity"
    rule: "count(aggregate.entities where is_root == true) == 1"
    severity: error
    message: "Aggregate {aggregate.id} has {count} root entities, expected exactly 1"

  - constraint_id: AGGREGATE_IDENTITY_REFERENCES
    description: "Aggregates should reference other aggregates by identity, not direct objec
    rule: "forall(relationship in aggregate.relationships where relationship.target.is_aggre
    severity: warning
    message: "Aggregate {aggregate.id} directly references aggregate {target.id}; consider i
```

grounding (Set):

References to grounding relationships (detailed below):

```
GroundingReference := {
  grounding_id: String
}
```

The full grounding relationships are stored in the separate grounding map (Component 120) to avoid circular dependencies in model loading.

version (SemanticVersion, required):

```
SemanticVersion := {
  major: Integer
  minor: Integer
  patch: Integer
}
```

Versioning follows semantic versioning 2.0 specification: - **Major**: Breaking changes (concepts removed, constraint tightened) - **Minor**: Backward-compatible additions (new concepts, new patterns) - **Patch**: Bug fixes (typos, clarifications)

metadata (Metadata):

```
Metadata := {
  schema_date: Date
  authors: Set<String>
  schema_purpose: String
  evolution_history: Set<ChangeRecord>
  tags: Set<String>
}

ChangeRecord := {
  version: SemanticVersion
  date: Date
  description: String
  breaking_changes: Boolean
  migration_guide: String
}
```

Storage Format Canonical domain models are stored as YAML or JSON files conforming to JSON Schema 2020-12 meta-schema:

File Naming Convention: `{canonical_model_id}.yaml` or `{canonical_model_id}.json`

Meta-Schema Location: Stored in repository as `meta-schema.json`

Validation: Every model validated against meta-schema on commit using jsonschema library (Python) or equivalent

Example File (domains/ddd/model-schema.yaml):

```yaml
$schema: "https://json-schema.org/draft/2020-12/schema"
canonical_model_id: model_ddd
domain: "Domain-Driven Design"
layer: foundation
```

```yaml
version:
  major: 1
  minor: 0
  patch: 0

concepts:
  - concept_id: BoundedContext
    name: "Bounded Context"
    # ... (as shown above)

  - concept_id: Aggregate
    name: "Aggregate"
    description: "Cluster of entities and value objects with transactional consistency bounc
    properties:
      - property_id: aggregate_id
        name: "Aggregate Identifier"
        type: string
        cardinality: "1..1"
      - property_id: root_entity
        name: "Aggregate Root Entity"
        type: reference
        cardinality: "1..1"
        constraints:
          - description: "Root must be an Entity"
            rule: "root_entity.type == 'Entity'"
    relationships:
      - relationship_type: contains
        target_concept:
          model_id: model_ddd
          concept_id: Entity
        cardinality: "1..*"
    examples:
      - "Order aggregate with OrderLine entities"

  # ... additional 11 concepts

patterns:
  - pattern_id: Repository
    # ... (as shown above)

constraints:
  - constraint_id: AGGREGATE_HAS_ROOT
    # ... (as shown above)

grounding: []  # Foundation model, no groundings
```

```
metadata:
  schema_date: "2024-10-01"
  authors: ["Domain experts"]
  schema_purpose: "Formal representation of DDD patterns and concepts"
```

Grounding Relationship Data Structure

Referring to Figure 8, grounding relationships are represented as a data structure comprising six primary components (6-tuple):

Formal Definition

```
GroundingRelationship := {
  grounding_id: String
  source: CanonicalModelID
  target: CanonicalModelID | Set<CanonicalModelID>
  type: GroundingType
  concept_mappings: Set<ConceptMapping>
  strength: GroundingStrength
  validation_rules: Set<ValidationRule>
  metadata: GroundingMetadata
}
```

Component Specifications **grounding_id** (String, required, unique): - Format: `grounding_{source}_{target}_{sequence}` (e.g., "grounding_ux_ddd_001") - Purpose: Unique identifier for referencing this grounding

source (CanonicalModelID, required): - The canonical domain model that has dependencies (e.g., "model_ux")

target (CanonicalModelID or Set, required): - The canonical domain model(s) being depended upon (e.g., "model_ddd") - Can be single target or set for multi-target groundings

type (GroundingType, required):

```
GroundingType := Enum {
  structural,
  semantic,
  procedural,
  epistemic
}
```

Grounding Type Definitions:

1. **structural**: Target provides foundational concepts that source builds upon
   - Characteristics: Source concepts are defined in terms of target concepts; strong dependency

   - Example: UX.Page structurally grounds in DDD.BoundedContext (page is scoped to a context)
   - Validation: Every source concept using target concepts must declare structural grounding
2. **semantic**: Target provides meaning or interpretation for source concepts
   - Characteristics: Source and target describe the same or related entities with different perspectives
   - Example: Data-Eng.Dataset semantically aligns with DDD.Aggregate (dataset schema mirrors aggregate attributes)
   - Validation: Semantic alignment score 70% (measured by attribute overlap)
3. **procedural**: Target defines processes, workflows, or sequences that source follows or validates
   - Characteristics: Source implements, validates, or depends on target-defined procedures
   - Example: QE.TestCase procedurally grounds in DDD.Invariant (test validates invariant)
   - Validation: Source procedures cover all mandatory target procedures
4. **epistemic**: Target provides assumptions, justifications, or foundational knowledge for source
   - Characteristics: Source's rationale or scope derives from target
   - Example: Agile.Epic epistemically grounds in DDD.BoundedContext (epic scope justified by context)
   - Validation: Source epistemic references are documented and valid

concept_mappings (Set, required, non-empty):

```
ConceptMapping := {
  mapping_id: String
  source_concept: ConceptReference
  target_concept: ConceptReference
  mapping_type: MappingType
  cardinality: String
  bidirectional: Boolean
  description: String
}

MappingType := Enum {
  implements,
  validates,
  references,
  aligns,
  derives_from
}
```

Example Concept Mappings (UX → DDD):

```
concept_mappings:
  - mapping_id: ux_page_ddd_context
    source_concept:
      model_id: model_ux
      concept_id: Page
    target_concept:
      model_id: model_ddd
      concept_id: BoundedContext
    mapping_type: references
    cardinality: "1..1"
    bidirectional: false
    description: "Every UX Page must reference exactly one DDD BoundedContext defining its c

  - mapping_id: ux_workflow_ddd_service
    source_concept:
      model_id: model_ux
      concept_id: Workflow
    target_concept:
      model_id: model_ddd
      concept_id: DomainService
    mapping_type: implements
    cardinality: "1..*"
    bidirectional: false
    description: "UX Workflow implements one or more DDD DomainService operations"
```

strength (GroundingStrength, required):

```
GroundingStrength := Enum {
  strong,
  weak,
  optional
}
```

- **strong**: Hard constraint; violation is an error; must be satisfied
- **weak**: Soft guidance; violation is a warning; should be satisfied
- **optional**: Informational only; violation is info-level message; may be satisfied

validation_rules (Set, optional):

```
ValidationRule := {
  rule_id: String
  description: String
  validator: (SourceArtifact, TargetArtifact) → ValidationResult
  error_message: String
}

ValidationResult := {
  passed: Boolean
  message: String
  violations: Set<Violation>
}

Violation := {
  location: String
  expected: String
  actual: String
  severity: Enum{error, warning, info}
}
```

Example Validation Rule:

```yaml
validation_rules:
  - rule_id: VR_UX_001
    description: "Verify every UX Page references a valid DDD BoundedContext"
    validator: |
      function validate(ux_artifact, ddd_model):
        for each page in ux_artifact.pages:
          context_ref = page.bounded_context_ref
          if context_ref not in ddd_model.bounded_contexts:
            return ValidationResult(
              passed=false,
              message=f"Page {page.id} references invalid context {context_ref}",
              violations=[Violation(
                location=f"page.{page.id}.bounded_context_ref",
                expected="Valid BoundedContext ID",
                actual=context_ref,
                severity=error
              )]
            )
        return ValidationResult(passed=true)
    error_message: "Invalid BoundedContext reference in UX Page"
```

metadata (GroundingMetadata):

```
GroundingMetadata := {
  created_date: Date
  authors: Set<String>
  rationale: String
  related_patterns: Set<PatternReference>
  examples: Set<String>
}
```

Storage Format All grounding relationships are stored in a single grounding map file:

File Location: `research-output/interdomain-map.yaml`

Structure:

```
version: "2.0.0"
metadata:
  total_canonical_models: 5
  total_groundings: 19
  terminology_version: "2.0.0"

canonical_models:
  - model_id: model_ddd
    layer: foundation
  - model_id: model_data_eng
    layer: foundation
  - model_id: model_ux
    layer: derived
  - model_id: model_qe
    layer: derived
  - model_id: model_agile
    layer: meta

groundings:
  - grounding_id: grounding_ux_ddd_001
    source: model_ux
    target: model_ddd
    type: structural
    strength: strong
    concept_mappings:
      - mapping_id: ux_page_ddd_context
        source_concept: "model_ux.Page"
        target_concept: "model_ddd.BoundedContext"
        cardinality: "1..1"
        mapping_type: references
    # ... additional mappings

  # ... additional 18 groundings
```

Grounding Graph Properties The grounding map forms a directed acyclic graph (DAG) with the following properties:

Property 1 - Acyclicity: No path exists such that: model_A → model_B → … → model_N → model_A

Verified using Tarjan's algorithm during validation.

Property 2 - Layering: - All groundings from foundation models: none (no outgoing edges) - All groundings from derived models: target foundation models - All groundings from meta models: target foundation or derived models - No grounding may target a model in the same or higher layer

Property 3 - Transitivity (for strong groundings): If model_A strongly grounds in model_B, and model_B strongly grounds in model_C, then model_A implicitly grounds in model_C through transitivity.

Property 4 - Uniqueness: At most one grounding relationship exists between any ordered pair (source, target). Multiple concept mappings are included within a single grounding.

Closure Calculation Method

Referring to Figure 3, the closure property validation method determines the completeness of a canonical domain model by calculating the percentage of concept references that are resolved either internally or through explicit grounding relationships.

Algorithm Specification

```
FUNCTION calculate_closure(model: CanonicalDomainModel, grounding_map: GroundingMap)
  RETURNS (closure_percentage: Float, ungrounded_refs: Set<ConceptReference>)

INPUT:
  model: The canonical domain model to validate
  grounding_map: The complete grounding relationship graph

OUTPUT:
  closure_percentage: Float in range [0.0, 100.0]
  ungrounded_refs: Set of concept references that are not grounded

ALGORITHM:

  // Step 1: Collect all concept references
  all_references := Set<ConceptReference>()

  FOR EACH concept IN model.concepts:
    FOR EACH relationship IN concept.relationships:
      target := relationship.target_concept
      all_references.ADD(target)

    FOR EACH property IN concept.properties:
      IF property.type IS reference_type:
        FOR EACH ref IN property.referenced_concepts:
          all_references.ADD(ref)

  // Step 2: Classify references as internal or external
  internal_refs := Set<ConceptReference>()
```

```
external_refs := Set<ConceptReference>()

FOR EACH ref IN all_references:
  IF ref.model_id == model.canonical_model_id:
    internal_refs.ADD(ref)
  ELSE:
    external_refs.ADD(ref)

// Step 3: Check external references for grounding
grounded_external := Set<ConceptReference>()
ungrounded_external := Set<ConceptReference>()

FOR EACH ext_ref IN external_refs:
  ref_model_id := ext_ref.model_id
  ref_concept_id := ext_ref.concept_id

  grounding_found := FALSE

  // Find grounding to target model
  FOR EACH grounding IN grounding_map.groundings:
    IF grounding.source == model.canonical_model_id AND
       (grounding.target == ref_model_id OR ref_model_id IN grounding.target):

      // Verify specific concept is in mapping
      FOR EACH mapping IN grounding.concept_mappings:
        target_concept := mapping.target_concept

        IF target_concept.model_id == ref_model_id AND
           target_concept.concept_id == ref_concept_id:
          grounding_found := TRUE
          grounded_external.ADD(ext_ref)
          BREAK

      IF grounding_found:
        BREAK

  IF NOT grounding_found:
    ungrounded_external.ADD(ext_ref)

// Step 4: Calculate closure percentage
total_refs := internal_refs.COUNT() + external_refs.COUNT()
resolved_refs := internal_refs.COUNT() + grounded_external.COUNT()

IF total_refs == 0:
  closure_percentage := 100.0  // No references means vacuously closed
ELSE:
    closure_percentage := (resolved_refs / total_refs) * 100.0

  // Step 5: Return results
  RETURN (closure_percentage, ungrounded_external)

END FUNCTION
```

Validation Thresholds After calculating closure percentage, the system applies quality thresholds:

```
FUNCTION validate_closure_thresholds(closure_percentage: Float)
  RETURNS validation_status: ValidationStatus

  IF closure_percentage < 80.0:
    RETURN ValidationStatus(
      level: ERROR,
      message: "CRITICAL: Model closure {closure_percentage:.1f}% is below minimum threshol
    )

  ELSE IF closure_percentage < 95.0:
    RETURN ValidationStatus(
      level: WARNING,
      message: "WARNING: Model closure {closure_percentage:.1f}% is below target threshold 
    )

  ELSE:
    RETURN ValidationStatus(
      level: SUCCESS,
      message: "SUCCESS: Model achieves production-ready closure of {closure_percentage:.1f
    )

END FUNCTION
```

System-Wide Closure Calculation For systems with multiple canonical domain models:

```
FUNCTION calculate_system_closure(models: Set<CanonicalDomainModel>, grounding_map: Groundi
  RETURNS system_closure_percentage: Float

  total_resolved := 0
  total_refs := 0

  FOR EACH model IN models:
    (closure_pct, ungrounded) := calculate_closure(model, grounding_map)

    model_internal := count_internal_refs(model)
    model_external := count_external_refs(model)
    model_total := model_internal + model_external
    model_resolved := (closure_pct / 100.0) * model_total

    total_resolved += model_resolved
    total_refs += model_total

  system_closure_percentage := (total_resolved / total_refs) * 100.0

  RETURN system_closure_percentage

END FUNCTION
```

Complexity Analysis **Time Complexity**: - Let n = number of concepts in model - Let m = average references per concept - Let r = number of external references - Let g = number of groundings in grounding map - Let p = average concept mappings per grounding

Step 1 (Collect references): O(n * m) **Step 2** (Classify): O(n * m) **Step 3** (Check grounding): O(r * g * p) **Step 4** (Calculate percentage): O(1)

Total: O(n * m + r * g * p)

For typical systems (n=10-30, m=5-10, r=5-20, g=15-25, p=2-5), this executes in <100ms.

Space Complexity: O(n * m) for storing all references.

Implementation Example Continuing with the system shown in Figure 2:

DDD Model: - Internal concepts: 13 - External references: 0 - Grounded external: 0 - **Closure: 100%**

UX Model: - Internal concepts: 11 - External references: 1 (to model_ddd.BoundedContext) - Grounding: `grounding_ux_ddd_001` includes mapping for BoundedContext - Grounded external: 1 - **Closure: (11 + 1) / (11 + 1) × 100% = 100%**

QE Model: - Internal concepts: 12 - External references: 6 (to DDD: Aggregate, Invariant; to UX: Workflow; to Data-Eng: Dataset; to Agile: AcceptanceCriteria, UserStory) - Groundings: `grounding_qe_ddd_001`, `grounding_qe_ux_001`, `grounding_qe_data_eng_001`, `grounding_qe_agile_001` - All 6 external references mapped in groundings - Grounded external: 6 - **Closure: (12 + 6) / (12 + 6) × 100% = 100%**

System Closure: Average across all 5 models = 100%

LLM-Constrained Generation Method

Referring to Figure 4, the LLM-constrained generation method coordinates four phases to produce software artifacts that conform to multiple canonical domain models with cross-domain consistency guarantees.

Phase 1: Schema Loading **Purpose**: Construct unified schema context containing all canonical domain models required for the task plus transitively grounded dependencies.

Algorithm:

```
FUNCTION load_schema_context(task_description: String, model_repository: Repository, groundi
  RETURNS schema_context: SchemaContext

INPUT:
  task_description: Natural language description of generation task
  model_repository: Repository containing all canonical domain models
  grounding_map: Complete grounding relationship graph

OUTPUT:
  schema_context: Unified context for LLM with all required schemas

ALGORITHM:

  // Step 1.1: Identify required models from task
  required_model_ids := identify_models_from_task(task_description)

  // Heuristic rules for identification:
  // - Contains "aggregate", "bounded context", "domain" → add model_ddd
  // - Contains "workflow", "page", "component", "UI" → add model_ux
  // - Contains "test", "quality", "validation" → add model_qe
  // - Contains "pipeline", "dataset", "schema" → add model_data_eng
  // - Contains "epic", "feature", "story", "vision" → add model_agile
  // - User can explicitly specify: "@model_ddd @model_ux" in task

  // Step 1.2: Load primary models
  loaded_models := Map<ModelID, CanonicalDomainModel>()

  FOR EACH model_id IN required_model_ids:
    model := model_repository.load(model_id)
    loaded_models[model_id] := model

  // Step 1.3: Resolve transitive grounding dependencies
  dependency_queue := Queue(loaded_models.keys())
  processed := Set<ModelID>()
```

```
WHILE dependency_queue IS NOT EMPTY:
  current_model_id := dependency_queue.DEQUEUE()

  IF current_model_id IN processed:
    CONTINUE

  processed.ADD(current_model_id)
  current_model := loaded_models[current_model_id]

  // Find all groundings from current model
  FOR EACH grounding IN grounding_map.groundings:
    IF grounding.source == current_model_id:
      target_model_id := grounding.target

      IF target_model_id NOT IN loaded_models:
        target_model := model_repository.load(target_model_id)
        loaded_models[target_model_id] := target_model
        dependency_queue.ENQUEUE(target_model_id)

// Step 1.4: Build unified schema context
schema_context := SchemaContext()
schema_context.models := loaded_models

// Merge vocabularies with qualified names
unified_vocabulary := Map<QualifiedName, ConceptDefinition>()

FOR EACH (model_id, model) IN loaded_models:
  FOR EACH concept IN model.concepts:
    qualified_name := f"{model_id}.{concept.concept_id}"
    unified_vocabulary[qualified_name] := concept

schema_context.vocabulary := unified_vocabulary

// Merge patterns
unified_patterns := Set<Pattern>()

FOR EACH (model_id, model) IN loaded_models:
  FOR EACH pattern IN model.patterns:
    pattern.qualified_id := f"{model_id}.{pattern.pattern_id}"
    unified_patterns.ADD(pattern)

schema_context.patterns := unified_patterns

// Merge constraints
unified_constraints := Set<Constraint>()
```

```
  FOR EACH (model_id, model) IN loaded_models:
    FOR EACH constraint IN model.constraints:
      constraint.qualified_id := f"{model_id}.{constraint.constraint_id}"
      constraint.source_model := model_id
      unified_constraints.ADD(constraint)

  schema_context.constraints := unified_constraints

  // Build grounding subgraph
  relevant_groundings := Set<GroundingRelationship>()

  FOR EACH grounding IN grounding_map.groundings:
    IF grounding.source IN loaded_models AND grounding.target IN loaded_models:
      relevant_groundings.ADD(grounding)

  schema_context.groundings := relevant_groundings

  // Step 1.5: Validate schema context
  validation_result := validate_schema_context(schema_context)

  IF NOT validation_result.passed:
    RAISE ValidationError(validation_result.message)

  RETURN schema_context

END FUNCTION
```

Validation of Schema Context:

```
FUNCTION validate_schema_context(schema_context: SchemaContext)
  RETURNS validation_result: ValidationResult

  errors := []

  // Check for cycles in grounding graph
  IF has_cycles(schema_context.groundings):
    errors.ADD("Circular dependency detected in grounding relationships")

  // Check for contradictory constraints
  FOR EACH constraint1 IN schema_context.constraints:
    FOR EACH constraint2 IN schema_context.constraints:
      IF constraint1 != constraint2:
        IF are_contradictory(constraint1.rule, constraint2.rule):
          errors.ADD(f"Contradictory constraints: {constraint1.qualified_id} and {constrain

  // Check for unresolved concept references in patterns
  FOR EACH pattern IN schema_context.patterns:
    FOR EACH participant IN pattern.participants:
      qualified_name := f"{participant.model_id}.{participant.concept_id}"
      IF qualified_name NOT IN schema_context.vocabulary:
        errors.ADD(f"Pattern {pattern.qualified_id} references undefined concept {qualified_

  IF errors.COUNT() > 0:
    RETURN ValidationResult(passed=FALSE, errors=errors)
  ELSE:
    RETURN ValidationResult(passed=TRUE)

END FUNCTION
```

Phase 2: Constrained Generation **Purpose**: Generate software artifact using LLM with real-time constraint validation, pruning invalid candidates.

Algorithm:

```
FUNCTION constrained_generate(task: String, schema_context: SchemaContext, llm_interface: LI
  RETURNS artifact: Artifact

INPUT:
  task: Task description
  schema_context: Unified schema context from Phase 1
  llm_interface: Interface to LLM API

OUTPUT:
  artifact: Generated software artifact conforming to all schemas

PARAMETERS:
  beam_width k := 5  // Number of candidate continuations per step
  max_iterations := 100  // Maximum generation steps
  temperature := 0.3  // Lower temperature for more deterministic generation

ALGORITHM:

  // Step 2.1: Construct prompt
  prompt := construct_prompt(task, schema_context)

  // Prompt structure:
  // 1. Task description
  // 2. Schema overview (1000-2000 tokens)
  // 3. Relevant concepts (5000-10000 tokens)
  // 4. Applicable patterns (2000-5000 tokens)
  // 5. Constraints (2000-5000 tokens)
  // 6. Examples from schemas (5000-10000 tokens)
  // 7. Validation checklist
  // Total: 20,000-50,000 tokens typical

  artifact := initialize_artifact(task)
  iteration := 0

  // Step 2.2: Iterative generation with beam search
  WHILE NOT artifact.is_complete() AND iteration < max_iterations:
    iteration += 1

    // Generate k candidate continuations
    candidates := llm_interface.generate_k_candidates(
      prompt=prompt + artifact.current_content,
      k=beam_width,
      temperature=temperature
    )

    // Validate each candidate
    valid_candidates := []

    FOR EACH candidate IN candidates:
      temp_artifact := artifact.clone()
      temp_artifact.append(candidate.continuation)

      validation_result := validate_artifact_incremental(temp_artifact, schema_context)

      IF validation_result.is_valid:
        valid_candidates.APPEND({
          candidate: candidate,
          artifact: temp_artifact,
          probability: candidate.log_probability,
          validation_score: validation_result.score
        })

    // Step 2.3: Handle no valid candidates
    IF valid_candidates.COUNT() == 0:
      // Relaxation strategy: Soften constraints or backtrack
      IF can_relax_constraints(schema_context):
        schema_context := relax_constraints(schema_context)
        CONTINUE  // Retry with relaxed constraints
      ELSE:
        // Backtracking: Remove last addition and try alternative
        IF artifact.can_backtrack():
          artifact := artifact.backtrack()
          CONTINUE
        ELSE:
          RAISE GenerationError("No valid continuations found and cannot backtrack")
    // Step 2.4: Select best valid candidate
    best_candidate := SELECT_MAX(valid_candidates,
                                key=lambda c: c.probability + 0.5 * c.validation_score)

    artifact := best_candidate.artifact

    // Check completion
    IF artifact.appears_complete():
      // Perform full validation
      final_validation := validate_artifact_full(artifact, schema_context)
      IF final_validation.is_valid:
        artifact.mark_complete()
        BREAK

  // Step 2.5: Handle timeout
  IF iteration >= max_iterations:
    RAISE GenerationError("Maximum iterations reached without completion")

  RETURN artifact

END FUNCTION
```

Incremental Validation:

```
FUNCTION validate_artifact_incremental(artifact: Artifact, schema_context: SchemaContext)
  RETURNS validation_result: ValidationResult

  // Incremental validation checks only new content since last validation
  // to minimize computation during beam search

  new_content := artifact.get_new_content()
  errors := []
  warnings := []
  score := 1.0

  // Syntactic check
  IF NOT can_parse(new_content):
    RETURN ValidationResult(is_valid=FALSE, score=0.0, errors=["Syntax error in new content'

  // Quick constraint check on new elements only
  FOR EACH constraint IN schema_context.constraints:
    IF constraint.severity == "error":
      IF NOT constraint.validator(new_content):
        errors.APPEND(constraint.message)
        score -= 0.3
  // Reference resolution check for new references
  new_references := extract_references(new_content)
  FOR EACH ref IN new_references:
    IF NOT can_resolve(ref, schema_context):
      errors.APPEND(f"Cannot resolve reference: {ref}")
      score -= 0.2

  is_valid := (errors.COUNT() == 0)

  RETURN ValidationResult(is_valid=is_valid, score=MAX(0.0, score), errors=errors, warnings=

END FUNCTION
```

Phase 3: Validation **Purpose**: Perform comprehensive multi-level validation of complete artifact.

Algorithm:

```
FUNCTION validate_artifact_full(artifact: Artifact, schema_context: SchemaContext)
  RETURNS validation_result: DetailedValidationResult

  errors := []
  warnings := []
  info_messages := []

  // Level 1: Syntactic Validation
  syntax_result := validate_syntax(artifact)
  IF NOT syntax_result.passed:
    errors.EXTEND(syntax_result.errors)
    RETURN DetailedValidationResult(is_valid=FALSE, errors=errors)

  // Level 2: Semantic Validation

  // 2.1: Resolve all references
  all_references := extract_all_references(artifact)
  unresolved := []

  FOR EACH ref IN all_references:
    qualified_name := ref.qualified_name  // Format: "model_id.concept_id"

    IF qualified_name NOT IN schema_context.vocabulary:
      errors.APPEND(f"Unresolved reference: {qualified_name} at {ref.location}")
      unresolved.APPEND(ref)

  // 2.2: Check constraints
  FOR EACH constraint IN schema_context.constraints:
```

```
  constraint_result := constraint.validator(artifact)

  IF NOT constraint_result:
    message := format_constraint_message(constraint, artifact)

    IF constraint.severity == "error":
      errors.APPEND(message)
    ELIF constraint.severity == "warning":
      warnings.APPEND(message)
    ELSE:  // info
      info_messages.APPEND(message)

// 2.3: Validate patterns
FOR EACH pattern_instance IN artifact.pattern_instances:
  pattern := schema_context.patterns.get(pattern_instance.pattern_id)

  IF pattern IS NULL:
    errors.APPEND(f"Reference to undefined pattern: {pattern_instance.pattern_id}")
    CONTINUE

  // Check pattern structure
  IF NOT matches_pattern_structure(pattern_instance, pattern):
    errors.APPEND(f"Pattern instance {pattern_instance.id} does not match structure of {pa

  // Check participants
  FOR EACH participant IN pattern.participants:
    IF NOT has_participant(pattern_instance, participant):
      errors.APPEND(f"Pattern instance {pattern_instance.id} missing required participant

// Level 3: Cross-Domain Consistency Validation

// 3.1: For each external reference, verify grounding
external_refs := filter(all_references, lambda r: r.model_id != artifact.primary_model_id)
ungrounded := []

FOR EACH ext_ref IN external_refs:
  grounding := find_grounding(artifact.primary_model_id, ext_ref.model_id, schema_context.

  IF grounding IS NULL:
    errors.APPEND(f"No grounding relationship from {artifact.primary_model_id} to {ext_ref
    ungrounded.APPEND(ext_ref)
    CONTINUE

  // 3.2: Verify concept mapping exists
  concept_mapping := find_concept_mapping(ext_ref, grounding.concept_mappings)
```

```
    IF concept_mapping IS NULL:
      errors.APPEND(f"Grounding {grounding.grounding_id} does not include mapping for concep
      CONTINUE

    // 3.3: Verify cardinality
    actual_cardinality := count_references(artifact, ext_ref)
    IF NOT satisfies_cardinality(actual_cardinality, concept_mapping.cardinality):
      errors.APPEND(f"Cardinality violation: {ext_ref.qualified_name} referenced {actual_car

    // 3.4: Run grounding-specific validation rules
    FOR EACH validation_rule IN grounding.validation_rules:
      rule_result := validation_rule.validator(artifact, schema_context)
      IF NOT rule_result.passed:
        IF grounding.strength == "strong":
          errors.APPEND(rule_result.message)
        ELIF grounding.strength == "weak":
          warnings.APPEND(rule_result.message)
        ELSE:  // optional
          info_messages.APPEND(rule_result.message)

  // Level 4: Semantic Alignment (for semantic groundings)
  semantic_groundings := filter(schema_context.groundings, lambda g: g.type == "semantic")

  FOR EACH grounding IN semantic_groundings:
    IF grounding.source == artifact.primary_model_id:
      alignment_score := calculate_semantic_alignment(artifact, grounding, schema_context)

      IF alignment_score < 0.70:  // 70% threshold
        warnings.APPEND(f"Semantic alignment with {grounding.target} is {alignment_score:.1%

  // Final result
  is_valid := (errors.COUNT() == 0)

  RETURN DetailedValidationResult(
    is_valid=is_valid,
    errors=errors,
    warnings=warnings,
    info_messages=info_messages,
    unresolved_references=unresolved,
    ungrounded_references=ungrounded
  )

END FUNCTION
```

Phase 4: Explanation Generation **Purpose**: Generate human-readable justification for design decisions with schema citations.

Algorithm:

```
FUNCTION generate_explanation(artifact: Artifact, schema_context: SchemaContext, generation_
  RETURNS explanation: Explanation

  justifications := []

  // For each major design decision
  FOR EACH decision IN artifact.design_decisions:

    // Find schema elements that influenced decision
    relevant_concepts := find_concepts_used(decision, schema_context)
    relevant_patterns := find_patterns_applied(decision, schema_context)
    relevant_constraints := find_constraints_satisfied(decision, schema_context)

    // Build justification
    justification := Justification()
    justification.decision_id := decision.id
    justification.what := decision.description

    // Why: Schema-based rationale
    justification.why := generate_rationale(decision, relevant_concepts, relevant_patterns)

    // Source: Schema citations
    justification.sources := []
    FOR EACH concept IN relevant_concepts:
      justification.sources.APPEND(f"{concept.qualified_id}: {concept.description}")
    FOR EACH pattern IN relevant_patterns:
      justification.sources.APPEND(f"{pattern.qualified_id}: {pattern.intent}")

    // Alternatives: Rejected options
    alternatives := generation_log.get_rejected_alternatives(decision)
    justification.alternatives := []
    FOR EACH alt IN alternatives:
      justification.alternatives.APPEND({
        description: alt.description,
        rejection_reason: alt.rejection_reason,
        validation_failures: alt.validation_failures
      })

    // Constraints satisfied
    justification.constraints_satisfied := []
    FOR EACH constraint IN relevant_constraints:
      justification.constraints_satisfied.APPEND(f"{constraint.qualified_id}: {constraint.de
    justifications.APPEND(justification)

  // Build hierarchical explanation
  explanation := Explanation()
  explanation.summary := generate_summary(artifact, schema_context)
  explanation.justifications := justifications
  explanation.schema_context_summary := summarize_schemas_used(schema_context)
  explanation.validation_evidence := "All constraints satisfied. No unresolved references. (

  // Add traceability
  explanation.traceability := {
    task: artifact.original_task,
    schemas_used: schema_context.models.keys(),
    groundings_used: schema_context.groundings.map(lambda g: g.grounding_id),
    validation_status: "PASSED",
    generation_timestamp: current_timestamp()
  }

  RETURN explanation

END FUNCTION
```

Explanation Output Format:

```
explanation:
  summary: |
    Generated UX workflow for checkout process grounded in DDD OrderManagement context.
    Workflow implements DDD.PlaceOrder domain service with 5 steps validating order invarian

  justifications:
    - decision_id: workflow_step_001
      what: "Workflow begins by loading Order aggregate"
      why: "DDD.Aggregate pattern requires all operations on aggregate start from root entit
      sources:
        - "model_ddd.Aggregate: Cluster of entities with transactional consistency boundary"
        - "model_ddd.Repository: Mediate between domain and data mapping using collection in
      constraints_satisfied:
        - "model_ddd.AGGREGATE_HAS_ROOT: Verified Order is root entity"
      alternatives:
        - description: "Load individual OrderLine entities directly"
          rejection_reason: "Violates aggregate pattern - entities must be accessed through
          validation_failures: ["AGGREGATE_HAS_ROOT constraint failed"]

    - decision_id: workflow_step_002
      what: "Validate Order.totalAmount matches sum of OrderLine.subtotal"
      why: "DDD.Invariant from Order aggregate requires total = sum(lines.subtotal). Must be
      sources:
        - "model_ddd.Invariant: Business rule that must always be true within aggregate bour
      constraints_satisfied:
        - "model_ddd.INVARIANT_ENFORCED: Order aggregate enforces total calculation invarian

  schema_context_summary:
    models_used:
      - model_ux (User Experience - v1.0.0)
      - model_ddd (Domain-Driven Design - v1.0.0)
    groundings_used:
      - grounding_ux_ddd_001 (structural, strong)
    concepts_referenced: 7
    patterns_applied: 3
    constraints_validated: 5

  validation_evidence: |
     Syntactic validation passed
     Semantic validation passed: All 7 concept references resolved
     Cross-domain consistency verified: UX.Workflow properly grounds in DDD.DomainService
     All 5 hard constraints satisfied
     No ungrounded external references

  traceability:
    task: "Design checkout workflow for e-commerce platform"
    schemas_used: ["model_ux", "model_ddd"]
    groundings_used: ["grounding_ux_ddd_001"]
    validation_status: "PASSED"
    generation_timestamp: "2024-10-14T10:30:00Z"
```

Ripple Effect Management

Referring to Figure 5, ripple effect management coordinates updates across multiple canonical domain models when a change occurs, maintaining system-wide consistency.

Algorithm Specification

```
FUNCTION manage_ripple_effect(
  change: ModelChange,
  all_models: Map<ModelID, CanonicalDomainModel>,
  grounding_map: GroundingMap,
  llm_interface: LLMInterface,
  user_interface: UserInterface
) RETURNS update_result: UpdateResult

INPUT:
  change: Specification of change to a canonical domain model
  all_models: All canonical domain models in system
  grounding_map: Complete grounding relationship graph
  llm_interface: Interface to LLM for regeneration
  user_interface: Interface for user approval

OUTPUT:
  update_result: Result of ripple effect propagation

ALGORITHM:

  changed_model_id := change.model_id
  changed_concept_id := change.concept_id
  change_type := change.type  // Added, Modified, Removed

  // Step 1: Build reverse grounding graph (incoming edges)
  reverse_graph := build_reverse_graph(grounding_map)
  // reverse_graph[model_id] = Set of models that ground IN model_id

  // Step 2: Identify directly affected models
  directly_affected := Set<ModelID>()

  FOR EACH grounding IN reverse_graph[changed_model_id]:
    source_model_id := grounding.source

    // Check if this grounding references the changed concept
    FOR EACH mapping IN grounding.concept_mappings:
      IF mapping.target_concept.model_id == changed_model_id AND
         mapping.target_concept.concept_id == changed_concept_id:
        directly_affected.ADD(source_model_id)
        BREAK

  // Step 3: Transitively identify affected models (propagate through grounding graph)
  all_affected := directly_affected.COPY()
  to_process := Queue(directly_affected)
  processed := Set<ModelID>()

  WHILE to_process IS NOT EMPTY:
    current_model_id := to_process.DEQUEUE()

    IF current_model_id IN processed:
      CONTINUE
    processed.ADD(current_model_id)

    // Find models grounding in current
    FOR EACH grounding IN reverse_graph[current_model_id]:
```

```
    transitively_affected := grounding.source
    IF transitively_affected NOT IN all_affected:
      all_affected.ADD(transitively_affected)
      to_process.ENQUEUE(transitively_affected)

// Step 4: Validate affected models with proposed change
impact_report := ImpactReport()
impact_report.changed_model := changed_model_id
impact_report.changed_concept := changed_concept_id
impact_report.change_description := change.description
impact_report.directly_affected := directly_affected
impact_report.transitively_affected := all_affected - directly_affected
impact_report.validation_failures := []

// Apply change tentatively
modified_changed_model := all_models[changed_model_id].clone()
apply_change(modified_changed_model, change)

// Validate each affected model
FOR EACH affected_model_id IN all_affected:
  affected_model := all_models[affected_model_id]

  // Run validation with modified changed model
  validation_result := validate_model_with_external_change(
    affected_model,
    changed_model_id,
    modified_changed_model,
    grounding_map
  )

  IF NOT validation_result.passed:
    impact_report.validation_failures.APPEND({
      model: affected_model_id,
      errors: validation_result.errors,
      warnings: validation_result.warnings
    })

// Step 5: Present impact report to user
user_interface.display_impact_report(impact_report)

approval := user_interface.request_approval(
  message="Proceed with ripple effect propagation to {all_affected.COUNT()} models?",
  options=["Approve", "Cancel", "Approve with Review"]
)

IF approval == "Cancel":
    RETURN UpdateResult(status="cancelled", message="User cancelled ripple effect")

  // Step 6: Regenerate affected artifacts
  updated_models := Map<ModelID, CanonicalDomainModel>()
  updated_models[changed_model_id] := modified_changed_model

  FOR EACH affected_model_id IN all_affected:
    affected_model := all_models[affected_model_id]

    // Find validation failures for this model
    failures := find_failures_for_model(impact_report.validation_failures, affected_model_ic

    IF failures.COUNT() == 0:
      // Model still valid, no regeneration needed
      CONTINUE

    // Build corrective prompt
    corrective_prompt := build_corrective_prompt(
      affected_model,
      failures,
      change,
      modified_changed_model
    )

    // Load schema context including updated changed model
    temp_models := all_models.COPY()
    temp_models[changed_model_id] := modified_changed_model

    schema_context := load_schema_context_for_model(
      affected_model_id,
      temp_models,
      grounding_map
    )

    // Regenerate affected artifacts using LLM
    updated_artifact := constrained_generate(corrective_prompt, schema_context, llm_interfac

    // Validate regenerated artifact
    final_validation := validate_artifact_full(updated_artifact, schema_context)

    IF NOT final_validation.is_valid:
      // Escalate: Cannot automatically resolve
      error_message := f"Cannot automatically resolve ripple effect for {affected_model_id}.

      IF approval == "Approve with Review":
        // User will manually fix
      user_interface.show_error(error_message)
      updated_models[affected_model_id] := affected_model  // Keep original, user will fi
    ELSE:
      RAISE RippleEffectError(error_message)
  ELSE:
    // Success: Update model
    updated_model := apply_artifact_to_model(affected_model, updated_artifact)
    updated_models[affected_model_id] := updated_model

// Step 7: Version bump all updated models
FOR EACH (model_id, model) IN updated_models:
  IF model_id == changed_model_id:
    // Bump based on change type
    IF change_type == "Removed" OR change.breaking == TRUE:
      bump_major_version(model)
    ELIF change_type == "Added":
      bump_minor_version(model)
    ELSE:  // Modified
      bump_patch_version(model)
  ELSE:
    // Affected models get minor bump (backward-compatible adjustment)
    bump_minor_version(model)

// Step 8: Calculate new system closure
new_system_closure := calculate_system_closure(updated_models.values(), grounding_map)

IF new_system_closure < 95.0:
  warning_message := f"WARNING: System closure degraded to {new_system_closure:.1f}% afte
  user_interface.show_warning(warning_message)

// Step 9: Commit updates
FOR EACH (model_id, model) IN updated_models:
  all_models[model_id] := model
  persist_model(model)  // Save to repository

// Step 10: Generate audit trail
audit_trail := AuditTrail()
audit_trail.timestamp := current_timestamp()
audit_trail.change := change
audit_trail.affected_models := all_affected
audit_trail.updated_models := updated_models.keys()
audit_trail.approval := approval
audit_trail.new_system_closure := new_system_closure

persist_audit_trail(audit_trail)
```

```
  RETURN UpdateResult(
    status="success",
    updated_models=updated_models.keys(),
    new_system_closure=new_system_closure,
    audit_trail=audit_trail
  )

END FUNCTION
```

Example Ripple Effect Scenario **Scenario**: Add new concept "SubscriptionPolicy" to DDD model

Step 1: Changed model = model_ddd

Step 2: Identify affected models - UX model has grounding_ux_ddd_001 (structural) → UX potentially affected - QE model has grounding_qe_ddd_001 (procedural) → QE potentially affected - Agile model has grounding_agile_ddd_001 (epistemic) → Agile potentially affected

Step 3: Check if specific concept is referenced - UX: No existing UX workflows reference "SubscriptionPolicy" → Not affected - QE: No existing tests reference "SubscriptionPolicy" → Not affected - Agile: Feature "Billing" references DDD concepts generally → Validate if new concept relevant

Step 4: Validation - Agile.Feature("Billing") description mentions "subscription" → Should reference new SubscriptionPolicy concept - Validation fails: Missing reference to new concept

Step 5: User approval granted

Step 6: Regenerate - LLM prompt: "Update Agile Feature 'Billing' to reference new DDD concept SubscriptionPolicy for subscription management" - LLM generates updated feature description including reference to model_ddd.SubscriptionPolicy - Validation passes

Step 7: Version bumps - model_ddd: 1.0.0 → 1.1.0 (minor, added concept) - model_agile: 2.0.0 → 2.1.0 (minor, updated feature description)

Step 8: System closure: Still 100% (new reference grounded through existing grounding_agile_ddd_001)

Step 9: Commit to repository

Step 10: Audit trail recorded

Greenfield Development Workflow System

Referring to Figure 6, the greenfield development workflow system orchestrates nine phases of systematic development from product vision through implementation with formal validation at each step. This workflow represents a novel end-to-end process for LLM-assisted software development with multi-domain coordination.

Phase 1: Vision Definition and Validation **Input**: Initial product concept from stakeholders (unstructured text)

Canonical Models Used: Agile canonical model

Process:

1. **Vision Document Creation**:
    - Stakeholders draft product vision document describing problem, target users, solution approach, and expected outcomes
    - Document may be incomplete or inconsistent initially
2. **LLM-Assisted Completeness Check**:
    - System loads Agile canonical model schema
    - Agile.Vision concept defines required elements:
        - `problem_statement` (String, 1..1, required)
        - `target_users` (Set, 1..*, required)
        - `value_proposition` (String, 1..1, required)
        - `success_metrics` (Set, 1..*, required)
        - `constraints` (Set, 0..*, optional)
        - `assumptions` (Set, 0..*, optional)
        - `high_level_scope` (String, 1..1, required)
    - System constructs prompt: "Given the Agile canonical model Vision concept definition, analyze this vision document for completeness. Identify missing required elements and suggest additions."
    - LLM generates completeness report with specific gaps identified
3. **Validation Report Example**:

```yaml
completeness_check:
  status: INCOMPLETE
  missing_required:
    - success_metrics: "No quantitative success metrics defined"
  missing_optional:
    - constraints: "Budget and timeline constraints not specified"
  suggestions:
    - "Add metrics: Monthly Recurring Revenue target, Customer Churn rate target, User Satis
    - "Specify budget constraint: e.g., Development budget $500K"
    - "Specify timeline: e.g., MVP in 6 months"
```

4. **Human Review and Iteration**:
    - Product owner reviews LLM suggestions
    - Updates vision document
    - Re-validates until completeness check passes
5. **Output Artifact**:

```yaml
# vision.yaml
```

```yaml
vision:
  vision_id: ECOM_PLATFORM_V1
  problem_statement: "Small/medium businesses lack affordable e-commerce platforms with inte
  target_users:
    - persona_id: SmallBusinessOwner
      description: "Business with 1-10 employees, <$1M annual revenue"
    - persona_id: OnlineShopkeeper
      description: "Individual selling products online, managing inventory manually"
  value_proposition: "Unified commerce and inventory management platform under $100/month"
  success_metrics:
    - metric_id: MRR
      name: "Monthly Recurring Revenue"
      target: "$500K within 18 months"
      measurement: "Subscription revenue per month"
    - metric_id: CustomerChurn
      name: "Customer Churn Rate"
      target: "<5% monthly"
      measurement: "Percentage of customers canceling per month"
    - metric_id: UserSatisfaction
      name: "User Satisfaction Score"
      target: ">4.5/5.0"
      measurement: "Average rating from user surveys"
  constraints:
    - constraint_id: BUDGET
      type: budget
      description: "Development budget limited to $500K"
    - constraint_id: TIMELINE
      type: timeline
      description: "MVP must launch within 6 months"
  assumptions:
    - assumption_id: MIGRATION_WILLINGNESS
      description: "SMBs willing to migrate from spreadsheets to dedicated platform"
      risk: medium
    - assumption_id: PAYMENT_INTEGRATION
      description: "Stripe integration sufficient for payment processing"
      risk: low
  high_level_scope: "Core commerce (product catalog, cart, checkout), inventory tracking, o
  validation_status: COMPLETE
```

Phase 2: Strategic Domain Model Definition **Input**: Validated vision.yaml, optional current state documentation

Canonical Models Used: DDD canonical model

Process:

1. **Bounded Context Identification**:

  - System loads DDD canonical model
  - Constructs prompt: "Based on this product vision and DDD canonical model (specifically BoundedContext pattern), identify bounded contexts for this system. For each context, define: responsibility, core concepts, and ubiquitous language."
  - LLM analyzes vision and proposes bounded contexts

2. **LLM Output Example**:

```
proposed_bounded_contexts:
  - context_id: CATALOG
    name: "Product Catalog"
    responsibility: "Manage product information, categories, attributes, pricing, and search
    rationale: "Product data has its own lifecycle and consistency requirements distinct fro
    core_concepts:
      - Product
      - Category
      - ProductAttribute
      - PriceList
      - SearchIndex
    ubiquitous_language:
      product: "Sellable item with SKU, description, and attributes"
      category: "Hierarchical grouping of products for navigation"
      price_list: "Set of prices for products, may vary by customer segment or time period"

  - context_id: INVENTORY
    name: "Inventory Management"
    responsibility: "Track stock levels, manage replenishment, record stock movements"
    rationale: "Inventory has different consistency requirements (near real-time) and differ
    core_concepts:
      - StockItem
      - Warehouse
      - StockMovement
      - ReorderPolicy
    ubiquitous_language:
      stock_item: "Physical quantity of a product at a specific warehouse"
      stock_movement: "Record of inventory change (received, sold, transferred, adjusted)"

  - context_id: ORDER
    name: "Order Management"
    responsibility: "Process customer orders, manage order lifecycle, coordinate fulfillment
    rationale: "Orders are central business transactions requiring strong consistency and au
    core_concepts:
      - Order
      - OrderLine
      - Customer
      - OrderStatus
```

```yaml
   - Payment
  ubiquitous_language:
   order: "Customer purchase request with line items and fulfillment status"
   order_line: "Individual product and quantity within an order"
```

3. **Domain Expert Review**:
   - Domain experts evaluate proposed contexts
   - Check for: Single responsibility, clear boundaries, appropriate granularity
   - Feedback example: "Split ORDER into ORDER and FULFILLMENT contexts - different responsibilities and teams"
4. **Context Map Generation**:
   - LLM generates relationships between contexts using DDD context map patterns
   - Patterns: Shared Kernel, Customer-Supplier, Conformist, Anti-Corruption Layer
5. **Ubiquitous Language Validation**:
   - LLM checks for term conflicts across contexts
   - Example: "Product" in CATALOG vs. "StockItem" in INVENTORY - different perspectives of same thing
   - Documents translation: CATALOG.Product INVENTORY.StockItem (via ProductSKU)
6. **Validation**:
   - DDD closure check: All concept references resolve within context or explicit cross-context references
   - Grounding check: Vision.Epic → DDD.BoundedContext mappings defined
   - Acyclicity: No circular dependencies between contexts
7. **Output Artifact**:

```yaml
# strategic-ddd-model.yaml
canonical_model_id: project_ddd_model_v1
based_on: model_ddd  # Grounded in DDD canonical model
version: 1.0.0

bounded_contexts:
 - context_id: CATALOG
  name: "Product Catalog"
  responsibility: "Manage product information, categories, search"
  aggregates:
   - aggregate_id: Product
    root_entity: Product
    entities:
     - ProductAttribute
    value_objects:
     - SKU
     - Price
```

```
        invariants:
          - "Product must have unique SKU"
          - "Price must be positive"
      - aggregate_id: Category
        root_entity: Category
        value_objects:
          - CategoryPath
        invariants:
          - "Category path must not have cycles"

  - context_id: INVENTORY
    # ... (similar detail)

  - context_id: ORDER
    # ... (similar detail)

context_map:
  relationships:
    - source: ORDER
      target: CATALOG
      relationship_type: Customer-Supplier
      integration: "ORDER consumes product information from CATALOG via Product SKU lookup"

    - source: ORDER
      target: INVENTORY
      relationship_type: Customer-Supplier
      integration: "ORDER requests stock allocation from INVENTORY, INVENTORY confirms avai

    - source: FULFILLMENT
      target: INVENTORY
      relationship_type: Customer-Supplier
      integration: "FULFILLMENT triggers stock movement when items shipped"

validation:
  closure: 100%
  acyclicity: PASSED
  grounding_to_vision: COMPLETE
```

Phase 3: Vision Decomposition to Epics and Features **Input**: vision.yaml, strategic-ddd-model.yaml

Canonical Models Used: Agile canonical model, DDD canonical model (via grounding)

Process:

1. **Epic Extraction**:

   - Prompt: “Decompose this vision into epics. Each epic must reference at least one bounded context from the strategic DDD model. Use Agile canonical model Epic structure.”
   - LLM generates epics aligned with bounded contexts
2. **Grounding Validation**:
   - For each epic, verify grounding: Agile.Epic → DDD.BoundedContext
   - Check grounding relationship `grounding_agile_ddd` exists and includes Epic → BoundedContext mapping
   - Validation rule: `epic.bounded_context_refs` must contain valid context IDs from strategic-ddd-model.yaml
3. **Feature Definition**:
   - For each epic, LLM generates features
   - Each feature includes: ID, name, description, user value, acceptance criteria, dependencies
   - Grounding: Agile.Feature → DDD.Aggregate (weak), Agile.Feature → UX.Workflow (strong, deferred to Phase 4)
4. **Cross-Model Validation**:
   - Verify all epic → bounded context references valid
   - Verify no orphaned features (every feature belongs to epic)
   - Validate feature dependencies align with context map relationships
5. **Product Owner Review**:
   - Evaluate epic/feature decomposition for completeness
   - Adjust priorities
   - Approve or request regeneration
6. **Output Artifact**:

```yaml
# roadmap.yaml
canonical_model_id: project_roadmap_v1
based_on: model_agile
version: 1.0.0

epics:
  - epic_id: CATALOG_MANAGEMENT
    name: "Product Catalog Management"
    description: "Enable business owners to create, update, and organize product catalog wit
    bounded_context_refs:
      - CATALOG  # Grounding validation: CATALOG exists in strategic-ddd-model.yaml
    business_value: "Core capability - cannot sell products without catalog"
    priority: P0
    estimated_effort: 8 weeks

  - epic_id: INVENTORY_TRACKING
    name: "Inventory Tracking and Replenishment"
    description: "Real-time inventory tracking with automated reorder alerts"
    bounded_context_refs:
      - INVENTORY
    business_value: "Prevents stockouts and overselling"
    priority: P0
    estimated_effort: 6 weeks

  - epic_id: ORDER_PROCESSING
    name: "Order Processing"
    description: "Complete order lifecycle from cart through payment and fulfillment"
    bounded_context_refs:
      - ORDER
      - FULFILLMENT
    business_value: "Core revenue-generating capability"
    priority: P0
    estimated_effort: 10 weeks

features:
  - feature_id: CATALOG_001
    epic_id: CATALOG_MANAGEMENT
    name: "Product Creation and Editing"
    description: "Business owner can create products with name, SKU, description, price, ima
    user_story_count: 5
    aggregate_refs:
      - CATALOG.Product  # Weak grounding to DDD aggregate
    acceptance_criteria:
      - "Product created with all required attributes"
      - "Product SKU must be unique"
      - "Price must be positive"
    dependencies: []

  - feature_id: CATALOG_002
    epic_id: CATALOG_MANAGEMENT
    name: "Category Management"
    description: "Organize products into hierarchical categories"
    aggregate_refs:
      - CATALOG.Category
    dependencies:
      - CATALOG_001  # Must create products before categorizing

  - feature_id: INVENTORY_001
    epic_id: INVENTORY_TRACKING
    name: "Stock Level Tracking"
    description: "Track quantity on hand for each product at each warehouse"
    aggregate_refs:
      - INVENTORY.StockItem
    dependencies:
      - CATALOG_001  # Needs products to exist
```

```yaml
validation:
  all_epics_grounded: true
  all_features_have_epic: true
  no_circular_dependencies: true
```

Phase 4: User Story Decomposition **Input**: roadmap.yaml (features prioritized for current sprint), strategic-ddd-model.yaml

Canonical Models Used: Agile, DDD, UX (via grounding)

Process:

1. **User Story Generation**:
   - For each feature in sprint backlog, generate user stories
   - Prompt includes: Agile.UserStory schema, DDD bounded context details, Feature description
   - Format: "As [Persona], I want [Goal] so that [Benefit]"
2. **Acceptance Criteria Definition**:
   - LLM generates testable acceptance criteria for each story
   - Grounding: Agile.AcceptanceCriteria → DDD.Invariant (must validate business rules)
   - Ensures each criterion is testable and aligned with domain invariants
3. **Workflow Mapping**:
   - LLM proposes UX workflow outline for each story
   - Deferred detailed design to Phase 6, but establishes references
4. **Technical Task Breakdown**:
   - LLM generates tasks: Domain model implementation, persistence, API, UI, testing
   - Grounding: Tasks reference DDD aggregates, future UX components, QE test cases
5. **Example Output**:

```yaml
# sprint-backlog.yaml
sprint: Sprint-001
duration: 2 weeks

user_stories:
  - story_id: CATALOG_001_US001
    feature_id: CATALOG_001
    title: "Create New Product"
    as_a: SmallBusinessOwner
    i_want: "to create a new product with name, SKU, price, and description"
    so_that: "I can list it for sale on my online store"

    acceptance_criteria:
      - criterion_id: AC001
        description: "Product created with all required fields"
        validates_invariant: CATALOG.Product.REQUIRED_FIELDS  # Grounding to DDD invariant
        test_type: unit

      - criterion_id: AC002
        description: "SKU must be unique across all products"
        validates_invariant: CATALOG.Product.UNIQUE_SKU
        test_type: integration

      - criterion_id: AC003
        description: "Price must be positive number"
        validates_invariant: CATALOG.Product.POSITIVE_PRICE
        test_type: unit

      - criterion_id: AC004
        description: "Product visible in product list after creation"
        validates_workflow: UX.ProductManagement.CreateProductWorkflow  # Forward reference
        test_type: e2e

    workflow_outline:
      steps:
        - "User navigates to Products page"
        - "User clicks 'Add Product' button"
        - "User fills product form (name, SKU, price, description)"
        - "System validates inputs"
        - "System creates Product aggregate"
        - "System redirects to product list"

    technical_tasks:
      - task_id: TASK_001
        description: "Implement Product aggregate in CATALOG context"
        type: domain_model
        references: DDD.Aggregate, strategic-ddd-model.CATALOG.Product
        estimated_hours: 4

      - task_id: TASK_002
        description: "Implement Product repository"
        type: persistence
        references: DDD.Repository
        estimated_hours: 3

      - task_id: TASK_003
        description: "Implement CreateProduct command handler"
        type: application_service
        references: DDD.ApplicationService
        estimated_hours: 3
      - task_id: TASK_004
        description: "Create Product form component"
        type: ui
        references: UX.Component (forward reference)
        estimated_hours: 5

      - task_id: TASK_005
        description: "Write unit tests for Product invariants"
        type: testing
        references: QE.UnitTest (forward reference)
        estimated_hours: 2

    story_points: 5
    priority: P0
```

Phase 5: Quality Engineering Model Refinement Input: sprint-backlog.yaml (with acceptance criteria), strategic-ddd-model.yaml

Canonical Models Used: QE, DDD, Agile, UX (groundings)

Process:

1. **Test Strategy Definition**:
   - LLM generates comprehensive test strategy
   - Defines test levels: unit, integration, E2E, performance, security
   - Coverage targets: >80% for unit, >60% for integration
   - Risk-based prioritization
2. **Invariant-Driven Unit Test Generation**:
   - Extract all invariants from strategic-ddd-model.yaml
   - For each invariant, generate unit test case
   - Grounding: QE.TestCase → DDD.Invariant (procedural, strong)
3. **Acceptance Criteria Test Generation**:
   - For each Agile.AcceptanceCriteria, generate QE.TestCase
   - Grounding: QE.TestCase → Agile.AcceptanceCriteria (epistemic, strong)
4. **Example Test Cases**:

```
# qe-model.yaml
canonical_model_id: project_qe_model_v1
based_on: model_qe
version: 1.0.0

test_strategy:
  approach: "Risk-based testing with emphasis on domain invariants and acceptance criteria 
  test_levels:
    - level: unit
      coverage_target: 85%
```

```yaml
        focus: "Domain invariants, value object validation, aggregate consistency"
      - level: integration
        coverage_target: 65%
        focus: "Cross-aggregate workflows, repository operations, event handling"
      - level: e2e
        coverage_target: "All critical user journeys"
        focus: "Complete workflows from UI through domain to persistence"

  test_suites:
    - suite_id: CATALOG_UNIT_TESTS
      suite_name: "Product Catalog Unit Tests"
      test_level: unit
      bounded_context: CATALOG

      test_cases:
        - test_case_id: TC_CATALOG_001
          name: "Product creation with valid attributes succeeds"
          validates_invariant: CATALOG.Product.REQUIRED_FIELDS  # Grounding to DDD
          test_type: unit
          given: "Valid product attributes (name='Widget', sku='W001', price=19.99)"
          when: "Create Product aggregate"
          then: "Product created successfully with all attributes set"
          assertions:
            - "product.name == 'Widget'"
            - "product.sku == 'W001'"
            - "product.price == 19.99"

        - test_case_id: TC_CATALOG_002
          name: "Product creation with negative price fails"
          validates_invariant: CATALOG.Product.POSITIVE_PRICE
          test_type: unit
          given: "Product attributes with negative price (price=-10.00)"
          when: "Attempt to create Product aggregate"
          then: "InvariantViolationException thrown"
          assertions:
            - "exception.message contains 'Price must be positive'"

        - test_case_id: TC_CATALOG_003
          name: "Product SKU must be unique"
          validates_invariant: CATALOG.Product.UNIQUE_SKU
          test_type: integration
          given: "Existing product with SKU='W001' in repository"
          when: "Attempt to create another product with SKU='W001'"
          then: "DuplicateSKUException thrown"
          assertions:
            - "exception.message contains 'SKU W001 already exists'"
    - suite_id: CATALOG_ACCEPTANCE_TESTS
      suite_name: "Product Catalog Acceptance Tests"
      test_level: e2e

      test_cases:
        - test_case_id: TC_CATALOG_AC_001
          name: "Create product end-to-end"
          validates_acceptance_criterion: CATALOG_001_US001.AC004  # Grounding to Agile
          test_type: e2e
          given: "User logged in as SmallBusinessOwner"
          when:
            - "User navigates to Products page"
            - "User clicks 'Add Product'"
            - "User enters: name='Widget', SKU='W001', price=19.99, description='Test product'
            - "User clicks 'Save'"
          then:
            - "Success message displayed"
            - "User redirected to product list"
            - "Product 'Widget' visible in list with price $19.99"
          automation_status: automated
          test_data_refs:
            - test_data_001  # Reference to test data fixture

  test_data:
    - test_data_id: test_data_001
      name: "Sample Products"
      type: domain_entities
      references_aggregate: CATALOG.Product  # Grounding to DDD
      data:
        - sku: "TEST001"
          name: "Test Product 1"
          price: 10.00
        - sku: "TEST002"
          name: "Test Product 2"
          price: 20.00

  validation:
    all_invariants_tested: true
    all_acceptance_criteria_tested: true
    closure: 100%
```

Phase 6: User Experience Model Refinement Input: sprint-backlog.yaml, strategic-ddd-model.yaml

Canonical Models Used: UX, DDD (via grounding)

Process:

1. **Information Architecture**:
   - LLM generates site map with pages
   - Each page must reference bounded context (grounding validation)
   - Navigation hierarchy generated
2. **Page Design**:
   - For each page, LLM defines: Components, data bindings, actions
   - Grounding: UX.Page → DDD.BoundedContext (structural, strong)
   - Grounding: UX.DataBinding → DDD.Repository (structural)
3. **Workflow Detailed Design**:
   - Expand workflow outlines from Phase 4
   - Define: Steps, validations, state transitions, error handling
   - Grounding: UX.Workflow → DDD.DomainService (procedural, strong)
4. **Example Output**:

```yaml
# ux-model.yaml
canonical_model_id: project_ux_model_v1
based_on: model_ux
version: 1.0.0

information_architecture:
  site_map:
    - page_id: ProductListPage
      path: "/products"
      name: "Products"
      bounded_context_ref: CATALOG  # Required by grounding
      children:
        - page_id: ProductDetailPage
          path: "/products/:id"
        - page_id: ProductFormPage
          path: "/products/new"

pages:
  - page_id: ProductListPage
    name: "Product List"
    bounded_context_ref: CATALOG  # Grounding validation
    purpose: "Display all products with search and filtering"

    components:
      - component_id: ProductSearchBar
        type: SearchInput
        binds_to:
          - property: searchQuery
            updates: ProductListTable.filter
      - component_id: AddProductButton
        type: Button
        label: "Add Product"
        action: navigate_to_ProductFormPage

      - component_id: ProductListTable
        type: DataTable
        columns:
          - SKU
          - Name
          - Price
          - Actions
        data_source:
          repository: CATALOG.ProductRepository  # Grounding to DDD
          query: findAll
          pagination: true
          page_size: 25

  - page_id: ProductFormPage
    name: "Product Form"
    bounded_context_ref: CATALOG
    purpose: "Create or edit product"

    components:
      - component_id: ProductForm
        type: Form
        fields:
          - field_id: name
            label: "Product Name"
            type: text
            required: true
            validation:
              - rule: "length >= 3"
                message: "Name must be at least 3 characters"
            binds_to_property: Product.name  # Grounding to DDD

          - field_id: sku
            label: "SKU"
            type: text
            required: true
            validation:
              - rule: "unique"
                validates_invariant: CATALOG.Product.UNIQUE_SKU
                message: "SKU must be unique"
            binds_to_property: Product.sku
```

```yaml
          - field_id: price
            label: "Price"
            type: number
            required: true
            validation:
              - rule: "value > 0"
                validates_invariant: CATALOG.Product.POSITIVE_PRICE
                message: "Price must be positive"
            binds_to_property: Product.price

        actions:
          - action_id: save
            label: "Save Product"
            triggers_workflow: CreateProductWorkflow

workflows:
  - workflow_id: CreateProductWorkflow
    name: "Create Product Workflow"
    trigger: ProductForm.save_action
    implements_domain_service: CATALOG.CreateProductService  # Grounding to DDD

    steps:
      - step_id: step_001
        order: 1
        description: "Validate form inputs"
        validation_rules:
          - field: name
            rule: "not empty"
          - field: sku
            rule: "unique"
            checks_invariant: CATALOG.Product.UNIQUE_SKU
          - field: price
            rule: "positive"
            checks_invariant: CATALOG.Product.POSITIVE_PRICE
        on_validation_failure: display_errors

      - step_id: step_002
        order: 2
        description: "Create Product aggregate"
        calls_domain_service: CATALOG.CreateProductService
        parameters:
          name: "form.name"
          sku: "form.sku"
          price: "form.price"
          description: "form.description"
        on_success: step_003
        on_failure: display_error_message

      - step_id: step_003
        order: 3
        description: "Navigate to product list"
        action: navigate_to_ProductListPage
        show_notification: "Product created successfully"

validation:
  all_pages_grounded: true
  all_data_bindings_valid: true
  all_workflows_implement_services: true
  closure: 100%
```

Phase 7: Data Engineering Model Definition **Input**: strategic-ddd-model.yaml (aggregates with attributes), ux-model.yaml

Canonical Models Used: Data-Eng, DDD (via semantic grounding)

Process:

1. **Dataset Identification**:
    - Analyze DDD aggregates for persistence needs
    - Generate dataset for each aggregate or aggregate group
    - Grounding: Data-Eng.Dataset semantically aligns with DDD.Aggregate (70% attribute overlap)
2. **Schema Generation**:
    - Map aggregate attributes to dataset schema fields
    - Calculate semantic alignment percentage
    - Warn if alignment <70%
3. **Example**:

```
# data-eng-model.yaml
canonical_model_id: project_data_eng_model_v1
based_on: model_data_eng
version: 1.0.0

datasets:
  - dataset_id: products_table
    name: "Products"
    storage_type: relational
    aligns_with_aggregate: CATALOG.Product  # Semantic grounding

    schema:
      table_name: products
      primary_key: id
      indexes:
```

```yaml
  - columns: [sku]
    unique: true
  - columns: [category_id]

fields:
  - field_name: id
    type: UUID
    nullable: false
    maps_to_property: Product.product_id  # Alignment

  - field_name: sku
    type: VARCHAR(50)
    nullable: false
    unique: true
    maps_to_property: Product.sku

  - field_name: name
    type: VARCHAR(255)
    nullable: false
    maps_to_property: Product.name

  - field_name: price
    type: DECIMAL(10,2)
    nullable: false
    check_constraint: "price > 0"
    maps_to_property: Product.price
    validates_invariant: CATALOG.Product.POSITIVE_PRICE

  - field_name: description
    type: TEXT
    nullable: true
    maps_to_property: Product.description

  - field_name: category_id
    type: UUID
    nullable: true
    foreign_key: categories.id
    maps_to_relationship: Product.belongsTo_Category

  - field_name: created_at
    type: TIMESTAMP
    nullable: false
    default: CURRENT_TIMESTAMP

  - field_name: updated_at
    type: TIMESTAMP
```

```
          nullable: false
          default: CURRENT_TIMESTAMP

    semantic_alignment:
      aggregate: CATALOG.Product
      total_aggregate_properties: 5
      mapped_properties: 5
      alignment_percentage: 100%
      status: EXCELLENT

  - dataset_id: inventory_stock_table
    name: "Inventory Stock"
    aligns_with_aggregate: INVENTORY.StockItem

    schema:
      table_name: inventory_stock
      # ... similar detail

    semantic_alignment:
      aggregate: INVENTORY.StockItem
      alignment_percentage: 92%
      status: GOOD

pipelines:
  - pipeline_id: product_catalog_sync
    name: "Product Catalog Sync Pipeline"
    type: ETL
    schedule: "real-time (event-driven)"

    source:
      type: database
      dataset: products_table

    transformations:
      - transformation_id: enrich_product_data
        description: "Join with categories and pricing rules"

    sink:
      type: search_index
      dataset: products_search_index
      purpose: "Enable fast product search in UX"

validation:
  all_aggregates_mapped: true
  semantic_alignment_threshold_met: true
  closure: 100%
```

Phase 8: Bounded Code Generation **Input**: All models from Phases 2-7, prioritized user stories

Canonical Models Used: All (DDD, UX, QE, Data-Eng, Agile)

Process:

1. **Code Generation with Full Schema Context**:
   - Developer selects task: "Implement Product aggregate"
   - System loads ALL relevant models:
     - strategic-ddd-model.yaml (Product aggregate spec)
     - ux-model.yaml (ProductForm, workflows using Product)
     - qe-model.yaml (Product invariant tests)
     - data-eng-model.yaml (products_table schema)
   - Schema context size: ~30,000-50,000 tokens
2. **Constrained Generation Example**:

Prompt to LLM:

```
Task: Implement the Product aggregate from the CATALOG bounded context.

Context:
- DDD Canonical Model: Aggregate pattern requires root entity, identity-based references, in
- Strategic DDD Model: Product aggregate specification (see below)
- QE Model: Unit tests that must pass (see below)
- Data-Eng Model: Database schema this aggregate persists to (see below)

Product Aggregate Specification:
{strategic-ddd-model.CATALOG.Product}

Invariants that MUST be enforced:
1. REQUIRED_FIELDS: name, sku, price must be provided
2. POSITIVE_PRICE: price > 0
3. UNIQUE_SKU: sku must be unique (checked at repository level)

Test Cases that MUST pass:
{qe-model.TC_CATALOG_001, TC_CATALOG_002, TC_CATALOG_003}

Generate Python code for Product aggregate following DDD patterns.
```

LLM Generated Code:

```python
# catalog/domain/model/product.py

from dataclasses import dataclass
from decimal import Decimal
from typing import Optional
from uuid import UUID, uuid4
```

```
@dataclass(frozen=True)
class SKU:
    """Value Object: Product SKU"""
    value: str

    def __post_init__(self):
        if not self.value:
            raise ValueError("SKU cannot be empty")
        if len(self.value) > 50:
            raise ValueError("SKU cannot exceed 50 characters")

@dataclass(frozen=True)
class Money:
    """Value Object: Price"""
    amount: Decimal
    currency: str = "USD"

    def __post_init__(self):
        if self.amount <= 0:
            raise ValueError("Price must be positive")  # Enforces POSITIVE_PRICE invariant

class Product:
    """
    Aggregate Root: Product

    Bounded Context: CATALOG
    Responsibility: Manage product information for catalog

    Invariants:
    - REQUIRED_FIELDS: name, sku, price must be provided
    - POSITIVE_PRICE: price > 0 (enforced by Money value object)
    - UNIQUE_SKU: sku must be unique (enforced by repository)
    """

    def __init__(
        self,
        product_id: UUID,
        sku: SKU,
        name: str,
        price: Money,
        description: Optional[str] = None
    ):
        # Enforce REQUIRED_FIELDS invariant
        if not name:
            raise ValueError("Product name is required")
        if not sku:
            raise ValueError("Product SKU is required")
        if not price:
            raise ValueError("Product price is required")

        self._product_id = product_id
        self._sku = sku
        self._name = name
        self._price = price
        self._description = description

    @staticmethod
    def create(sku: str, name: str, price: Decimal, description: Optional[str] = None) -> 'P
        """Factory method for creating new Product"""
        return Product(
            product_id=uuid4(),
            sku=SKU(sku),
            name=name,
            price=Money(price),
            description=description
        )

    @property
    def product_id(self) -> UUID:
        return self._product_id

    @property
    def sku(self) -> SKU:
        return self._sku

    @property
    def name(self) -> str:
        return self._name

    @property
    def price(self) -> Money:
        return self._price

    def update_price(self, new_price: Decimal) -> None:
        """Update product price (enforces POSITIVE_PRICE)"""
        self._price = Money(new_price)  # Money validates positive

    def __eq__(self, other):
        if not isinstance(other, Product):
            return False
        return self._product_id == other._product_id
    def __hash__(self):
        return hash(self._product_id)
```

3. **Validation**:
   - Run generated unit tests from QE model
   - Verify invariants enforced
   - Check code follows DDD patterns (aggregate root, value objects, etc.)
4. **Result**: Code that is:
   - Consistent with domain model specification
   - Validates all invariants
   - Passes all tests
   - Follows canonical patterns

Phase 9: Continuous Evolution with Impact Analysis **Scenario**: Add "quantity discount" feature requiring new aggregate concepts

Process:

1. **Change Request**: "Add volume pricing: customers get discounts when buying quantities >10"
2. **Impact Analysis** (using Ripple Effect Management from earlier):
   - Changed model: DDD (add QuantityDiscount concept to Product aggregate)
   - Affected models:
     - UX: ProductForm needs discount fields
     - QE: New tests for discount calculation
     - Data-Eng: products_table needs discount columns
     - Agile: Update feature description
3. **Coordinated Regeneration**:
   - All affected artifacts regenerated with new constraint
   - Validation ensures consistency
   - Version bumps: All models go from x.y.z $\rightarrow$ x.(y+1).0
4. **Outcome**: System-wide consistency maintained through formal process

This completes the detailed nine-phase workflow specification demonstrating the invention's systematic approach to multi-domain LLM-assisted development with formal validation at every step.

Implementation Details

Programming Languages: Python 3.10+ for validation engine and workflow orchestration; TypeScript/JavaScript for web-based visualization.

Data Formats: YAML or JSON for canonical domain models conforming to JSON Schema 2020-12; GraphML or DOT for grounding graph visualization.

LLM Integration: REST API clients for OpenAI, Anthropic, or local inference engines with authentication, rate limiting, and retry logic.

Validation Libraries: `jsonschema` (Python) for meta-schema validation; custom validators for closure calculation and grounding verification.

Workflow Orchestration: LangGraph or Apache Airflow for state machine management with persistence and audit trails.

Version Control: Git with semantic versioning for all canonical domain models; Git hooks for automated validation on commit.

Visualization: Graphviz for DOT graph generation; React-based web application for interactive model navigation.

Deployment: Docker containers with Kubernetes orchestration; PostgreSQL for workflow state; Redis for schema context caching.

CLAIMS

What is claimed is:

1. A computer-implemented system for coordinating multi-domain knowledge in software development, comprising:

a memory storing a plurality of canonical domain model data structures, each canonical domain model data structure comprising: a unique identifier, a domain layer designation selected from foundation, derived, and meta, a set of concepts representing core abstractions within a knowledge domain, each concept comprising a concept identifier, properties, and relationships, a set of grounding relationships referencing other canonical domain model data structures, each grounding relationship comprising: a grounding type selected from structural, semantic, procedural, and epistemic, a set of concept mappings defining relationships between concepts in a source canonical domain model and concepts in a target canonical domain model with cardinality constraints, and a strength designation selected from strong, weak, and optional, a set of constraints defining validation rules with severity levels, and a semantic version;

a processor configured to: receive a request to generate a software artifact spanning multiple knowledge domains, identify a set of required canonical domain models based on the request, load the required canonical domain models from the memory, transitively load additional canonical domain models referenced by grounding relationships from the loaded required canonical domain models, construct a unified schema context by merging concepts, patterns, and constraints from all loaded canonical domain models with qualified naming to prevent terminology conflicts, validate the unified schema context for acyclicity in the grounding relationships and absence of contradictory constraints, transmit the unified schema context and the request to a large language model API for constrained generation, receive a generated software artifact from the large language model API, validate the generated software artifact against the unified schema context by: verifying that all concept references resolve within the loaded canonical domain models, verifying that all external concept references are supported by grounding relationships from the artifact's primary canonical domain model to the target canonical domain models containing the referenced concepts, verifying that concept mappings exist in the grounding relationships for all external concept references, and verifying that all constraints in the unified schema context are satisfied, and output the validated software artifact when validation succeeds.

2. The system of claim 1, wherein:

the grounding type structural indicates that the target canonical domain model provides foundational concepts that the source canonical domain model builds upon, the grounding type semantic indicates that the target canonical domain model provides meaning or interpretation for concepts in the source canonical domain model, the grounding type procedural indicates that the target canonical domain model defines processes that the source canonical domain model follows or validates, and the grounding type epistemic indicates that the target canonical domain model provides assumptions or justifications for concepts in the source canonical domain model.

3. The system of claim 1, wherein:

the strength designation strong indicates a hard constraint that must be satisfied and violations generate errors, the strength designation weak indicates soft guidance and violations generate warnings, and the strength designation optional indicates informational guidance and violations generate informational messages.

4. The system of claim 1, wherein the processor is further configured to:

for each canonical domain model, calculate a closure percentage by: identifying all concept references within the canonical domain model, classifying each concept reference as internal if the referenced concept is within the same canonical domain model or external if the referenced concept is in a different canonical domain model, for each external concept reference, determining whether a grounding relationship exists from the canonical domain model to the canonical domain model containing the referenced concept and whether the grounding relationship includes a concept mapping for the specific referenced concept, calculating the closure percentage as: (count of internal concept references + count of external concept references with valid grounding and concept mapping) / (total count of all concept references) multiplied by 100, compare the closure percentage to a threshold value, and generate a warning if the closure percentage is below the threshold value.

5. The system of claim 4, wherein the threshold value is 95 percent, and the processor generates an error preventing production use if the closure percentage is below 80 percent.

6. The system of claim 1, wherein the processor transmits the unified schema context and the request to the large language model API by:

constructing a prompt comprising: the request, a schema overview from the unified schema context, relevant concepts from the unified schema context with properties and relationships, applicable patterns from the unified schema context with intent and structure, constraints from the unified schema context with validation rules, and examples from the canonical domain models in the unified schema context.

7. The system of claim 1, wherein the processor validates the generated software artifact by:

performing syntactic validation by parsing the artifact and verifying data types, performing semantic validation by resolving all concept references and evaluating all constraint predicates, performing cross-domain consistency validation by, for each external concept reference: locating a grounding relationship from the artifact's primary canonical domain model to the target canonical domain model, verifying the grounding relationship includes a concept mapping for the referenced concept, verifying cardinality constraints are satisfied, and executing grounding-specific validation rules.

8. The system of claim 1, wherein the processor is further configured to:

detect a change to a concept in a first canonical domain model, traverse the grounding relationships to identify one or more second canonical domain models having grounding relationships targeting the first canonical domain model and referencing the changed concept, generate an impact report listing the identified second canonical domain models and validation failures resulting from the change, receive user approval to propagate the change, for each identified second canonical domain model: construct a corrective prompt describing the change and validation failures, transmit the corrective prompt and an updated unified schema context including the changed first canonical domain model to the large language model API, receive a regenerated artifact from the large language model API, validate the regenerated artifact, and update the second canonical domain model with the regenerated artifact if validation succeeds, update semantic versions of all modified canonical domain models, and recalculate system-wide closure percentage across all canonical domain models.

9. The system of claim 8, wherein the processor updates semantic versions by:

incrementing a major version number for the first canonical domain model if the change removes a concept or introduces breaking changes, incrementing a minor version number for the first canonical domain model if the change adds a new concept, and incrementing a minor version number for each second canonical domain model that was updated through ripple effect propagation.

10. A computer-implemented method for validating completeness of a canonical domain model, comprising:

loading, by a processor, a canonical domain model data structure from a memory, the canonical domain model data structure comprising: a unique identifier, a set of concepts, each concept comprising properties and relationships to other concepts, and a set of grounding relationships to other canonical domain models, each grounding relationship comprising a set of concept mappings;

identifying, by the processor, all concept references within the canonical domain model from the relationships in the concepts;

classifying, by the processor, each concept reference as internal if a target concept identifier matches a concept in the set of concepts of the canonical domain model, or external if the target concept identifier references a concept in a different canonical domain model;

for each external concept reference: determining, by the processor, whether a grounding relationship exists in the set of grounding relationships targeting the canonical domain model containing the referenced concept, if the grounding relationship exists, determining, by the processor, whether the grounding relationship includes a concept mapping that maps to the specific referenced concept, incrementing, by the processor, a grounded external count if both the grounding relationship exists and the concept mapping exists for the referenced concept, otherwise, adding, by the processor, the external concept reference to an ungrounded references list;

calculating, by the processor, a closure percentage as: (count of internal concept references + grounded external count) divided by (count of internal concept references + count of all external concept references), multiplied by 100;

comparing, by the processor, the closure percentage to a threshold value of 95 percent;

if the closure percentage is below the threshold value: generating, by the processor, a validation message indicating insufficient model completeness and including the ungrounded references list; and

outputting, by the processor, the closure percentage and the validation message.

11. The method of claim 10, further comprising:

if the closure percentage is below 80 percent, generating, by the processor, an error status preventing use of the canonical domain model in production environments.

12. The method of claim 10, wherein the concept mappings in each grounding relationship further comprise:

a source concept identifier referencing a concept in a source canonical domain model, a target concept identifier referencing a concept in a target canonical domain model, a mapping type selected from implements, validates, references, aligns, and derives_from, a cardinality specification, and a bidirectional indicator.

13. The method of claim 10, further comprising:

for a plurality of canonical domain models in a system, calculating, by the processor, individual closure percentages for each canonical domain model, calculating, by the processor, a system-wide closure percentage as a weighted average of the individual closure percentages, and if the system-wide closure percentage exceeds 95 percent, designating, by the processor, the system as production-ready.

14. A computer-implemented method for generating software artifacts with large language model constraint, comprising:

receiving, by a processor, a task description for generating a software artifact;

identifying, by the processor, a set of canonical domain models relevant to the task description based on keywords in the task description;

loading, by the processor, canonical domain model schemas for the identified canonical domain models from a repository, wherein each canonical domain model schema defines: concepts with properties and relationships, patterns with applicability rules, constraints with validation functions and severity levels, and grounding relationships to other canonical domain models with concept mappings;

constructing, by the processor, a unified schema context by: merging concepts from all loaded canonical domain model schemas with qualified naming using model identifiers as prefixes, merging patterns from all loaded canonical domain model schemas, merging constraints from all loaded canonical domain model schemas, building a grounding relationship graph from the grounding relationships, and validating absence of cycles in the grounding relationship graph;

transmitting, by the processor, a prompt to a large language model system, the prompt comprising the task description and the unified schema context;

receiving, by the processor, from the large language model system, a generated software artifact;

validating, by the processor, the generated software artifact against the unified schema context by: resolving all concept references in the generated software artifact to concepts in the unified schema context, for each concept reference that resolves to a concept in a different canonical domain model than a primary canonical domain model of the generated software artifact: verifying that a grounding relationship exists from the primary canonical domain model to the canonical domain model containing the referenced concept, and verifying that the grounding relationship includes a concept mapping for the referenced concept, evaluating all constraint validation functions from the unified schema context against the generated software artifact, and accumulating validation errors for constraint violations with error severity, warnings for constraint violations with warning severity, and informational messages for constraint violations with info severity;

if validation errors exist: constructing, by the processor, a corrective prompt comprising the task description, the unified schema context, the generated software artifact, and the validation errors, transmitting, by the processor, the corrective prompt to the large language model system, receiving, by the processor, a regenerated software artifact from the large language model system, and re-validating, by the processor, the regenerated software artifact;

outputting, by the processor, a valid software artifact when validation succeeds with no errors.

15. The method of claim 14, wherein identifying the set of canonical domain models comprises:

parsing the task description for domain-specific keywords, applying heuristic rules that map keywords to canonical domain model identifiers, wherein: keywords including "aggregate", "bounded context", "domain", "invariant", or "entity" map to a domain-driven design canonical domain model, keywords including "workflow", "page", "component", "navigation", or "user interface" map to a user experience canonical domain model, keywords including "test", "quality", "validation", "assertion", or "coverage" map to a quality engineering canonical domain model, keywords including "pipeline", "dataset", "schema", "ETL", or "data warehouse" map to a data engineering canonical domain model, and keywords including "epic", "feature", "story", "backlog", or "vision" map to an agile project management canonical domain model, and including all canonical domain models identified by the heuristic rules in the set of canonical domain models.

16. The method of claim 14, wherein constructing the unified schema context further comprises:

for each loaded canonical domain model schema, extracting grounding relationships, for each grounding relationship, identifying a target canonical domain model, if the target canonical domain model is not in the loaded canonical domain model schemas, recursively loading the target canonical domain model schema and its transitive grounding dependencies, and repeating the recursive loading until no new canonical domain models are identified.

17. The method of claim 14, wherein transmitting the prompt to the large language model system comprises:

generating k candidate software artifact continuations using beam search with a beam width of k, wherein k is an integer between 3 and 10, for each candidate continuation: appending the candidate continuation to a current partial software artifact, performing incremental validation of the appended partial software artifact against the unified schema context, if the incremental validation succeeds, retaining the candidate continuation with an associated probability score, if the incremental validation fails, discarding the candidate continuation, selecting, by the processor, a highest-probability retained candidate continuation, and repeating the generation and selection until the software artifact is complete.

18. The method of claim 14, further comprising:

generating, by the processor, an explanation for the generated software artifact comprising: for each design decision in the generated software artifact: identifying concepts from the unified schema context referenced by the design decision, identifying patterns from the unified schema context applied by the design decision, identifying constraints from the unified schema context satisfied by the design decision, generating a justification describing why the design decision was made based on the identified concepts, patterns, and constraints, and including citations to the canonical domain model schemas for the identified concepts, patterns, and constraints, and outputting the explanation with the valid software artifact.

19. A computer-implemented method for coordinated multi-model software development, comprising:

receiving, by a processor, a product vision document;

validating, by the processor, the product vision document for completeness using an Agile canonical domain model schema that defines required elements comprising problem statement, target users, value proposition, success metrics, constraints, and assumptions;

generating, by a processor, using a large language model, a strategic domain-driven design model comprising a set of bounded contexts based on the validated product vision document, wherein each bounded context includes a context identifier, responsibility statement, and set of core concepts;

decomposing, by the processor, using the large language model, the product vision into a set of epics and features, wherein each epic comprises an epic identifier and a reference to at least one bounded context from the strategic domain-driven design model;

validating, by the processor, that the reference from each epic to at least one bounded context is supported by a grounding relationship from an Agile canonical domain model to a domain-driven design canonical domain model;

generating, by the processor, using the large language model, a set of user stories for a subset of the features selected for implementation, wherein each user story comprises a user story identifier, acceptance criteria, and references to workflows and domain concepts;

generating, by the processor, using the large language model, a quality engineering model comprising a test strategy and a set of test cases, wherein each test case validates at least one of: a domain invariant from the strategic domain-driven design model, an acceptance criterion from a user story, or a workflow from a user experience model;

generating, by the processor, using the large language model, the user experience model comprising a set of pages and workflows, wherein each page references a bounded context from the strategic domain-driven design model through a structural grounding relationship;

generating, by the processor, using the large language model, a data engineering model comprising a set of datasets, wherein each dataset is semantically aligned with at least one aggregate from the strategic domain-driven design model with an alignment score of at least 70 percent;

calculating, by the processor, a system-wide closure percentage across the strategic domain-driven design model, the quality engineering model, the user experience model, and the data engineering model;

if the system-wide closure percentage exceeds 95 percent: designating, by the processor, the generated models as production-ready, and generating, by the processor, using the large language model constrained by all generated models, implementation code for the selected features; and

outputting, by the processor, the generated models and implementation code.

20. The method of claim 19, further comprising:

receiving, by the processor, a change request to modify a concept in the strategic domain-driven design model;

traversing, by the processor, grounding relationships to identify models from the quality engineering model, the user experience model, and the data engineering model that reference the concept to be modified;

generating, by the processor, an impact report listing the identified models;

receiving, by the processor, user approval to proceed with the change;

applying, by the processor, the change to the concept in the strategic domain-driven design model;

for each identified model: using, by the processor, the large language model to regenerate artifacts in the identified model that reference the modified concept, wherein the regeneration is constrained by the modified strategic domain-driven design model, validating, by the processor, the regenerated artifacts, and updating, by the processor, the identified model with the regenerated artifacts;

incrementing, by the processor, version numbers for the strategic domain-driven design model and all updated identified models according to semantic versioning; and

outputting, by the processor, an audit trail documenting the change, affected models, regenerated artifacts, and updated version numbers.

21. The method of claim 19, wherein generating the strategic domain-driven design model comprises:

transmitting, by the processor, to the large language model, a prompt comprising the validated product vision document and a domain-driven design canonical domain model schema defining bounded context patterns and ubiquitous language principles, receiving, by the processor, from the large language model, a proposed strategic domain-driven design model, presenting, by the processor, the proposed strategic domain-driven design model to a domain expert for review, receiving, by the processor, feedback from the domain expert comprising approvals and requested modifications, if modifications are requested: regenerating, by the processor, using the large language model, the strategic domain-driven design model incorporating the requested modifications, and repeating the presentation and feedback steps until the domain expert approves the strategic domain-driven design model, and persisting, by the processor, the approved strategic domain-driven design model to a repository with version 1.0.0.

22. The method of claim 19, wherein calculating the system-wide closure percentage comprises:

for each model in the set comprising the strategic domain-driven design model, the quality engineering model, the user experience model, and the data engineering model: calculating an individual closure percentage as described in claim 10, calculating a weighted average of the individual closure percentages based on a number of concepts in each model, and designating the weighted average as the system-wide closure percentage.

23. A non-transitory computer-readable storage medium storing instructions that, when executed by a processor, cause the processor to perform the method of claim 10.

24. A non-transitory computer-readable storage medium storing instructions that, when executed by a processor, cause the processor to perform the method of claim 14.

25. A non-transitory computer-readable storage medium storing instructions that, when executed by a processor, cause the processor to perform the method of claim 19.

ABSTRACT

A computer-implemented system and method for multi-domain knowledge coordination in large language model-assisted software development uses canonical domain models with explicit grounding relationships. Each canonical domain model comprises concepts, patterns, constraints, and typed grounding relationships (structural, semantic, procedural, epistemic) to other models. The system calculates a closure property indicating reference resolution completeness and validates acyclicity. During software artifact generation, the system loads relevant canonical domain models with transitive dependencies, constructs a unified schema context, and constrains LLM generation through validation. Generated artifacts are validated for cross-domain consistency via grounding relationships. The system automates ripple effect management by traversing the grounding graph to identify and update affected models when changes occur. Empirical results show 25-50% accuracy improvement, 4-7x faster solution synthesis, and 3x fewer integration defects with >95% closure. Applications include systematic greenfield development from product vision through strategic domain modeling, epic decomposition, user story generation, quality engineering refinement, user experience design, data engineering modeling, to bounded code generation with continuous evolution.

END OF PATENT APPLICATION