Intent Specification Language (ISL) v1.4

Official Specification Document

Table of Contents

Abstract

Intent Specification Language (ISL) is a human-first, LLM-executable specification format designed for deterministic code generation. ISL enables developers to write high-level behavioral specifications in structured Markdown that can be consistently interpreted by Large Language Models to produce semantically identical code across multiple generations.

Key Characteristics

- **Human-Writable**: Natural language in Markdown format
- **LLM-Executable**: Structured semantics for deterministic interpretation
- **Universal**: Supports both frontend (Presentation) and backend (Backend) components
- **Contract-Based**: Explicit input/output signatures with behavioral contracts
- **Test-Integrated**: Built-in acceptance criteria and test scenarios
- **Deterministic**: Canonical rules enforce consistent code generation

Version Information

- **Current Version**: 1.4
- **Release Date**: January 2026
- **Status**: Stable
- **License**: [To be determined]

Motivation

The Problem

Modern software development faces several challenges when translating requirements into code:

1. **Ambiguity**: Natural language specifications are often interpreted differently by different developers (or AI models)
2. **Non-Determinism**: LLM-generated code varies significantly between runs, even with identical inputs
3. **Incomplete Specifications**: Traditional documentation lacks formalized contracts, constraints, and test criteria
4. **Context Loss**: Implementation details are separated from behavioral intent
5. **Multi-Domain Friction**: Frontend and backend specifications use different formats and vocabularies

The Solution: ISL

ISL addresses these challenges by providing:

- **Structured Natural Language**: Readable by humans, parseable by LLMs
- **Canonical Interpretation Rules**: Eliminates ambiguity in LLM code generation
- **Contract-Driven Design**: Explicit input/output signatures with behavioral promises
- **Integrated Testing**: Acceptance criteria and test scenarios embedded in specifications
- **Universal Format**: Single specification language for UI components, backend services, and APIs
- **Deterministic Output**: Same specification → semantically identical code (verified across 10+ generations)

Target Users

- **Software Architects**: Define system components with precise contracts
- **Frontend Developers**: Specify UI components with behavioral guarantees
- **Backend Engineers**: Document API endpoints and service methods
- **QA Engineers**: Extract test scenarios directly from specifications
- **AI-Assisted Development**: Provide deterministic input for LLM code generators

---

Design Principles

1. Human-First, Machine-Executable

Principle: Specifications should be written in natural language that humans find intuitive, while maintaining sufficient structure for deterministic machine interpretation.

Implementation:

- Markdown as the base format (universal, readable)
- Semantic emoji tags (📐 📦 ⚡ 🚨 ☑ 💡 🔧) for visual categorization
- Optional sections to avoid unnecessary verbosity

2. Contract-Based Behavior

Principle: Every capability must explicitly declare what it promises to do (contract), what triggers it, and what it produces.

Implementation:

- Mandatory `Contract` field for all capabilities
- Optional but recommended `Signature` (input/output) for type safety
- Clear separation between promise (contract) and implementation (hints)

3. Deterministic Interpretation

Principle: Given the same ISL specification, any compliant LLM must generate semantically equivalent code.

Implementation:

- Canonical Rules define interpretation precedence
- Normative sections (⚡ 🚨 ☑ 🔧) are mandatory to implement

- Domain Concepts enforce shared vocabulary

4. Separation of Concerns

Principle: Presentation logic and business logic must be clearly separated.

Implementation:

- Mandatory `Role` field (Presentation | Backend)
- Role-specific validation rules (Rule 4 in Canonical Rules)
- Different section semantics based on role

5. Test-Driven by Design

Principle: Specifications should directly generate testable assertions.

Implementation:

- Component-level Acceptance Criteria
- Capability-level Test Scenarios
- Clear expected outcomes for each scenario

6. Evolutionary Flexibility

Principle: The format must accommodate new patterns without breaking existing specifications.

Implementation:

- Most sections marked (OPTIONAL)
- Clear versioning strategy
- Extensible via custom sections

Document Structure

Hierarchical Organization

```
# Project
├─ Domain Concepts
│  └─ Entities (shared vocabulary)
├─ Component A
│  ├─ Role (context)
│  ├─ Appearance/Interface (visual/API)
│  ├─ Content/Structure (composition)
│  ├─ Capabilities/Methods (behaviors)
│  │  ├─ Signature (I/O types)
│  │  ├─ Contract (promise)
│  │  ├─ Trigger (activation)
│  │  ├─ Flow (steps)
│  │  ├─ Side Effect (mutations)
│  │  ├─ Cleanup (finalization)
│  │  ├─ Implementation Hint (guidance)
│  │  ├─ Constraint (limits)
│  │  └─ Acceptance + Tests (verification)
│  ├─ Global Implementation Hints
│  ├─ Global Constraints
│  ├─ Acceptance Criteria
│  └─ Test Scenarios
└─ Component B...
```

Section Levels

Level	Markdown Syntax	Purpose
Project	`# Project: Name`	Top-level container
Domain	`## Domain Concepts`	Shared vocabulary
Component	`## Component: Name`	Unit of implementation
Subsection	`### Role`, `### 📐 Appearance`	Component properties
Capability	`#### CapabilityName`	Individual behaviors

Canonical Rules

The Canonical Rules define how ISL documents MUST be interpreted by compliant LLM code generators. These rules are **normative** and override any ambiguity in the template structure.

Rule 1: Interpretation Rules

Sections marked with semantic emoji (⚡ 🚨 ☑ 🧪) are NORMATIVE.

- ⚡ **Capabilities/Methods**: MUST be implemented exactly as specified
- 🚨 **Constraints**: MUST be enforced (violations are errors)
- ☑ **Acceptance Criteria**: Define completion (tests must pass)
- 🧪 **Test Scenarios**: MUST be implementable as automated tests

Other sections (📐 📦 💡) are INFORMATIVE:

- 📐 **Appearance/Interface**: Guidance for styling/API design
- 📦 **Content/Structure**: Recommended composition
- 💡 **Implementation Hints**: Suggested strategies (not mandatory)

Interpretation:

```
IF section has normative emoji ( ⚡ 🚨 ☑ 🧪 ) THEN
  Implementation MUST comply exactly
ELSE
  Implementation MAY adapt for target platform
```

Rule 2: Precedence Hierarchy

When conflicts arise, apply this precedence order (highest to lowest):

1. **Capability-Level Constraints** (🚨 in specific capability)
2. **Global Constraints** (🚨 at component level)
3. **Acceptance Criteria** (☑)
4. **Contracts** (behavioral promises)
5. **Implementation Hints** (💡 suggestions)

Example:

markdown

```
🚨 Global Constraint: Use REST for all APIs

Component: WebSocketService
⚡ Capability: streamUpdates
🚨 Constraint: MUST use WebSocket protocol

→ Capability constraint wins (more specific)
```

Rule 3: Determinism Requirements

Requirement: Given the same ISL specification, output code MUST be semantically identical across multiple generations.

Semantic Equivalence Defined:

- Same input → same output (function signatures match)
- Same side effects (state mutations equivalent)
- Same error handling (exception types match)
- Same API contracts (external interfaces identical)

Allowed Variations:

- Variable names (if not specified in Domain Concepts)
- Code comments
- Whitespace and formatting
- Implementation internals (if contract satisfied)

Naming Rule:

- Entity names MUST follow Domain Concepts vocabulary
- Capability names MUST be preserved in generated code
- Type names MUST match or be semantically equivalent

Verification Test:

```
FOR i = 1 TO 10:
  code[i] = generateCode(sameISL)

ASSERT all(semanticallyEquivalent(code[i], code[1]))
```

Rule 4: Role Separation Rules

Presentation Components (Role: Presentation):

MUST:

- Define visual appearance (CSS, layout, styling)
- Handle user interactions (clicks, keyboard, gestures)
- Render UI elements (HTML, JSX, templates)

MUST NOT:

- Implement business logic (validation, calculations)
- Directly access databases or external APIs
- Contain authentication/authorization logic

Backend Components (Role: Backend):

MUST:

- Define API contracts (routes, methods, schemas)
- Implement business logic
- Handle data persistence and external integrations

MUST NOT:

- Define visual properties (colors, fonts, layouts)
- Contain UI rendering code
- Reference DOM elements

Violation Examples:

markdown

```
❌ INVALID:
Component: LoginButton
Role: Presentation
⚡ validatePassword
  Flow:
    1. Check password length >= 8
    2. Verify special characters
    → Business logic in Presentation (violates Rule 4)

☑ VALID:
Component: LoginButton
Role: Presentation
⚡ onClick
  Contract: Trigger authentication via AuthService
  Flow:
    1. Collect form data
    2. Call AuthService.login(email, password)
    3. Handle response (redirect or show error)
    → Delegates business logic to Backend
```

Rule 5: Optionality Semantics

Definition of (OPTIONAL):

- **May be omitted**: If a section is not needed, it can be left out entirely
- **May NOT be ignored**: If a section is present, it MUST be implemented

Contrast with (REQUIRED):

- **Must be present**: Section cannot be omitted
- **Must be implemented**: Content must be reflected in generated code

Example:

markdown

```
### Role: Presentation (REQUIRED)
→ Must appear in every component specification

### 📐 Appearance (OPTIONAL)
- Button with blue background
→ If present, generated code MUST include blue background
→ If absent, default styling may be applied

💡 Implementation Hint (OPTIONAL)
- Use Tailwind utility classes
→ If present, Tailwind is RECOMMENDED but not mandatory
→ Other CSS solutions acceptable if contracts met
```

Interpretation:

```
IF section marked (OPTIONAL) AND section.isEmpty THEN
  Skip section
ELSE IF section marked (OPTIONAL) AND section.isPresent THEN
  Implement section content
ELSE IF section marked (REQUIRED) THEN
  Error if section.isEmpty
  Implement section content
```

Rule 6: Error Handling (Supplementary)

Constraint Violations:

- LLM MUST NOT generate code that violates 🚨 Constraints
- If implementation is impossible, LLM MUST report error with reasons
- Silent failures are NOT permitted

Ambiguity Resolution:

- If specification is ambiguous, LLM MUST request clarification
- LLM MAY suggest interpretations but MUST NOT assume

Unimplementable Requirements:

- If technical limitation prevents implementation, LLM MUST:
  1. Report which requirement cannot be met
  2. Explain technical reason
  3. Suggest alternative approaches that preserve intent

Example:

markdown

```
🚨 Constraint: Response time MUST be < 1ms

→ If physically impossible (network latency):
   LLM reports: "1ms response time unachievable over network.
                 Suggest: local caching or relaxed constraint."
```

Notation & Conventions

Semantic Emoji Tags

ISL uses emoji as semantic markers to categorize information:

Emoji	Section	Type	Meaning
📐	Appearance/Interface	INFORMATIVE	Visual properties or API structure
📦	Content/Structure	INFORMATIVE	Component composition
⚡	Capabilities/Methods	**NORMATIVE**	Behaviors that MUST be implemented
💡	Implementation Hint	INFORMATIVE	Suggested strategies (not mandatory)
🚨	Constraint	**NORMATIVE**	Hard limits that MUST be enforced
☑	Acceptance Criteria	**NORMATIVE**	Conditions for completion
🧪	Test Scenarios	**NORMATIVE**	Testable assertions

Required vs Optional Markers

markdown

```
(REQUIRED)  → Section MUST be present
(OPTIONAL)  → Section MAY be omitted, but if present MUST be implemented
```

RFC 2119 Keywords

ISL adopts RFC 2119 terminology for constraint specifications:

- **MUST** / **REQUIRED**: Absolute requirement
- **MUST NOT**: Absolute prohibition
- **SHOULD** / **RECOMMENDED**: Strong preference (may be ignored with justification)
- **SHOULD NOT**: Strong discouragement
- **MAY** / **OPTIONAL**: Truly optional

Usage in Constraints:

markdown

```
🚨 Constraint:
- Password MUST be hashed (absolute)
- Email SHOULD be verified (strong recommendation)
- Nickname MAY contain emojis (truly optional)
```

Type Notation

ISL uses TypeScript-like notation for types (adaptable to target language):

markdown

```
**Signature:**
- **input**: {email: string, password: string}
- **output**: {token: string, expiresAt: datetime} | {error: string}
```

Supported Notations:

- **Primitives**: `string`, `number`, `boolean`, `datetime`, `uuid`
- **Collections**: `string[]`, `Array<User>`
- **Objects**: `{field1: type1, field2: type2}`

- **Unions**: `Type1 | Type2`
- **Optional**: `field?: type`
- **Enums**: `'value1' | 'value2' | 'value3'`

Flow Notation

Flows use numbered steps with optional branching:

markdown

```
**Flow:**
1. Step description
2. Conditional step:
   IF condition THEN
     a. Action A
   ELSE
     b. Action B
3. FOR EACH item IN collection:
   a. Process item
4. Final step
```

Supported Control Structures:

- `IF...THEN...ELSE`
- `FOR EACH...IN`
- `WHILE condition`
- `TRY...CATCH`
- `BRANCH: [conditions]`

Section Reference

Project Header

markdown

```
# Project: [Name]
[Brief summary of the project (1-3 sentences)]
```

Purpose: Identifies the system being specified and provides context.

Required Fields:

- Project name
- Brief summary

Optional Metadata (recommended):

markdown

```
# Project: E-Commerce Platform

**Version**: 1.0.0
**ISL Version**: 1.4
**Target Stack**: React 18 + Node.js + PostgreSQL
**Created**: 2026-01-15
**Updated**: 2026-01-20
```

Domain Concepts

markdown

```
## Domain Concepts

### [Entity Name]
**Identity**: How it's uniquely identified (e.g., UUID, composite key)
**Properties**: Semantic attributes (NOT implementation fields)
**Relationships**: Connections to other entities
```

Purpose: Establishes shared vocabulary for the entire specification. Ensures consistent naming across components.

Guidelines:

- Use business domain terminology, not technical implementation
- Focus on "what it represents" not "how it's stored"
- Define relationships explicitly (1:1, 1:N, N:M)

Example:

markdown

```
## Domain Concepts

### User
**Identity**: UUID (universally unique identifier)
**Properties**:
- email: unique identifier for authentication
- displayName: user's chosen public name
- accountStatus: enum (active, suspended, deleted)
**Relationships**:
- Has many Orders (1:N)
- Belongs to one Organization (N:1)

### Order
**Identity**: UUID
**Properties**:
- orderNumber: human-readable reference
- totalAmount: monetary value
- orderStatus: enum (pending, confirmed, shipped, delivered, cancelled)
**Relationships**:
- Belongs to one User (N:1)
- Contains many OrderItems (1:N)
```

Component Declaration

markdown

```
## Component: [Component Name]
[Concise description of component's role (1-2 sentences)]

### Role: Presentation / Backend (REQUIRED)
```

Purpose: Defines a unit of implementation (UI component, service, API endpoint, etc.)

Role Values:

- `Presentation`: UI components, visual elements, user interactions
- `Backend`: Services, APIs, business logic, data access

Naming Convention:

- Use PascalCase for components: `LoginButton`, `UserService`
- Be specific: `UserProfileCard` better than `Card`
- Include role context if helpful: `UserAuthService`, `PaymentGateway`

Appearance / Interface

markdown

```
### 📐 Appearance / Interface (OPTIONAL)
```

For Presentation (Role: Presentation):

markdown

```
### 📐 Appearance
- Primary button styling
- Background: #3b82f6 (blue)
- Text color: white
- Border radius: 8px
- Padding: 12px 24px
- States:
  - Hover: background #2563eb (darker blue)
  - Active: background #1d4ed8
  - Disabled: background #9ca3af (gray), cursor not-allowed
```

For Backend (Role: Backend):

markdown

```
### 📐 Interface
- HTTP Method: POST
- Route: /api/auth/login
- Content-Type: application/json
- Request Schema: {email: string, password: string}
- Response Schema: {token: string, expiresAt: ISO8601} | {error: string, code: number}
- Headers Required: none
- Headers Optional: X-Client-Version
```

Guidelines:

- Presentation: Focus on visual properties, not implementation (e.g., "blue" not "`#3b82f6`" if exact color not critical)
- Backend: Define complete API contract (method, route, schemas)
- Use consistent terminology from Domain Concepts

Content / Structure

markdown

```
### 📦 Content / Structure (OPTIONAL)
```

For Presentation:

markdown

```
### 📦 Content
- Form container
  - EmailInput (child component)
  - PasswordInput (child component)
  - SubmitButton (child component)
  - ErrorMessage (conditional, shown on validation failure)
```

For Backend:

markdown

```
### 📦 Structure
**Dependencies**:
- UserRepository (data access)
- JWTService (token generation)
- EmailService (notifications)

**Class Members**:
- private userRepo: UserRepository
- private jwtService: JWTService
- private emailService: EmailService
```

Guidelines:

- Presentation: Describe DOM/component hierarchy
- Backend: List dependencies and class structure
- Keep implementation-agnostic (e.g., "token generation" not "JWT library v3.2")

Capabilities / Methods

markdown

```
### ⚡ Capabilities / Methods (OPTIONAL)

#### [Capability Name]
```

Purpose: Defines individual behaviors or methods. This is the core of ISL specifications.

Structure (all fields within a capability):

Signature (OPTIONAL but RECOMMENDED)

markdown

```
**Signature:**
- **input**: {param1: type, param2: type}
- **output**: {result: type} | {error: type}
```

When to include:

- Always for Backend methods
- For Presentation when data flow is non-trivial
- Omit for simple event handlers with no parameters

Contract (REQUIRED)

markdown

```
**Contract**: Clear statement of what this capability promises to do
```

Guidelines:

- Start with action verb: "Authenticate user", "Validate input", "Render chart"
- Be specific about outcome: "Generate JWT token valid for 24 hours"
- One sentence, clear and testable

Trigger (REQUIRED for Presentation, OPTIONAL for Backend)

markdown

```
**Trigger**: Event that activates this capability
```

Presentation Examples:

- `Click on submit button`
- `DoubleClick on node`
- `KeyPress: Enter key`
- `MouseDown + Drag on canvas`

Backend Examples:

- `HTTP POST to /api/users`
- `Scheduled cron job (daily at 2:00 AM)`
- `Message received on Kafka topic: user-events`

Flow (OPTIONAL - REQUIRED if multi-step)

markdown

```
**Flow:**
1. Step 1
2. Step 2
   IF condition THEN
     a. Branch A
   ELSE
     b. Branch B
3. Step 3
```

Guidelines:

- Use numbered steps for sequential operations
- Use IF/ELSE for conditional logic
- Use FOR EACH for iterations
- Keep steps at appropriate abstraction level (not too detailed)

When to include:

- Multi-step processes (login, checkout, data transformation)
- Complex algorithms
- Branching logic

When to omit:

- Single atomic operations ("toggle boolean", "return value")

Side Effect (OPTIONAL)

markdown

```
**Side Effect**: State changes or external interactions
```

Examples:

- `Updates user.lastLogin timestamp`
- `Sends email notification via EmailService`
- `Triggers re-render of parent component`
- `Writes to database: users table`

Cleanup (OPTIONAL)

markdown

```
**Cleanup**: Finalization actions
```

Examples:

- `Restore cursor to default`
- `Remove temporary DOM elements`
- `Close database connection`
- `Reset form fields`

Implementation Hint (OPTIONAL)

markdown

```
** 💡 Implementation Hint**: Suggested approach or code snippet
```

Purpose: Guides implementation without mandating specific code.

Examples:

markdown

```
💡 Implementation Hint:
- Use bcrypt with cost factor 12 for password hashing
- Consider debouncing search input (300ms delay)
- Prefer CSS Grid over Flexbox for this layout
```

With Code Snippet:

markdown

```
💡 Implementation Hint:
```javascript
// Suggested error handling pattern
try {
  const user = await authenticateUser(email, password);
  return {success: true, user};
} catch (error) {
  if (error instanceof AuthenticationError) {
    return {success: false, error: 'Invalid credentials'};
  }
  throw error; // Re-throw unexpected errors
}
```
```

```
**Guidelines**:

- Hints are SUGGESTIONS, not requirements (unless also in 🚨 Constraint)
- Include rationale: "Use bcrypt for OWASP compliance"
- Provide alternatives when appropriate

#### Constraint (OPTIONAL but RECOMMENDED)

markdown

```markdown
**🚨 Constraint**: Hard limits and rules (NORMATIVE)
```
```

Purpose: Defines what MUST or MUST NOT happen. These are non-negotiable.

Format:

markdown

```
🚨 Constraint:
- MUST [requirement]
- MUST NOT [prohibition]
- SHOULD [strong recommendation]
```

Examples:

markdown

```
🚨 Constraint:
- Password MUST be hashed before storage (never plaintext)
- Response time MUST be < 200ms (99th percentile)
- MUST NOT modify node position during resize operation
- MUST validate all inputs before processing
- UI MUST remain responsive during long operations (show loading state)
```

Guidelines:

- Use RFC 2119 keywords (MUST, MUST NOT, SHOULD)
- Be specific and measurable
- Separate security, performance, and functional constraints

Acceptance Criteria (OPTIONAL but RECOMMENDED)

markdown

```
**☑ Acceptance Criteria**: Conditions for completion
```

Purpose: Defines when this capability is considered "done".

Format:

markdown

```
☑ Acceptance Criteria:
- [ ] Criterion 1 (testable condition)
- [ ] Criterion 2
```

Examples:

markdown

```
☑ Acceptance Criteria:
- [ ] User can login with valid credentials
- [ ] Invalid credentials show error message
- [ ] Form is disabled during authentication
- [ ] Successful login redirects to dashboard
- [ ] Failed login increments attempt counter
```

Guidelines:

- Each criterion should be independently testable
- Use checkbox format for clarity
- Focus on observable behavior, not implementation

Test Scenarios (OPTIONAL but RECOMMENDED)

markdown

```
**🧪 Test Scenarios**: Specific test cases
```

Purpose: Provides concrete test cases that can be automated.

Format:

markdown

```
🧪 Test Scenarios:
1. **Scenario Name**: Input → Expected Output
2. **Edge Case Name**: Condition → Expected Behavior
```

Examples:

markdown

```
🧪 Test Scenarios:
1. **Valid Login**:
   - Input: {email: "user@test.com", password: "Valid123!"}
   - Expected: {token: <JWT>, expiresAt: <timestamp>}

2. **Invalid Password**:
   - Input: {email: "user@test.com", password: "wrong"}
   - Expected: {error: "Invalid credentials"}

3. **Account Locked**:
   - Precondition: 5 failed login attempts
   - Input: valid credentials
   - Expected: {error: "Account locked", code: 423}

4. **Concurrent Logins**:
   - Input: Same user logs in from 2 devices
   - Expected: Both sessions valid, old session NOT invalidated
```

Guidelines:

- Include happy path, error cases, and edge cases
- Specify preconditions when relevant
- Define expected behavior clearly
- Use realistic test data

Global Sections (Component-Level)

Global Implementation Hints

markdown

```
### 💡 Global Implementation Hints (OPTIONAL)
- [Architectural notes]
- [Recommended libraries]
- [Performance considerations]
```

Purpose: Provides component-level implementation guidance.

Examples:

markdown

```
### 💡 Global Implementation Hints
- Use React 18+ with TypeScript
- State management: Zustand (lightweight, < 1KB)
- Form validation: Zod schemas (type-safe)
- Consider memoization for expensive renders (useMemo, React.memo)
- Bundle size target: < 50KB gzipped
```

Global Constraints

markdown

```
### 🚨 Global Constraints (OPTIONAL)
- [Component-wide limitations]
```

Purpose: Constraints that apply to the entire component, not just one capability.

Examples:

markdown

```
### 🚨 Global Constraints
- Component MUST be accessible (WCAG 2.1 Level AA)
- All API calls MUST include authentication token
- Component MUST NOT directly access localStorage (use provided storage service)
- Maximum bundle size: 100KB
- Render time MUST be < 16ms (60fps)
```

Acceptance Criteria (Component-Level)

markdown

```
### ☑ Acceptance Criteria (OPTIONAL)
- [ ] Component-wide completion criteria
```

Purpose: Overall conditions for component completion (beyond individual capabilities).

Examples:

markdown

```
### ☑ Acceptance Criteria
- [ ] Component passes all accessibility audits
- [ ] Component has 80%+ test coverage
- [ ] Component works in Chrome, Firefox, Safari (latest versions)
- [ ] Component handles all error states gracefully
- [ ] Component documentation is complete
```

Test Scenarios (Component-Level)

markdown

```
### 🩹 Test Scenarios (OPTIONAL)
1. **Integration Scenario**: Complex multi-capability test
```

Purpose: End-to-end or integration tests that span multiple capabilities.

Examples:

markdown

```
### 🩹 Test Scenarios
1. **Complete Login Flow**:
   - User enters email
   - User enters password
   - User clicks submit
   - Loading state appears
   - Success: redirect to dashboard
   - Token stored securely

2. **Error Recovery**:
   - Network failure during login
   - User sees error message
   - User retries
   - Success on second attempt
```

Complete Examples

Example 1: Frontend Component (Presentation)

markdown

```
# Project: Task Management App

Simple task tracking application with drag-and-drop kanban board.

---

## Domain Concepts

### Task
**Identity**: UUID
**Properties**:
- title: short description of task
- status: enum (todo, in-progress, done)
- priority: enum (low, medium, high)
- assignee: reference to User
**Relationships**:
- Belongs to one Project (N:1)
- Assigned to zero or one User (N:0..1)

### User
**Identity**: UUID
**Properties**:
- email: unique, authentication identifier
- displayName: user's chosen name
**Relationships**:
- Assigned to many Tasks (1:N)

---

## Component: TaskCard
Displays a single task in the kanban board with drag capability.

### Role: Presentation

### 📐 Appearance
- Card layout: white background, rounded corners (8px)
- Border: 1px solid #e5e7eb (gray-200)
- Padding: 16px
- Shadow on hover: 0 4px 6px rgba(0,0,0,0.1)
- Dragging state: opacity 0.5, cursor grabbing
- Priority indicator:
  - Low: blue dot
  - Medium: yellow dot
  - High: red dot

### 📦 Content
- Priority indicator (colored dot, top-left)
- Task title (bold, 16px)
- Assignee avatar (if assigned, bottom-right)
- Status badge (hidden, inferred from column)
```

```
---

### ⚡ Capabilities

#### dragStart

**Signature:**
- **input**: NONE (user-initiated)
- **output**: NONE (side effect only)

**Contract**: Initiate drag operation to move task between columns

**Trigger**: MouseDown on card + MouseMove (drag gesture)

**Flow**:
1. Capture card initial position
2. Create ghost element (semi-transparent copy)
3. Hide original card (opacity 0.5)
4. Ghost element follows cursor
5. On MouseUp:
   IF dropped on valid column THEN
     a. Update task.status
     b. Move card to new column
   ELSE
     a. Animate card back to original position

**Side Effect**:
- Task.status updated in application state
- Parent KanbanBoard re-renders affected columns
- Cursor changes to 'grabbing'

**Cleanup**:
- Remove ghost element
- Restore card opacity to 1.0
- Cursor restored to default

** 💡 Implementation Hint**:
```javascript
// Use HTML5 Drag and Drop API
card.addEventListener('dragstart', (e) => {
  e.dataTransfer.effectAllowed = 'move';
  e.dataTransfer.setData('text/plain', task.id);
});
```

🚨 **Constraint**:

- MUST NOT allow drag if user lacks edit permission
- Drag operation MUST complete in < 100ms (perceived instant)
```

- MUST prevent text selection during drag

☑ **Acceptance Criteria**:

- [ ] Card is draggable with mouse
- [ ] Card is draggable with touch (mobile)
- [ ] Ghost element appears during drag
- [ ] Original card shows reduced opacity during drag
- [ ] Drop updates task status correctly

🩹 **Test Scenarios**:

1. **Successful Drag**:
    - User drags task from "Todo" to "In Progress"
    - Expected: task.status = "in-progress", card appears in new column
2. **Cancelled Drag**:
    - User drags task and releases outside valid drop zone
    - Expected: card animates back to original position, status unchanged
3. **Permission Denied**:
    - User without edit permission attempts drag
    - Expected: card does not move, cursor shows 'not-allowed'

---

editTitle

Signature:

- **input**: NONE (trigger shows inline editor)
- **output**: NONE (updates task.title)

Contract: Allow inline editing of task title

Trigger: DoubleClick on task title text

Flow:

1. Replace title text with input field
2. Input field pre-filled with current title

3. Input field focused automatically
4. User edits text
5. On Enter key: save changes
6. On Escape key: discard changes
7. On blur (click outside): save changes

Side Effect:

- Updates task.title in application state
- Input field replaces title text (temporarily)

Cleanup:

- Remove input field
- Restore title text display
- Re-enable card dragging (disabled during edit)

🚨 **Constraint**:

- Title MUST NOT be empty (min 1 character)
- Title MUST NOT exceed 200 characters
- MUST disable drag during edit mode
- Input field MUST prevent keyboard shortcuts from affecting card (e.g., Delete key shouldn't delete card)

☑ **Acceptance Criteria**:

- [ ] DoubleClick enters edit mode
- [ ] Enter saves changes
- [ ] Escape cancels changes
- [ ] Click outside saves changes
- [ ] Empty title rejected with error message

🧪 **Test Scenarios**:

1. **Edit and Save**:
    - DoubleClick title, type "New Title", press Enter
    - Expected: task.title = "New Title"
2. **Edit and Cancel**:
    - DoubleClick title, type "Changes", press Escape
    - Expected: task.title unchanged

3. **Empty Title Rejection**:
    - DoubleClick title, delete all text, press Enter
    - Expected: Error message shown, title unchanged

💡 Global Implementation Hints

- Use React 18+ functional component
- Props: {task: Task, onUpdate: (task: Task) => void, draggable: boolean}
- Consider using react-beautiful-dnd library for drag-and-drop
- Memoize component with React.memo (re-render only when task changes)

🚨 Global Constraints

- Component MUST be keyboard accessible (Tab navigation)
- Component MUST work on touch devices (mobile)
- Component MUST NOT make direct API calls (use provided onUpdate callback)

☑ Acceptance Criteria

- [ ] Component renders correctly with all task properties
- [ ] All capabilities functional on desktop and mobile
- [ ] Component accessible via keyboard
- [ ] No prop drilling (use context if needed)
- [ ] Component has 90%+ test coverage

Example 2: Backend Component

```
# Project: E-Commerce Platform

Online store with product catalog, shopping cart, and checkout.

---

## Domain Concepts

### User
**Identity**: UUID
**Properties**:
- email: unique, for authentication
- passwordHash: bcrypt hashed password
- role: enum (customer, admin)
**Relationships**:
- Has one Cart (1:1)
- Has many Orders (1:N)

### Cart
**Identity**: UUID
**Properties**:
- totalAmount: calculated sum of items
**Relationships**:
- Belongs to one User (1:1)
- Contains many CartItems (1:N)

### CartItem
**Identity**: UUID
**Properties**:
- quantity: positive integer
- priceAtAdd: price snapshot when added
**Relationships**:
- Belongs to one Cart (N:1)
- References one Product (N:1)

---

## Component: CartService
Manages shopping cart operations: add/remove items, calculate totals.

### Role: Backend

### 📐 Interface
- Service class (not HTTP endpoint)
- Used by CartController for HTTP API
- Methods return Promise<Result<T, Error>>

### 📦 Structure
**Dependencies**:
- CartRepository (database access)
```

```
- ProductRepository (product info)
- PricingService (price calculations, discounts)

**Class Members**:
```typescript
class CartService {
  constructor(
    private cartRepo: CartRepository,
    private productRepo: ProductRepository,
    private pricingService: PricingService
  ) {}
}
```

Methods

addItem

Signature:

```
- **input**: {
    userId: UUID,
    productId: UUID,
    quantity: number
  }
- **output**: {
    cart: Cart,
    item: CartItem
  } | {
    error: string,
    code: 'PRODUCT_NOT_FOUND' | 'INSUFFICIENT_STOCK' | 'INVALID_QUANTITY'
  }
```

Contract: Add product to user's cart with specified quantity, create cart if doesn't exist

Flow:

1. Validate inputs: IF quantity < 1 THEN RETURN error('INVALID_QUANTITY')
2. Fetch product: product = ProductRepository.findById(productId) IF product is null THEN RETURN error('PRODUCT_NOT_FOUND')
3. Check stock availability: IF product.stock < quantity THEN RETURN error('INSUFFICIENT_STOCK')
4. Get or create cart: cart = CartRepository.findByUserId(userId) IF cart is null THEN cart = CartRepository.create({userId})
```

5. Check if item already in cart: existingItem = cart.items.find(i => i.productId == productId) IF existingItem THEN a. Update quantity: existingItem.quantity += quantity b. Recalculate cart.totalAmount ELSE a. Create new CartItem:

   ```
   item = {productId, quantity, priceAtAdd: product.currentPrice}
   ```

   b. Add to cart.items c. Recalculate cart.totalAmount
6. Save cart: CartRepository.save(cart)
7. Return success: {cart, item}

Side Effect:

- Cart record created or updated in database
- CartItem record created or updated
- Product.reservedStock incremented (prevents overselling)

Cleanup: NONE

💡 **Implementation Hint**:

```
// Use transaction for atomic cart updates
async addItem(userId, productId, quantity) {
  return await db.transaction(async (trx) => {
    // All operations within transaction
    const product = await productRepo.findById(productId, {trx});
    // ... rest of logic
    await cartRepo.save(cart, {trx});
  });
}
```

🚨 **Constraint**:

- MUST use database transaction (all-or-nothing)
- MUST validate product stock availability BEFORE adding
- MUST NOT allow quantity < 1
- MUST store price snapshot (priceAtAdd) for historical accuracy
- Operation MUST complete in < 500ms

☑ **Acceptance Criteria**:

- [ ] Successfully adds new product to cart
- [ ] Successfully updates quantity if product already in cart

- [ ] Rejects invalid quantity (< 1)
- [ ] Rejects non-existent product
- [ ] Rejects if insufficient stock
- [ ] Creates cart if user doesn't have one
- [ ] Calculates totalAmount correctly

🩹 **Test Scenarios**:

1. **Add New Item**:
    - Input: {userId: "user-1", productId: "prod-1", quantity: 2}
    - Precondition: User has empty cart
    - Expected: {cart: {items: [item], totalAmount: 39.98}, item: {quantity: 2}}
2. **Update Existing Item**:
    - Input: {userId: "user-1", productId: "prod-1", quantity: 3}
    - Precondition: Cart already contains prod-1 with quantity 2
    - Expected: {cart: {items: [item], totalAmount: 99.95}, item: {quantity: 5}}
3. **Insufficient Stock**:
    - Input: {userId: "user-1", productId: "prod-1", quantity: 100}
    - Precondition: Product has stock of 10
    - Expected: {error: "Insufficient stock", code: "INSUFFICIENT_STOCK"}
4. **Invalid Quantity**:
    - Input: {userId: "user-1", productId: "prod-1", quantity: 0}
    - Expected: {error: "Quantity must be at least 1", code: "INVALID_QUANTITY"}

---

removeItem

Signature:

```
- **input**: {userId: UUID, itemId: UUID}
- **output**: {cart: Cart} | {error: string, code: 'ITEM_NOT_FOUND' |
'CART_NOT_FOUND'}
```

Contract: Remove item from cart and recalculate total

Trigger: User clicks remove button in cart UI

Flow:

1. Fetch cart: cart = CartRepository.findByUserId(userId) IF cart is null THEN RETURN error('CART_NOT_FOUND')
2. Find item in cart: item = cart.items.find(i => i.id == itemId) IF item is null THEN RETURN error('ITEM_NOT_FOUND')
3. Remove item: cart.items.remove(item) Recalculate cart.totalAmount
4. Release reserved stock: Product.reservedStock -= item.quantity
5. Save cart: CartRepository.save(cart)
6. Return success: {cart}

Side Effect:

- CartItem record deleted from database
- Cart.totalAmount updated
- Product.reservedStock decremented

💡 **Implementation Hint**:

```
// Consider soft delete for audit trail
async removeItem(userId, itemId) {
  const item = await cartItemRepo.findById(itemId);
  item.deletedAt = new Date(); // Soft delete
  await cartItemRepo.save(item);
}
```

🚨 **Constraint**:

- MUST use database transaction
- MUST release product reserved stock
- MUST NOT allow removing items from other users' carts (authorization check)

☑ **Acceptance Criteria**:

- [ ] Successfully removes item from cart
- [ ] Recalculates totalAmount correctly
- [ ] Returns error if item not found
- [ ] Returns error if cart not found
- [ ] Prevents removing items from another user's cart

🧪 **Test Scenarios**:

1. **Remove Item**:
    - Precondition: Cart has 2 items
    - Input: {userId: "user-1", itemId: "item-1"}
    - Expected: {cart: {items: [remaining item], totalAmount: }}
2. **Item Not Found**:
    - Input: {userId: "user-1", itemId: "non-existent"}
    - Expected: {error: "Item not found", code: "ITEM_NOT_FOUND"}

💡 Global Implementation Hints

- Use dependency injection for repositories
- Consider implementing Result<T, E> type for error handling (no exceptions)
- Use TypeScript for type safety
- Cache cart data in Redis (invalidate on update)

🚨 Global Constraints

- All database operations MUST use transactions
- Service MUST NOT directly access HTTP request/response
- Service MUST be stateless (no instance variables for request data)
- All methods MUST be unit testable (mockable dependencies)

☑ Acceptance Criteria

- [ ] All methods have comprehensive unit tests
- [ ] Service integrates correctly with CartController
- [ ] Database queries are optimized (< 50ms per operation)
- [ ] Service handles concurrent cart updates correctly (pessimistic locking)

````
### Example 3: API Endpoint (Backend with Interface)

```markdown
# Project: Social Media Platform

---

## Domain Concepts

### Post
**Identity**: UUID
**Properties**:
- content: text (max 500 characters)
- authorId: reference to User
- createdAt: timestamp
- likesCount: integer (denormalized)
**Relationships**:
- Authored by one User (N:1)
- Has many Comments (1:N)
- Has many Likes (1:N)

---

## Component: CreatePostEndpoint
HTTP API endpoint for creating new posts.

### Role: Backend

### 📐 Interface
- **Method**: POST
- **Route**: /api/posts
- **Authentication**: Required (Bearer token)
- **Content-Type**: application/json
- **Rate Limit**: 10 posts per minute per user

**Request Schema**:
```json
{
  "content": "string (1-500 characters)",
  "attachments": ["url"] (optional, max 4 images)
}
````

Response Schema (Success - 201):

```
{
  "post": {
    "id": "uuid",
    "content": "string",
    "authorId": "uuid",
    "createdAt": "ISO8601 timestamp",
    "likesCount": 0,
    "attachments": ["url"]
  }
}
```

Response Schema (Error - 4xx):

```
{
  "error": {
    "message": "string",
    "code": "VALIDATION_ERROR | RATE_LIMIT_EXCEEDED | UNAUTHORIZED",
    "field": "string (optional, for validation errors)"
  }
}
```

Methods

handleRequest

Signature:

```
- **input**: HTTPRequest {
    headers: {Authorization: "Bearer <token>"},
    body: {content: string, attachments?: string[]}
  }
- **output**: HTTPResponse {
    status: 201 | 400 | 401 | 429,
    body: {post: Post} | {error: ErrorDetails}
  }
```

Contract: Create new post after validating content and checking rate limits

Trigger: HTTP POST request to /api/posts

Flow:

1. Authenticate request: token = extractBearerToken(request.headers.Authorization) user = AuthService.validateToken(token) IF user is null THEN RETURN 401 {error: {message: "Unauthorized", code: "UNAUTHORIZED"}}

2. Validate request body: content = request.body.content IF content is empty OR content.length > 500 THEN RETURN 400 {error: {message: "Content must be 1-500 characters", code: "VALIDATION_ERROR", field: "content"}}

   attachments = request.body.attachments || [] IF attachments.length > 4 THEN RETURN 400 {error: {message: "Maximum 4 attachments", code: "VALIDATION_ERROR", field: "attachments"}}

3. Check rate limit: isAllowed = RateLimiter.check(userId, "create_post", limit: 10, window: "1 minute") IF NOT isAllowed THEN RETURN 429 {error: {message: "Rate limit exceeded. Try again in 1 minute", code: "RATE_LIMIT_EXCEEDED"}}

4. Create post: post = PostService.create({ content: content, authorId: user.id, attachments: attachments })

5. Return success: RETURN 201 {post: post}

Side Effect:

- Post record created in database
- User's post count incremented
- Rate limiter counter incremented
- Followers notified (async job queued)

💡 **Implementation Hint**:

```
// Use Express.js middleware for auth and validation
router.post('/api/posts',
  authenticateToken,
  validateRequestBody(createPostSchema),
  checkRateLimit,
  async (req, res) => {
    const post = await postService.create(req.body, req.user.id);
    res.status(201).json({post});
  }
);
```

🚨 **Constraint**:

- MUST authenticate before processing

- MUST validate all inputs (never trust client)
- MUST enforce rate limits (prevent spam)
- MUST sanitize content (prevent XSS)
- MUST return appropriate HTTP status codes
- Response time MUST be < 300ms (p95)

☑ **Acceptance Criteria**:

- [ ] Creates post with valid authenticated request
- [ ] Returns 401 for unauthenticated requests
- [ ] Returns 400 for invalid content
- [ ] Returns 429 when rate limit exceeded
- [ ] Sanitizes content to prevent XSS
- [ ] Returns correct HTTP status codes

🧪 **Test Scenarios**:

1. **Successful Post Creation**:
    - Input: POST /api/posts, Auth: valid token, Body: {content: "Hello world"}
    - Expected: 201, {post: {id: uuid, content: "Hello world", ...}}
2. **Unauthorized Request**:
    - Input: POST /api/posts, Auth: missing or invalid, Body: {content: "Hello"}
    - Expected: 401, {error: {message: "Unauthorized", code: "UNAUTHORIZED"}}
3. **Content Too Long**:
    - Input: POST /api/posts, Auth: valid, Body: {content: <501 characters>}
    - Expected: 400, {error: {message: "Content must be 1-500 characters", code: "VALIDATION_ERROR", field: "content"}}
4. **Rate Limit Exceeded**:
    - Precondition: User has made 10 posts in last minute
    - Input: POST /api/posts, Auth: valid, Body: {content: "Another post"}
    - Expected: 429, {error: {message: "Rate limit exceeded...", code: "RATE_LIMIT_EXCEEDED"}}

Global Implementation Hints

- Use Express.js or Fastify for HTTP server
- Apply helmet middleware for security headers
- Use express-validator for request validation
- Consider implementing request/response logging middleware

Global Constraints

- MUST use HTTPS in production (no plain HTTP)
- MUST include CORS headers for web clients
- MUST log all requests (for auditing)
- MUST implement request timeouts (30s max)
- API MUST be versioned (/api/v1/posts)

Acceptance Criteria

- [ ] Endpoint passes security audit (OWASP Top 10)
- [ ] Endpoint has 95%+ test coverage
- [ ] Endpoint meets performance requirements (p95 < 300ms)
- [ ] Endpoint properly handles concurrent requests

```
---

## Best Practices

### 1. Writing Effective Contracts

**DO:**

- Start with action verb: "Authenticate user", "Calculate total"
- Be specific about outcome: "Generate JWT token valid for 24 hours"
- Make it testable: contract should directly translate to assertion

**DON'T:**

- Be vague: "Handle user login" (handle how?)
- Mix contract with implementation: "Use bcrypt to hash password" (that's a hint, not
contract)
- Make it too high-level: "Make the user happy" (not testable)

**Examples:**

```markdown
❌ BAD:
**Contract**: Process the form

☑ GOOD:
**Contract**: Validate form inputs and save user profile data to database
```

---

2. Defining Precise Constraints

DO:

- Use RFC 2119 keywords (MUST, SHOULD, MAY)
- Be measurable: "< 200ms" not "fast"
- Separate concerns: security, performance, functional

DON'T:

- Use ambiguous terms: "should be quick", "reasonably secure"
- Conflate hints with constraints: "Consider using Redis" is a hint, not constraint

Examples:
```

```
❌ BAD:
🚨 Constraint: The API should be fast and secure

☑ GOOD:
🚨 Constraint:
- Response time MUST be < 200ms (p95)
- MUST use TLS 1.3 or higher
- MUST validate all inputs before processing
- MUST NOT log sensitive data (passwords, tokens)
```

---

3. Scoping Flows Appropriately

When to include Flow:

- Multi-step processes (login, checkout, complex calculations)
- Branching logic (IF/ELSE conditions)
- Error handling paths
- Integration with multiple services

When to omit Flow:

- Simple atomic operations: "Toggle dark mode" → just flip boolean
- Single function call: "Get user by ID" → straightforward lookup
- Self-explanatory contracts: "Validate email format"

Flow Abstraction Level:

```
❌ TOO DETAILED (implementation):
Flow:
1. Create bcrypt instance with cost 12
2. Call bcrypt.hash(password, salt)
3. Store result in user.passwordHash field

☑ GOOD (behavioral):
Flow:
1. Hash password using bcrypt (cost factor 12)
2. Store hash in database
3. Return success confirmation
```

---

4. Balancing Hints vs Constraints

Implementation Hint (💡):

- Suggestions, best practices
- Can be ignored if alternative achieves same contract
- "Consider using...", "Recommended approach..."

Constraint (🚨):

- Mandatory requirements
- Cannot be violated
- "MUST", "MUST NOT"

Examples:

```
💡 Implementation Hint:
- Consider using Redis for session storage (fast, scales well)
- Alternative: Memcached or in-memory store acceptable if performance requirements met

🚨 Constraint:
- Session data MUST persist across server restarts
- Session tokens MUST expire after 24 hours
```

Rule of Thumb:

- If violation breaks the system → Constraint
- If violation is suboptimal but workable → Hint

---

5. Writing Testable Acceptance Criteria

DO:

- Make each criterion independently verifiable
- Use observable behavior: "User sees error message"
- Include both happy path and error cases
- Be specific: "< 200ms" not "fast"

DON'T:

- Test implementation details: "Function uses bcrypt" (test contract instead)
- Make criteria too broad: "Everything works" (not actionable)
- Duplicate information already in constraints

Examples:

```
❌ BAD:
☑ Acceptance Criteria:
- [ ] Code is written correctly
- [ ] No bugs
- [ ] Works well

☑ GOOD:
☑ Acceptance Criteria:
- [ ] User can submit form with valid inputs
- [ ] Invalid email shows error: "Please enter valid email"
- [ ] Form submission completes in < 500ms
- [ ] Success shows confirmation: "Profile updated"
- [ ] Network failure shows retry option
```

---

6. Defining Realistic Test Scenarios

Components of Good Test Scenario:

1. **Clear preconditions**: State of system before test
2. **Specific inputs**: Exact data used in test
3. **Expected outcomes**: Precise results (not "should work")
4. **Edge cases**: Boundary conditions, errors, race conditions

Examples:

```
❌ BAD:
🩹 Test: User logs in and it works

☑ GOOD:
🩹 Test Scenarios:
1. **Successful Login**:
   - Precondition: User exists with email "test@example.com"
   - Input: {email: "test@example.com", password: "Valid123!"}
   - Expected: {token: <JWT>, expiresAt: <now + 24h>}

2. **Account Locked After 5 Failed Attempts**:
   - Precondition: User has 4 failed login attempts in last 10 minutes
   - Input: {email: "test@example.com", password: "wrong"}
   - Expected: {error: "Account locked for 10 minutes", code: "ACCOUNT_LOCKED"}

3. **Concurrent Login Attempts**:
   - Precondition: User credentials are valid
   - Input: 2 simultaneous login requests from different IPs
   - Expected: Both succeed, each gets unique session token
```

---

7. Domain Concepts Best Practices

DO:

- Use business terminology (User, Order, Payment)
- Define relationships clearly (1:1, 1:N, N:M)
- Keep properties semantic (what it represents, not how it's stored)
- Include enums for constrained values

DON'T:

- Use technical terms (UserEntity, OrderDTO, PaymentRecord)
- Include implementation details (database table names, column types)
- Mix data model with UI concerns

Examples:

```
❌ BAD:
### UserEntity
**Properties**:
- user_id: integer (primary key)
- email_address: varchar(255)
- password_bcrypt: varchar(60)
- created_timestamp: datetime

☑ GOOD:
### User
**Identity**: UUID (universally unique identifier)
**Properties**:
- email: unique authentication identifier
- displayName: user's chosen public name
- accountStatus: enum (active, suspended, deleted)
- registrationDate: timestamp of account creation
**Relationships**:
- Has many Orders (1:N)
```

---

8. Component Naming Conventions

Frontend (Presentation):

- Components: `UserProfileCard`, `LoginButton`, `TaskList`
- Capabilities: `onClick`, `onHover`, `dragStart`, `editTitle`
- Use action verbs for capabilities: `select`, `move`, `resize`

Backend:

- Services: `AuthenticationService`, `PaymentService`, `EmailService`
- Endpoints: `CreatePostEndpoint`, `GetUserEndpoint`
- Methods: `authenticateUser`, `processPayment`, `sendEmail`
- Use business domain verbs: `authenticate`, `authorize`, `validate`

General Rules:

- PascalCase for component names
- camelCase for capabilities/methods
- Be specific: `UserProfileCard` not `Card`
- Avoid abbreviations: `Authentication` not `Auth` (unless domain standard)

---

9. Versioning ISL Documents

Include Version Metadata:

```
# Project: TaskManager

**Version**: 2.1.0
**ISL Version**: 1.4
**Last Updated**: 2026-01-20
**Breaking Changes**: None since 2.0.0
```

Version Numbering:

- **Major** (X.0.0): Breaking changes to Domain Concepts or component contracts
- **Minor** (0.X.0): New components or capabilities (backwards compatible)
- **Patch** (0.0.X): Clarifications, constraint additions, test improvements

Document Changes:

```
## Changelog

### v2.1.0 (2026-01-20)
- Added Component: NotificationService
- Enhanced TaskCard: added priority indicator
- Clarified constraint on UserService.deleteAccount

### v2.0.0 (2026-01-10) - BREAKING
- Changed User.role from string to enum
- Renamed Component: TaskManager → TaskService
```

---

10. Common Pitfalls to Avoid

Pitfall 1: Mixing Levels of Abstraction

```
❌ BAD (mixes intent with implementation):
**Contract**: Hash password with bcrypt cost 12 and store in PostgreSQL users table

☑ GOOD (separates layers):
**Contract**: Securely store user password
🚨 Constraint: Password MUST be hashed (never plaintext)
💡 Implementation Hint: Use bcrypt with cost factor 12
```

Pitfall 2: Over-Specifying Visual Details

```
❌ BAD (too specific for intent spec):
📐 Appearance:
- Font: Inter, 14px, weight 500, line-height 1.5, letter-spacing -0.01em
- Margin: 16px top, 12px right, 8px bottom, 12px left
- Box shadow: 0 1px 3px 0 rgba(0, 0, 0, 0.1), 0 1px 2px 0 rgba(0, 0, 0, 0.06)

☑ GOOD (intent-level):
📐 Appearance:
- Typography: Clear, readable body text
- Spacing: Comfortable padding and margins
- Elevation: Subtle shadow for card effect
💡 Implementation Hint: Consider using Tailwind classes: text-sm font-medium shadow-sm
p-4
```

Pitfall 3: Unclear Constraint Scope

```
❌ BAD (ambiguous):
🚨 Constraint: Must be fast

☑ GOOD (measurable, scoped):
🚨 Constraint:
- Initial page load MUST complete in < 2 seconds (p95)
- API response time MUST be < 200ms (p95)
- UI MUST remain responsive during data loading (show loading state)
```

Pitfall 4: Test Scenarios Without Expected Outcomes

```
❌ BAD:
🩹 Test: User clicks login button

☑ GOOD:
🩹 Test Scenarios:
1. **Login with Valid Credentials**:
   - Input: {email: "valid@test.com", password: "correct"}
   - Expected: Redirect to /dashboard, token stored in auth context
2. **Login with Invalid Credentials**:
   - Input: {email: "valid@test.com", password: "wrong"}
   - Expected: Error message "Invalid email or password", stay on login page
```

Pitfall 5: Forgetting Role Separation

```
❌ BAD (business logic in Presentation):
Component: PriceDisplay
Role: Presentation
⚡ calculateDiscount
  Flow:
    1. IF user.isPremium THEN discount = 0.20
    2. ELSE discount = 0.10
    3. finalPrice = originalPrice * (1 - discount)
    → Violates Rule 4 (business logic in Presentation)

☑ GOOD:
Component: PriceDisplay
Role: Presentation
⚡ displayPrice
  **Contract**: Render formatted price with discount applied
  **Input**: {originalPrice: number, finalPrice: number}
  **Flow**:
    1. Display originalPrice (strikethrough if discount applied)
    2. Display finalPrice (prominent)
    3. Show discount percentage if applicable
  → Calculation delegated to Backend (PricingService)
```

FAQ

General Questions

Q: What is ISL and who should use it?

A: ISL (Intent Specification Language) is a structured Markdown format for writing software specifications that LLMs can deterministically convert into code. It's designed for:

- Software architects defining system components
- Developers documenting APIs and UI components
- Teams using AI-assisted development tools
- Anyone who wants specifications that are both human-readable and machine-executable

---

Q: How is ISL different from traditional documentation?

A: Traditional docs describe what exists; ISL prescribes what should be built. Key differences:

- **Contracts**: Explicit input/output signatures (like API specs, but for all components)

- **Constraints**: Normative rules that code generators must follow
- **Test Integration**: Acceptance criteria and test scenarios built into spec
- **Determinism**: Canonical rules ensure consistent LLM interpretation

---

Q: Can ISL be used without LLMs?

A: Yes! ISL specifications are valuable for human developers as:

- Detailed design documents with clear contracts
- Test plan templates (acceptance criteria → test cases)
- API documentation with examples
- Shared vocabulary (Domain Concepts) for team communication

However, ISL's true power emerges when paired with LLM code generators.

Q: What languages/frameworks does ISL support?

A: ISL is **language-agnostic**. The same specification can generate:

- Frontend: React, Vue, Svelte, Angular
- Backend: Node.js, Python, Java, Go, Rust
- Mobile: React Native, Flutter, Swift

The spec describes *behavior*, not *implementation*. Implementation Hints can suggest preferred tech stack.

Writing ISL Specs

Q: How much detail should I include?

A: Follow the "Goldilocks Principle":

- **Too little**: Vague contracts → non-deterministic output
- **Too much**: Implementation details → inflexible, harder to maintain
- **Just right**: Clear contracts, measurable constraints, suggested strategies

Rule of thumb: Include enough for someone (or an LLM) to implement it correctly on first try, without dictating *how* to implement.

Q: When should I use Flow vs just Contract?

A: Use **Flow** when:

- Multi-step process (login, checkout, wizard)
- Branching logic (IF/ELSE conditions)
- Integration with multiple services
- Complex algorithm or transformation

Omit **Flow** when:

- Single atomic operation ("toggle boolean", "return value")
- Contract is self-explanatory ("Validate email format")
- Implementation is trivial

Q: How do I handle optional features?

A: Three approaches:

1. **Optional Section**: Mark entire capability as future enhancement

```
⚡ Capabilities
#### exportPDF (OPTIONAL)
**Contract**: Generate PDF report
→ May be omitted in MVP, but if implemented must follow this spec
```

2. **Conditional Constraint**: Use SHOULD instead of MUST

```
🚨 Constraint:
- MUST validate email format
- SHOULD send confirmation email (if EmailService available)
```

Q: Can I reference other ISL documents?

A: Yes, using explicit references:

```
## Domain Concepts
(see: `domain-model.isl.md`)

OR

### Product
**Identity**: UUID
**Properties**: (as defined in ProductCatalog component spec)
**Relationships**:
- Referenced by CartItem (see: CartService specification)
```

For large projects, consider:

- One ISL file per subsystem
- Shared `domain-concepts.isl.md` imported by all
- Index file listing all components

Canonical Rules

Q: What happens if my specification violates a Canonical Rule?

A: Depends on the LLM implementation:

- **Compliant LLM**: Should report error and request clarification
- **Best-effort LLM**: May generate code but warn about violation
- **Non-compliant LLM**: May silently ignore (not recommended)

Example violations:

```
❌ VIOLATION (Rule 4):
Component: UserButton
Role: Presentation
⚡ validateUserPermissions
  Flow: Check database for user.role
  → Presentation accessing database directly

→ Compliant LLM should reject or request clarification
```

Q: Can I extend ISL with custom sections?

A: Yes! ISL is designed for extensibility. You can add:

```
### 🔒 Security Considerations (CUSTOM)
- Authentication required: Yes
- Authorization: Admin role only
- Data encryption: At rest and in transit

### 📊 Analytics Tracking (CUSTOM)
- Track event: "user_login"
- Properties: {method: "email" | "oauth", provider: string}
```

Guidelines for custom sections:

- Use unique emoji to differentiate from standard ISL
- Document meaning in project README
- Keep custom sections (OPTIONAL) to maintain spec portability

---

Q: How do I handle breaking changes in Canonical Rules?

A: Canonical Rules are versioned independently from ISL template:

```
# ISL Canonical Rules v1.4
(rules defined here)

# ISL Canonical Rules v2.0 (hypothetical future)
## Breaking Changes from v1.4:
- Rule 4 now requires explicit data flow diagrams for Backend components
```

When rules change:

1. Documents specify which Canonical Rules version they follow
2. LLMs must support multiple rule versions
3. Migration guides provided for upgrading specs

LLM Code Generation

Q: How do I ensure deterministic code generation?

A: Follow these practices:

1. **Explicit Domain Concepts**: Define all entity names, types
2. **Complete Contracts**: Include input/output signatures

3. **Measurable Constraints**: Use numbers, not adjectives ("< 200ms" not "fast")
4. **Comprehensive Test Scenarios**: Cover happy path, errors, edge cases
5. **Clear Implementation Hints**: Suggest specific libraries/patterns

Verification test:

```
# Generate code 10 times
for i in {1..10}; do
  llm generate spec.isl.md > output_$i.js
done

# Compare outputs (should be semantically identical)
diff output_1.js output_2.js
```

Q: What if the LLM generates code that violates constraints?

A: This indicates either:

1. **Specification ambiguity**: Constraint not clear enough
2. **LLM non-compliance**: LLM not following Canonical Rules
3. **Constraint impossibility**: Technical limitation

Solutions:

- Refine constraint with more specificity
- Add Implementation Hint showing how to satisfy constraint
- If constraint is impossible, relax it (MUST → SHOULD)

Example:

```
❌ VAGUE CONSTRAINT:
🚨 Constraint: Must be secure

☑ SPECIFIC CONSTRAINT:
🚨 Constraint:
- Passwords MUST be hashed with bcrypt (cost factor ≥ 12)
- Session tokens MUST be cryptographically random (≥ 256 bits entropy)
- API keys MUST NOT be logged or exposed in error messages

💡 Implementation Hint:
```javascript
// Use crypto.randomBytes for tokens
const token = crypto.randomBytes(32).toString('hex'); // 256 bits
```

Q: Can ISL specifications be version-controlled?

A: Absolutely! ISL documents are plain Markdown, perfect for Git:

```bash
# Example repository structure
/specs
  /domain
    domain-concepts.isl.md
  /frontend
    user-profile-card.isl.md
    login-form.isl.md
  /backend
    auth-service.isl.md
    payment-service.isl.md
  README.md (ISL version, conventions)
```
```

Benefits:

- Track specification evolution over time
- Review specification changes via pull requests
- Link commits to issues/tickets
- Generate diffs to see exactly what changed

Integration & Tooling

Q: Are there tools to validate ISL specifications?

A: As of ISL v1.4, reference tools are in development. Planned tools:

1. **ISL Linter**: Check structure, required sections
2. **Canonical Rules Validator**: Verify compliance with rules
3. **Contract Checker**: Ensure input/output types are consistent
4. **Test Generator**: Extract test scenarios into test files

Community contributions welcome!

Q: How do I integrate ISL with my development workflow?

A: Common integration patterns:

Pattern 1: Spec-First Development

Write ISL specification Generate initial code with LLM Implement tests based on Acceptance Criteria Refine code to pass tests Update ISL spec if requirements change

Pattern 2: Documentation-Driven

Write ISL spec as design document Team reviews and approves spec Developers implement manually (using spec as guide) Tests derived from Test Scenarios Spec serves as living documentation

Pattern 3: Hybrid

Write ISL spec Generate boilerplate/scaffolding with LLM Implement business logic manually LLM generates tests from spec CI/CD runs tests on every commit

---

Q: Can ISL be used for legacy code documentation?

A: Yes! Reverse engineering process:

1. **Analyze existing code** → identify components and their behaviors
2. **Extract contracts** → what does each function/component actually do?
3. **Document constraints** → what rules does code enforce?
4. **Write test scenarios** → based on existing tests or behavior
5. **Refine spec** → clarify ambiguities, add missing details

Benefits:

- Creates maintainable documentation for legacy systems
- Enables gradual refactoring (spec defines target state)
- Facilitates knowledge transfer to new team members

Advanced Usage

Q: How do I specify performance requirements?

A: Use measurable constraints with percentiles:

```
🚨 Constraint:
**Performance:**
- Response time MUST be < 200ms (p95)
- Response time MUST be < 100ms (p50)
- Throughput MUST support 1000 requests/second
- Memory usage MUST NOT exceed 512MB (p99)

**Scalability:**
- System MUST handle 10,000 concurrent users
- Database queries MUST complete in < 50ms (p95)

💡 Implementation Hint:
- Use connection pooling (max 100 connections)
- Implement query result caching (TTL: 5 minutes)
- Consider horizontal scaling for > 5000 users
```

Q: How do I handle multi-language/i18n requirements?

A: Specify at component and capability level:

```
Component: UserGreeting
Role: Presentation

🚨 Global Constraints:
- Component MUST support i18n (internationalization)
- All user-facing text MUST be translatable
- Supported languages: en, es, fr, de, ja

⚡ displayGreeting

**Contract**: Show personalized greeting in user's language

**Flow**:
1. Get user's preferred language (user.locale)
2. Load translation string for "greeting"
3. Interpolate user's name into translated template
4. Render greeting

💡 Implementation Hint:
- Use react-i18next or similar library
- Translation keys: "greeting.hello" → "Hello, {{name}}!"
- Fallback to English if translation missing

🧪 Test Scenarios:
1. **English User**: locale="en" → "Hello, John!"
2. **Spanish User**: locale="es" → "¡Hola, John!"
3. **Unsupported Locale**: locale="pt" → fallback to "Hello, John!"
```

Q: Can ISL specify database schemas?

A: Yes, using Backend components:

```
Component: UserTable
Role: Backend (Data)

### 📐 Interface
**Table Name**: users
**Primary Key**: id (UUID)

**Schema**:
| Column | Type | Constraints |
|--------|------|-------------|
| id | UUID | PRIMARY KEY, DEFAULT gen_random_uuid() |
| email | VARCHAR(255) | UNIQUE, NOT NULL |
| password_hash | VARCHAR(60) | NOT NULL |
| created_at | TIMESTAMP | NOT NULL, DEFAULT NOW() |
| last_login | TIMESTAMP | NULL |

**Indexes**:
- `idx_users_email` ON email (UNIQUE)
- `idx_users_created_at` ON created_at

🚨 Constraints:
- Email MUST be unique (enforced at database level)
- password_hash MUST be bcrypt hash (enforced at application level)

💡 Implementation Hint:
- Use PostgreSQL 14+ (for gen_random_uuid())
- Consider partitioning by created_at if > 10M rows
```

Changelog

v1.4 (2026-01-15) - Current

Added:

- Canonical Rules section (Rule 1-5)
- Signature subsection in Capabilities (input/output types)
- Rule 6: Error Handling (supplementary)
- Complete Examples section (3 detailed examples)
- Best Practices section (10 topics)
- FAQ section (20+ questions)

Changed:

- Reorganized document structure for better navigation
- Enhanced Notation & Conventions with type notation
- Expanded Section Reference with detailed guidelines
- Clarified OPTIONAL semantics in Rule 5

Improved:

- Determinism requirements (Rule 3) now more explicit
- Role separation rules (Rule 4) with concrete examples
- Test Scenarios examples now include preconditions

---

v1.3 (2026-01-10)

Added:

- Input/Output signatures in Capabilities
- Global sections (Implementation Hints, Constraints, Acceptance, Tests)

Changed:

- Renamed "Actions" to "Capabilities / Methods" for clarity
- Split "Style" into "Appearance / Interface" (role-dependent)

---

v1.2 (2025-12-20)

Added:

- Acceptance Criteria subsection
- Test Scenarios subsection
- Side Effect and Cleanup fields in capabilities

---

v1.1 (2025-12-01)

Added:

- Domain Concepts section
- Role field (Presentation / Backend)
- Implementation Hint subsection

Changed:

- Formalized emoji semantics (📐 📦 ⚡ 🚨 ☑ 💡 🔧)

---

v1.0 (2025-11-15) - Initial Release

Initial Features:

- Basic component structure
- Capabilities with Contract and Trigger
- Markdown format with emoji categories

Contributing

ISL is an open specification. Contributions are welcome!

How to Contribute

1. **Propose Changes**: Open an issue describing the enhancement
2. **Draft Specification**: Submit pull request with changes
3. **Community Review**: Gather feedback from users
4. **Approval**: Maintainers review and approve/reject
5. **Versioning**: Breaking changes → major version, additions → minor version

Contribution Guidelines

- **Backwards Compatibility**: Prefer additive changes over breaking changes
- **Clarity**: Specifications should be unambiguous
- **Examples**: Include examples demonstrating the change
- **Tooling Impact**: Consider impact on validators, generators, linters

Areas for Contribution

- **Language Bindings**: Tools to validate ISL in various languages
- **Examples**: Real-world ISL specifications for common patterns
- **Generators**: LLM plugins to generate code from ISL
- **Validators**: Tools to check ISL compliance and Canonical Rules

- **Extensions**: Domain-specific ISL extensions (e.g., ISL-ML for machine learning)

License

[To be determined - suggest MIT or Apache 2.0 for open adoption]

Contact & Resources

- **Repository**: [To be published]
- **Discussion Forum**: [To be created]
- **Issue Tracker**: [To be created]
- **Examples Gallery**: [To be published]

Appendix A: Quick Reference Card

```
# ISL Quick Reference

## Document Structure
# Project: Name
## Domain Concepts
### Entity
## Component: Name
### Role: Presentation | Backend
### 📐 Appearance / Interface
### 📦 Content / Structure
### ⚡ Capabilities / Methods
#### CapabilityName
### 💡 Global Implementation Hints
### 🚨 Global Constraints
### ☑ Acceptance Criteria
### 🧪 Test Scenarios

## Capability Structure
#### CapabilityName
**Signature:**
  - input: {params}
  - output: {result} | {error}
**Contract**: Promise
**Trigger**: Event
**Flow**: Steps
**Side Effect**: Mutations
**Cleanup**: Finalization
💡 **Implementation Hint**: Guidance
🚨 **Constraint**: Rules
☑ **Acceptance Criteria**: Tests
🧪 **Test Scenarios**: Cases

## Canonical Rules
1. ⚡🚨☑🧪 = NORMATIVE
2. Capability > Global > Contract > Hint
3. Same ISL → Same Code
4. Presentation ≠ Business Logic
5. OPTIONAL = may omit, not ignore
```

Appendix B: ISL Template

```
# Project: [Your Project Name]

[Brief project description]

**Version**: 1.0.0
**ISL Version**: 1.4
**Created**: YYYY-MM-DD

---

## Domain Concepts

### [EntityName]
**Identity**: How uniquely identified
**Properties**:
- property1: description
- property2: description
**Relationships**:
- Relationship to Entity (cardinality)

---

## Component: [ComponentName]

[Component description]

### Role: Presentation | Backend

### 📐 Appearance / Interface (OPTIONAL)

- [Presentation: CSS, colors, layout]
- [Backend: Routes, schemas, headers]

### 📦 Content / Structure (OPTIONAL)

- [Presentation: DOM structure]
- [Backend: Dependencies, class members]

---

### ⚡ Capabilities / Methods (OPTIONAL)

#### [CapabilityName]

**Signature:** (OPTIONAL)
- **input**: {param: type}
- **output**: {result: type} | {error: type}

**Contract**: What this capability promises
```

```
**Trigger**: Activation event

**Flow**: (REQUIRED if multi-step)
1. Step 1
2. Step 2

**Side Effect**: State mutations

**Cleanup**: Finalization actions

** 💡 Implementation Hint**: Guidance

** 🚨 Constraint**: Rules

**☑ Acceptance Criteria**:
- [ ] Criterion 1
- [ ] Criterion 2

** 🧪 Test Scenarios**:
1. **Scenario**: Input → Expected

---

### 💡 Global Implementation Hints (OPTIONAL)

- Architectural notes
- Recommended libraries

### 🚨 Global Constraints (OPTIONAL)

- Component-wide rules

### ☑ Acceptance Criteria (OPTIONAL)

- [ ] Component criterion 1
- [ ] Component criterion 2

### 🧪 Test Scenarios (OPTIONAL)

1. **Integration Scenario**: Description
```

END OF SPECIFICATION

This document is the official specification for Intent Specification Language (ISL) version 1.4. For questions, contributions, or support, please visit [repository/forum URL].