DRAFT — UNOFFICIAL — NOT FOR OPERATIONAL USE

ARCHITECTURE REFERENCE

ODT-CDA

USAREUR-AF

Bad: Task type determines existence

Architecture Reference

HEADQUARTERS
UNITED STATES ARMY EUROPE AND AFRICA
(USAREUR-AF)
Wiesbaden, Germany

DRAFT — NOT FOR OFFICIAL USE. FOR TRAINING PLANNING PURPOSES ONLY.

20 MARCH 2026

DRAFT — UNOFFICIAL — NOT FOR OPERATIONAL USE

sidebar_position: 1 title: "Identity vs Classification"

DATA MODELING FUNDAMENTALS: IDENTITY VS. CLASSIFICATION

A data engineer's perspective for the analytically optimistic

Fundamental Separation Between Identity and Classification

identity is "what something *is*," and **classification** is "what bucket we currently put it in." This isn't philosophy—it's the difference between systems that scale and systems that collapse under their own technical debt. Tasks are tasks. Study issues are study issues. Countries are countries. The **specified/flytrap/NATO part lives on the relationship**, not on the entity type.

Identity = intrinsic, stable properties that define an entity's existence - A `Task` is a task: "Secure Route TAMPA from CP 1 to CP 7" - A `StudyIssue` is a study issue: "Munitions expenditure rate exceeds production capacity" - A `Country` is a country: Finland, Sweden, Poland - A `Unit` is a unit: 1st Battalion, 75th Ranger Regiment

Classification = contextual, changeable attributes describing current or historical state - `TaskType` in the context of an order: specified vs. implied vs. essential - `IssueStatus` in the context of a study: open, deferred, resolved - `AllianceMembership`: NATO, EU, Quad, AUKUS - `ReadinessLevel`: C1, C2, C3, C4 - 'Studyissue': [studies] When you collapse classification into identity (like `SpecifiedTask`, `FlyTrapStudyIssue`, `NATOCountry` as types), you're embedding temporal and relational context into the entity's fundamental structure. That's like naming database tables `January2024SalesData` and `February2024SalesData` instead of having one `Sales` table with a date column—sounds convenient for one query, becomes unmaintainable immediately.

Example 1: Tasks Are Tasks

What happens: Your battalion S3 analyzes the OPORD and realizes that "Secure Route TAMPA" was initially a specified task, but after the commander's guidance, it's now an essential task. Now you need to: 1. **Delete** from `specified_tasks` (breaks any foreign keys) 2. **Insert** into `essential_tasks` (new primary key? same key? resurrection problem) 3. **Update** every downstream table that referenced the old task_id 4. **Lose history** of the task's evolution through mission analysis

Worse: The exact same task—"Secure Route TAMPA"—might be: - A **specified task** for 1-75 Ranger Regiment in FRAGO 003 - An **implied task** for 2-75 Ranger Regiment (derived from "Support 1-75's main effort") - An **essential task** for the overall brigade operation

If task type is baked into the entity type, you now need three separate Task entities for the same tactical action. Your mission tracking system explodes into a graph of duplicates, and when someone asks "what's the status of Route TAMPA security?" you get three different answers.

CORRECT APPROACH (task is intrinsic; classification is relational):

```
@dataclass
class Task:
    """A task is a unit of tactical work. Period."""
    task_id: UUID
    description: str
    purpose: Optional[str]
    method: Optional[str]
    end_state: Optional[str]
    created_at: datetime
    created_by: str

@dataclass
class OrderTask:
    """The relationship between an order and a task carries the classification."""
    order_id: UUID
    task_id: UUID
    task_type: Literal['specified', 'implied', 'essential']  # Lives on the
relationship!
    priority: int
    assigned_unit: str
    specified_in_paragraph: Optional[str]  # "Para 3.a.(2)" for specified tasks
    derived_from_task: Optional[UUID]  # For implied tasks
    valid_from: datetime
    valid_to: Optional[datetime]

@dataclass
class Order:
    order_id: UUID
    order_number: str
    issuing_unit: str
    effective_datetime: datetime
```

Now you can: - Query: "Give me all tasks in OPORD 25-001" → JOIN on order_id - Query: "What orders assigned 'Secure Route TAMPA' as a specified task?" → JOIN on task_id WHERE task_type='specified' - Handle evolution: When the task becomes essential, INSERT a new OrderTask row with the same task_id but task_type='essential' and a new valid_from timestamp - Avoid duplication: "Secure Route TAMPA" exists once in the Task table, referenced by multiple OrderTask relationships - Preserve history: You can see that the task was reclassified from specified to essential on 2024-06-15

Real MDMP scenario:

```
-- "Show me how task priorities evolved through COA development"
SELECT
    t.description,
    ot.task_type,
    ot.priority,
    o.order_number,
    ot.valid_from
FROM tasks t
JOIN order_tasks ot ON t.task_id = ot.task_id
JOIN orders o ON ot.order_id = o.order_id
WHERE t.description LIKE '%Route TAMPA%'
ORDER BY ot.valid_from;

-- Returns a timeline showing the same task's classification changing
-- across WARNO, FRAGO 001, FRAGO 002 as the operation evolved
```

Example 2: Study Issues Are Study Issues

WRONG APPROACH (classification as type):

```
class FlyTrapStudyIssue(Base):
    __tablename__ = 'flytrap_study_issues'
    issue_id = Column(UUID, primary_key=True)
    description = Column(Text)

class ProjectConvergenceStudyIssue(Base):
    __tablename__ = 'project_convergence_study_issues'
    issue_id = Column(UUID, primary_key=True)
    description = Column(Text)

class PacificDeterrenceStudyIssue(Base):
    __tablename__ = 'pacific_deterrence_study_issues'
    issue_id = Column(UUID, primary_key=True)
    description = Column(Text)
```

The nightmare: You discover that "Munitions expenditure rate exceeds production capacity" is relevant to: - Flytrap (peer conflict in Europe) - Project Convergence (joint all-domain operations) - Pacific Deterrence (China contingency) - Global Force Management (strategic readiness)

Now you have **four separate issue objects** for the same underlying problem. When someone asks "what's the resolution status on the munitions production issue?" you get four different answers because each study team tracked it independently. When DOD issues guidance resolving the issue, you need to update four separate tables. When you want to report "common issues across all studies," you're doing massive UNION queries or—more likely—giving up and building a spreadsheet.

The parallel type explosion: - Year 1: 3 studies × 1 classification = 3 issue types - Year 2: 7 studies × 2 classifications (technical vs. policy) = 14 types - Year 3: 12 studies × 3 classifications × 2 urgency levels = 72 types - Year 4: You're maintaining hundreds of parallel tables with duplicate schemas

CORRECT APPROACH (issue is intrinsic; study association is relational):

```
@dataclass
class StudyIssue:
    """A study issue is a problem, question, or finding. Period."""
    issue_id: UUID
    title: str
    description: str
    category: Literal['technical', 'policy', 'doctrinal', 'resourcing']
    identified_date: date
    identified_by: str

@dataclass
class Study:
    """A study is an analytical effort."""
    study_id: UUID
    study_name: str  # "Flytrap", "Project Convergence", etc.
    study_code: str  # "FT-2024", "PC-2025"
    sponsor: str
    start_date: date
    end_date: Optional[date]

@dataclass
class StudyIssueAssociation:
    """The relationship between a study and an issue carries study-specific
context."""
    study_id: UUID
    issue_id: UUID
    priority: Literal['high', 'medium', 'low']
    status: Literal['open', 'in_progress', 'deferred', 'resolved']
    assigned_to: str
    date_associated: date
    study_specific_notes: Optional[str]
    target_resolution_date: Optional[date]
    actual_resolution_date: Optional[date]
```

Now you can: - **Deduplicate**: One issue, multiple study associations - **Cross-study analysis**: "What issues appear in >3 studies?" → GROUP BY issue_id HAVING COUNT(DISTINCT study_id) > 3 - **Track divergence**: Same issue, different resolution status/priority per study - **Preserve context**: Study-specific notes and timelines without fragmenting the core issue - **Report efficiently**: "All open issues across the portfolio" → WHERE status='open' (not UNION across 47 tables)

Real analysis scenario:

```
-- "Show me high-priority unresolved issues that span multiple theaters"
SELECT
    si.title,
    si.description,
    COUNT(DISTINCT s.study_id) as num_studies,
    STRING_AGG(s.study_name, ', ') as affected_studies
FROM study_issues si
JOIN study_issue_associations sia ON si.issue_id = sia.issue_id
JOIN studies s ON sia.study_id = s.study_id
```

```
WHERE sia.priority = 'high'
  AND sia.status IN ('open', 'in_progress')
GROUP BY si.issue_id, si.title, si.description
HAVING COUNT(DISTINCT s.study_id) > 1
ORDER BY num_studies DESC;

-- Instantly reveals enterprise-level risks spanning multiple efforts
-- Try doing this with FlyTrapStudyIssue, ProjectConvergenceStudyIssue, etc.
```

Example 3: Units Are Units

WRONG APPROACH:

```
class RegularUnit(Base):
    __tablename__ = 'regular_units'

class NationalGuardUnit(Base):
    __tablename__ = 'national_guard_units'

class ReserveUnit(Base):
    __tablename__ = 'reserve_units'
```

The problem: When the 29th Infantry Division (ARNG) gets mobilized for CENTCOM rotation, do you migrate it from `national_guard_units` to `regular_units`? When it demobilizes, do you migrate it back? What about dual-status units? What about when you need to run force structure analysis that includes "all brigade combat teams regardless of component"?

CORRECT APPROACH:

```
@dataclass
class Unit:
    uic: str  # Unit Identification Code - the REAL identity
    name: str
    echelon: Literal['team', 'squad', 'section', 'platoon', 'company', 'battalion',
'brigade', 'division', 'corps', 'army']

@dataclass
class UnitStatus:
    uic: str
    component: Literal['regular', 'national_guard', 'reserve']  # Mutable attribute!
    mobilization_status: Literal['demobilized', 'alerted', 'mobilized', 'deployed']
    readiness_level: Literal['C1', 'C2', 'C3', 'C4']
    effective_date: date
    end_date: Optional[date]
```

The 29th ID is always UIC W1AAAA. Its component, mobilization status, and readiness are temporal facts about that unit, not different types of units.

Example 4: Weapon Systems Are Weapon Systems

WRONG APPROACH:

```
class OperationalWeaponSystem(Base):
    __tablename__ = 'operational_weapon_systems'

class DevelopmentalWeaponSystem(Base):
    __tablename__ = 'developmental_weapon_systems'

class LegacyWeaponSystem(Base):
    __tablename__ = 'legacy_weapon_systems'
```

The lifecycle problem: The F-35 was developmental, became operational, and will eventually become legacy. If status is baked into the type, you're migrating objects across tables every few years. When you want to run an attrition model that includes "all fighter aircraft," you're UNION-ing three tables and praying the schemas stayed aligned.

CORRECT APPROACH:

```
@dataclass
class WeaponSystem:
    system_id: str  # "F-35A", "M1A2 SEPv3"
    nomenclature: str
    category: Literal['ground', 'air', 'naval', 'cyber', 'space']

@dataclass
class SystemAcquisitionStatus:
    system_id: str
    lifecycle_phase: Literal['concept', 'development', 'production', 'operational',
'sustainment', 'disposal']
    milestone: str  # "Milestone B", "IOC", "FOC"
    effective_date: date
    end_date: Optional[date]
```

Migrate Objects Between Types

When you model classification as object type, changing classification requires **type migration**—physically moving the object from one table/collection to another, or changing its discriminator column in a way that breaks polymorphic assumptions.

Temporal

"Temporal" means **time-dependent**—the value changes based on *when* you ask the question. In military operations, near everything is temporal, which is why modeling it correctly is mission-critical.

Temporal facts in military context:

1. **Task assignment temporality**:
2. "What tasks did 2-75 have in OPORD 24-015?" (valid-time)
3. "When did we record that task assignment?" (transaction-time)
4. "What did we think 2-75's tasks were when we briefed the CG on Tuesday?" (transaction-time query)
5. **Study issue status temporality**:
6. "What was the status of the munitions issue during the FY25 PPBE cycle?" (valid-time)
7. "When did we learn the issue was resolved?" (transaction-time)
8. "Show me all issues that were open for >90 days" (valid-time duration)
9. **Force structure temporality**:
10. "What units were assigned to V Corps on 2022-02-24?" (valid-time - day Russia invaded)
11. "What did our MTOE database show for V Corps when we ran the February wargame?" (transaction-time)
12. "How has 1-75's subordinate structure changed since 2020?" (valid-time evolution)

Mutable Attribute

"Mutable" = the value can change. The question is: **what changes, and what stays the same?**

Immutable (identity): - A task's description: "Secure Route TAMPA from CP 1 to CP 7" - A study issue's core problem: "Munitions expenditure exceeds production" - A unit's UIC: W1AAAA for 29th Infantry Division - A weapon system's baseline: M1A2 is an M1A2

Mutable (classification): - Task type in context of an order: specified → essential - Issue status in context of a study: open → in_progress → resolved - Unit readiness level: C1 → C2 → C1 - System acquisition phase: development → production → operational

The trap for military data:

"But task type is really important for MDMP!" → Yes, so model it **explicitly as a time-variant relationship between the task and the order**, not implicitly by making SpecifiedTask and ImpliedTask different entity types.

"But we need to treat open issues differently from resolved issues!" → Yes, so query `WHERE status='open'`, don't create OpenStudyIssue and ResolvedStudyIssue types that require migration when status changes.

Smell test for military context: - Can a commander's decision change this attribute? → Mutable, not ontological - Can a FRAGO change this attribute? → Mutable, not ontological - Does this attribute appear in time-phased force deployment data (TPFDD)? → Probably mutable - Would this attribute changing break foreign keys? → You modeled it wrong

Ontological Category

"Ontological" = relating to the fundamental nature of being. An **ontological category** is a type distinction that reflects what something *fundamentally is*, not what role it currently plays.

Ontological (legitimate type distinctions in military domain): - `Task` vs. `Event` vs. `Decision` (fundamentally different things) - `Unit` vs. `WeaponSystem` vs. `Facility` (different entity categories) - `Study` vs. `Exercise` vs. `Operation` (different types of activities) - `GroundVehicle` vs. `Aircraft` vs. `Ship` (fundamentally different platforms)

Not ontological (context-dependent classifications): - `SpecifiedTask` vs. `ImpliedTask` vs. `EssentialTask` (all are Tasks, differ only in relationship to Order) - `FlyTrapStudyIssue` vs. `EFDLStudyIssue` (all are StudyIssues, differ only in relationship to Study) - `NATOCountry` vs. `NonNATOCountry` (all are Countries, differ only in alliance membership) - `2crUnit` vs. `Unit` (Should be a [[saved public object set]] and not a type)

The test: Can the entity fundamentally transform from one type to another through normal military operations? If yes, it's not an ontological distinction.

- Can a task become a different task? No—but it can be reclassified (specified→essential)
- Can a study issue become a different issue? No—but it can change status (open→resolved)
- Can a unit become a different unit? No—but it can change component, readiness, mission
- Can a tank become a helicopter? No—ontologically distinct
- Can a task be specified in one order and implied in another? **Yes**—so task type is not ontological

Real consequences:

If you make `SpecifiedTask` an ontological type, you need: - Separate tables or inheritance hierarchy - Separate business logic for each task type - Migration procedures when tasks are reclassified - Duplicate code for operations that apply to all tasks (status tracking, assignment, completion)

When an analyst says "let's make a FlyTrapStudyIssue type," ask: - "Is this a fundamentally different KIND of issue, or just an issue associated with a particular study?" - "If we create 12 more studies, do we need 12 more issue types?" - "Can the same underlying problem exist in multiple studies?"

The answer reveals whether it's ontological (different kind of thing) or classificational (same thing in different context).

Handle Entity Resurrection

"Entity resurrection" is the nightmare scenario where an entity gets deleted, then needs to be recreated—often with the same ID, leading to ambiguity about whether it's the same entity or a new one.

Scenario 1: Task resurrection

Your S3 removes "Establish CCP at Grid 12345678" from OPORD 25-001 because it's no longer needed. Two weeks later, enemy situation changes—task is back. If tasks are modeled with type-based deletion:

```
# Bad: Task type determines existence
session.delete(specified_task)  # Task is gone from database

# Two weeks later...
new_task = SpecifiedTask(
    task_id=uuid4(),  # New ID? Same ID as deleted one?
    description="Establish CCP at Grid 12345678"
)
```

Problems: - Is this the same task or a new task with same description? - Historical status reports referenced the old task_id—now broken - Execution checklists for the deleted task—orphaned - After-action review can't connect "original task cancelled" to "task reinstated" - Mission tracking dashboard shows two separate tasks instead of one with interrupted timeline

Scenario 2: Study issue resurrection

"Directed energy weapon countermeasures" issue is closed in Flytrap study when requirements are validated. One year later, new intelligence reveals the requirement was insufficient—issue is reopened.

Problems: - Lost all the original analysis, decisions, and resolution rationale - Can't link "why we reopened this" to "why we closed it originally" - Reports showing "issues closed in FY24" now undercount - No way to query "issues that were resolved then reopened"

Correct approach (identity-based modeling):

```
# Tasks never die, they just become inactive
@dataclass
class Task:
    task_id: UUID  # Permanent identity
    description: str

@dataclass
class OrderTask:
    order_id: UUID
    task_id: UUID  # References permanent Task
    task_type: Literal['specified', 'implied', 'essential']
    status: Literal['active', 'suspended', 'cancelled', 'complete']
    valid_from: datetime
    valid_to: Optional[datetime]  # NULL = currently valid

# Original assignment
order_task_1 = OrderTask(
    order_id=opord_001,
    task_id=ccp_task.task_id,
    task_type='specified',
    status='active',
    valid_from='2024-06-01'
)
```

```
# Task cancelled
order_task_1.status = 'cancelled'
order_task_1.valid_to = '2024-06-15'

# Task reinstated (SAME task_id!)
order_task_2 = OrderTask(
    order_id=frago_003,
    task_id=ccp_task.task_id,  # Same task!
    task_type='essential',  # Now essential instead of specified
    status='active',
    valid_from='2024-06-29'
)

# Query: "Show me the lifecycle of the CCP task"
SELECT
    o.order_number,
    ot.task_type,
    ot.status,
    ot.valid_from,
    ot.valid_to
FROM tasks t
JOIN order_tasks ot ON t.task_id = ot.task_id
JOIN orders o ON ot.order_id = o.order_id
WHERE t.description LIKE '%CCP%'
ORDER BY ot.valid_from;

# Returns:
# OPORD 25-001 | specified | cancelled | 2024-06-01 | 2024-06-15
# FRAGO 003    | essential | active    | 2024-06-29 | NULL
```

No resurrection. No ambiguity. Full history preserved.

Study issue approach:

```
@dataclass
class StudyIssue:
    issue_id: UUID  # Permanent identity
    title: str
    description: str

@dataclass
class StudyIssueAssociation:
    study_id: UUID
    issue_id: UUID  # References permanent StudyIssue
    status: Literal['open', 'in_progress', 'deferred', 'resolved', 'reopened']
    opened_date: date
    resolved_date: Optional[date]
    reopened_date: Optional[date]
    resolution_notes: Optional[str]

# Issue closed
association.status = 'resolved'
association.resolved_date = date(2024, 8, 1)
association.resolution_notes = "Requirements validated per XYZ memo"
```

```
# Issue reopened
association.status = 'reopened'
association.reopened_date = date(2025, 9, 15)
association.resolution_notes += "\n\nREOPENED: New intel shows requirement
insufficient"

# Query: "Issues that were resolved then reopened"
SELECT * FROM study_issue_associations
WHERE reopened_date IS NOT NULL;
```

No entity deletion. No resurrection ambiguity. Full audit trail.

Parallel Types

Once you start modeling classification as type, you inevitably create an explosion of parallel types—each combination of classifications requires its own type, and shared logic gets duplicated across all of them.

The task type explosion:

```
Week 1: "Let's separate specified, implied, and essential tasks"
→ 3 types: SpecifiedTask, ImpliedTask, EssentialTask

Week 3: "We need to track which tasks are complete"
→ 6 types: SpecifiedActiveTask, SpecifiedCompleteTask,
           ImpliedActiveTask, ImpliedCompleteTask,
           EssentialActiveTask, EssentialCompleteTask

Week 5: "Actually, we need to track if tasks are assigned or unassigned"
→ 12 types (3 task types × 2 statuses × 2 assignment states)

Week 7: "Boss wants to track priority: routine, priority, immediate"
→ 36 types (3 × 2 × 2 × 3)

Week 10: "Can we add whether it's a collective or individual task?"
→ 72 types and your schema is unmaintainable

Week 12: "Why does it take 3 seconds to query task counts?"
→ You're UNION-ing across 72 tables
```

The study issue explosion:

```
Year 1: 4 studies (Flytrap, Convergence, Pacific, Arctic)
→ 4 issue types: FlyTrapIssue, ConvergenceIssue, PacificIssue, ArcticIssue

Year 2: Added 6 more studies, now tracking priority (high/med/low)
→ 10 studies × 3 priorities = 30 types

Year 3: Now tracking technical vs. policy vs. doctrinal issues
→ 10 × 3 × 3 = 90 types

Year 4: DoD mandates tracking joint vs. service-specific issues
```

```
→ 10 × 3 × 3 × 2 = 180 types

Year 5: You've been replaced by a spreadsheet
```

First-Class Temporal Facts

A "first-class" entity is one that exists in its own right in your data model, with its own identity, attributes, and lifecycle—not just as a column or embedded attribute.

THE BOTTOM LINE (EXTENDED)

Separate what things ARE from what STATE they're IN:

- **Identity** (what it is) → entity type
- Task, StudyIssue, Country, Unit, WeaponSystem
- **Classification** (what state it's in) → temporal attributes/relationships
- TaskType (specified/implied/essential) lives on OrderTask relationship
- IssueStatus (open/resolved) lives on StudyIssueAssociation relationship
- AllianceMembership (NATO/EU/etc.) lives on membership relationship
- ComponentStatus (regular/guard/reserve) lives on UnitStatus relationship

When you collapse these, you get:

1. **Unmaintainable schemas**: 72 parallel task types instead of one Task + relationships
2. **Migration nightmares**: Moving objects between tables when classification changes
3. **Inability to query history**: "What were the tasks in June?" becomes impossible
4. **Loss of provenance**: Can't track who changed what, when, or why
5. **Duplication chaos**: Same task appears in multiple "types" with different IDs
6. **Query explosion**: UNION across dozens of tables instead of simple WHERE clauses
7. **Code duplication**: Identical methods copy-pasted across parallel types

In a military context, where you need to: - Model counterfactuals: "What if we'd made this an essential task from the start?" - Retrospective analysis: "Why did our March wargame predict different attrition?" - Cross-study analysis: "Which issues appear in multiple studies?" - Historical queries: "What was force structure when we deployed?" - Audit decisions: "Who reclassified this task and when?"

Getting temporality wrong isn't just technical debt—**it's mission failure**.