Table of 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 × 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 → DDD.BoundedContext (1:1) and UX.Workflow → 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 applicability scope of a ubiquitous language and provides a context for model interpretation."  
 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 interface for accessing domain objects"  
 structure: "Interface defining collection-like operations (add, remove, find) for aggregate access"  
 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 with in-memory implementations. Aggregate boundaries enforced through repository."  
 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 object reference"  
 rule: "forall(relationship in aggregate.relationships where relationship.target.is\_aggregate: relationship.type == 'identity\_reference')"  
 severity: warning  
 message: "Aggregate {aggregate.id} directly references aggregate {target.id}; consider identity reference"

**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):

$schema: "https://json-schema.org/draft/2020-12/schema"  
canonical\_model\_id: model\_ddd  
domain: "Domain-Driven Design"  
layer: foundation  
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 boundary"  
 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 domain scope"  
  
 - 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**:

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 threshold of 80%. Model not suitable for production use. Add missing grounding relationships."  
 )  
  
 ELSE IF closure\_percentage < 95.0:  
 RETURN ValidationStatus(  
 level: WARNING,  
 message: "WARNING: Model closure {closure\_percentage:.1f}% is below target threshold of 95%. Model may be used but additional grounding recommended for production readiness."  
 )  
  
 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: GroundingMap)  
 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, grounding\_map: GroundingMap)  
 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 {constraint2.qualified\_id}")  
  
 // 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\_name}")  
  
 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: LLMInterface)  
 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=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 {pattern.qualified\_id}")  
  
 // 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 {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.groundings)  
  
 IF grounding IS NULL:  
 errors.APPEND(f"No grounding relationship from {artifact.primary\_model\_id} to {ext\_ref.model\_id} for reference at {ext\_ref.location}")  
 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 concept {ext\_ref.concept\_id}")  
 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\_cardinality} times, expected {concept\_mapping.cardinality}")  
  
 // 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%}, below recommended 70%")  
  
 // 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\_log: GenerationLog)  
 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.description}")  
  
 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. Cross-domain consistency verified."  
  
 // 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 invariants.  
  
 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 entity (Order). DDD.Repository pattern provides aggregate loading."  
 sources:  
 - "model\_ddd.Aggregate: Cluster of entities with transactional consistency boundary"  
 - "model\_ddd.Repository: Mediate between domain and data mapping using collection interface"  
 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 root"  
 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 checked before persisting."  
 sources:  
 - "model\_ddd.Invariant: Business rule that must always be true within aggregate boundary"  
 constraints\_satisfied:  
 - "model\_ddd.INVARIANT\_ENFORCED: Order aggregate enforces total calculation invariant"  
  
 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\_id)  
  
 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\_interface)  
  
 // 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}. Errors: {final\_validation.errors}"  
  
 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 fix  
 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}% after ripple effect"  
 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**:

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 Satisfaction score"  
 - "Specify budget constraint: e.g., Development budget $500K"  
 - "Specify timeline: e.g., MVP in 6 months"

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

# vision.yaml  
vision:  
 vision\_id: ECOM\_PLATFORM\_V1  
 problem\_statement: "Small/medium businesses lack affordable e-commerce platforms with integrated inventory management"  
 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, order management"  
 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 from inventory and orders"  
 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 different stakeholders than product information"  
 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 audit trail"  
 core\_concepts:  
 - Order  
 - OrderLine  
 - Customer  
 - OrderStatus  
 - Payment  
 ubiquitous\_language:  
 order: "Customer purchase request with line items and fulfillment status"  
 order\_line: "Individual product and quantity within an order"

1. **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”
2. **Context Map Generation**:
   * LLM generates relationships between contexts using DDD context map patterns
   * Patterns: Shared Kernel, Customer-Supplier, Conformist, Anti-Corruption Layer
3. **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)
4. **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
5. **Output Artifact**:

# 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 availability"  
  
 - 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**:

# 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 with categories and attributes"  
 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, images"  
 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  
  
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**:

# 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 validation"  
 test\_levels:  
 - level: unit  
 coverage\_target: 85%  
 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**:

# 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  
  
 - 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:  
 - 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, invariant enforcement  
- 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**:

# 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) -> 'Product':  
 """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)

1. **Validation**:
   * Run generated unit tests from QE model
   * Verify invariants enforced
   * Check code follows DDD patterns (aggregate root, value objects, etc.)
2. **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 → 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