# PROJECT IRON PRELUDE: THE CHRONICLES FLIGHT SIMULATOR ARCHITECTURE

## Executive Summary: The Philosophy of the Simulation

The objective of Project Iron Prelude is not merely to replicate the visual interface of Epic Hyperspace but to reconstruct the cognitive architecture of the underlying database, Chronicles. For the Principal Trainer (PT) candidate or the aspiring analyst, the graphical user interface (GUI) is often a deception. It presents a flattened, sanitized view of data that, in reality, exists within a complex, hierarchical, and often counter-intuitive tree structure. To truly master the Training Environment Design (TED) and Master Train (MST) certification exams, the candidate must look past the "Hyperspace veneer" and understand the "Anatomy of Data".

This report serves as the technical blueprint for a "Flight Simulator" web application designed to train these candidates. Unlike a traditional sandbox, which resets indiscriminately, this simulator must obey the rigid laws of Epic's proprietary MUMPS-based architecture. It must enforce the distinction between Static (Administrative) and Dynamic (Transactional) master files. It must rigorously apply the Rule of Specificity, determining how a Department-level setting overrides a Service Area configuration. It must replicate the "Guarantor Trap"—the complex liability logic that governs pediatric registration.

We are building a simulation of the "Chronicles Kernel." This document outlines the schema, the logic, and the operational scripts required to instantiate this environment. It is written for the Senior Systems Architect and the Lead Instructional Designer, merging high-level pedagogical strategy with low-level code implementation. The simulation methodology described herein—Mitosis—is designed to solve the perennial problem of "environment rot" in training environments by mathematically sliding patient data relative to an anchor date, ensuring that a "newborn" admitted three days ago remains exactly three days old for every subsequent trainee, in perpetuity.

This architectural specification is divided into four primary phases. Phase 1 details the backend schema design, explicitly rejecting relational norms in favor of a hierarchical structure that mirrors the sparse arrays of MUMPS. Phase 2 constructs the pediatric workflow engine, enforcing hard stops and validation logic that mirror the liability constraints of a children's health system. Phase 3 defines the "Mitosis" engine, the Python-based automation layer responsible for the nightly regeneration of the environment. Finally, Phase 4 outlines the "Game Over" scenarios—educational traps designed to force candidates to confront the nuances of inheritance, filing order, and bed planning logic.

# Chapter 1: The Chronicles Kernel (Phase 1)

## 1.1 The MUMPS Paradigm Shift and Database Architecture

To build a faithful simulator, one must first discard the Relational Database Management System (RDBMS) paradigm that dominates modern web development. Chronicles is not a table-based

system; it is a hierarchical, tree-based system derived from MUMPS (Massachusetts General Hospital Utility Multi-Programming System). In the context of our simulator, attempting to force this logic into a normalized SQL structure would be a fundamental design error, leading to a simulation that fails to capture the very quirks—such as the difference between "Lines" and "Rows"—that trip up candidates on their exams.

In a standard SQL environment, a patient's address might be a row in a Addresses table linked by a foreign key. In Chronicles, data is stored in "Globals"—persistent, sparse, multi-dimensional arrays. A patient record is a "node" in a global tree, and specific data points (Contacts, Items, Values) are branches and leaves off that node. The simulator must replicate this. We cannot simply use a PostgreSQL database with standard normalization; instead, we must design a NoSQL document store (using a complex JSON structure) that faithfully mimics the INI > Record > Contact > Item > Value hierarchy described in the technical audit.

The MUMPS architecture, as described in the research literature, is characterized by its use of "globals"—variables that persist on disk exactly as they are represented in memory. This unification of memory and storage logic is alien to most modern developers who are accustomed to an Object-Relational Mapper (ORM) layer. In MUMPS, and by extension in Chronicles, the "database" is simply a structured array. For example, the variable ^EPT (representing the Patient Master File) might have a subscript for the Patient ID, a sub-subscript for the Item Number, and a value. Our simulator's backend must treat data with this same level of granularity. If we simply stored a patient as a flat JSON object {"name": "John", "dob": "2020-01-01"}, we would fail to teach the candidate about "Contacts" (datestamps of change) or "Items" (discrete field definitions), both of which are critical for passing the TED 300 exam.

### 1.1.1 The Anatomy of Data Definitions

The simulator must implement the following taxonomy, which serves as the vocabulary for the entire Epic ecosystem. Understanding these terms is the first step in the "cognitive shift" required of the candidate.

- **Master File (INI):** The "File Cabinet Drawer." This is the highest level of organization. A Master File stores all data of a specific type. It is identified by a three-character "INI" (e.g., EPT for Patient, DEP for Department, EAR for Guarantor). In our JSON schema, these will be the root keys of our document store.
- **Record:** The "File Folder." A specific entity within a Master File, identified by a unique ID (internally known as the .1 Item). For a patient, the Record ID is the Medical Record Number (MRN) or internal system ID. For a Department, it is the Department ID.
- **Contact:** The "Sheet of Paper." This concept is unique to the temporal nature of Chronicles. A Contact represents a snapshot of data valid for a specific instant or period. For example, an "Admission" is a contact on a patient record. An "Address Change" is a contact. This allows Chronicles to track the history of a patient over time without creating duplicate records. In our simulator, the contacts array will be the primary mechanism for storing clinical data.
- **Item:** The "Form Question." A discrete field definition. Every piece of data in Epic has an Item Number. For example, Item 50 might be "Sex," and Item 110 might be "Date of Birth." The simulator must store data as Item: Value pairs to allow for the specific "Item-level" logic often tested in exams.
- **Value:** The "Written Answer." The actual data stored in the item (e.g., "Male", "01/01/2020").

## 1.2 The TypeScript Schema Design for the ChroniclesNode

The following TypeScript interfaces define the ChroniclesNode structure. This schema is designed to handle both the "Static" infrastructure and the "Dynamic" patient data, allowing for the "Mitosis" logic (Section 3) to function correctly. The design explicitly avoids relational foreign keys in favor of nested structures or direct ID references, mimicking the "Global" storage mechanism of MUMPS.

The schema differentiates between Static and Dynamic volatility at the Type level. This is a critical architectural decision. By strictly typing DepartmentRecord as STATIC and PatientRecord as DYNAMIC, the TypeScript compiler itself will help enforce the logic of the Mitosis engine—preventing us from accidentally writing a function that "wipes" the Departments or "archives" the Patients.

```
/**
 * PROJECT IRON PRELUDE: CHRONICLES KERNEL SCHEMA
 *
 * This schema mimics the hierarchical nature of Epic's Chronicles
database.
 * It favors nested structures over relational foreign keys to
replicate
 * the 'Global' storage mechanism of MUMPS.
 *
 * Source Truth: "Anatomy of Data"
 */

//
-------------------------------------------------------------------
-----
// CORE CHRONICLES TYPES
//
-------------------------------------------------------------------
-----

/**
 * The Master File (INI).
 * Represents the highest level of organization (e.g., EPT, EAR, DEP).
 * Each INI acts as a namespace for a specific type of data.
 */
export enum INI {
  EPT = "EPT", // Patient (Dynamic) - The Clinical "Who"
  EAR = "EAR", // Guarantor (Dynamic) - The Financial "Payer"
  DEP = "DEP", // Department (Static) - The Login Context
  EAF = "EAF", // Facility/Location/Service Area (Static) - The
Structure
  ROM = "ROM", // Room (Static) - Physical Space
  BED = "BED", // Bed (Static) - Census Capacity
  HSP = "HSP", // Hospital Account (Dynamic) - The Billing Bucket
}
```

```
/**
 * Data Volatility Type.
 * Critical for the 'Mitosis' engine to know what to wipe and what to
keep.
 * Static files are 'Administrative' and migrated via Data Courier.
 * Dynamic files are 'Transactional' and grow with hospital
operations.
 */
export enum Volatility {
  STATIC = "STATIC",   // Administrative
  DYNAMIC = "DYNAMIC", // Transactional
}

//
---------------------------------------------------------------------
-----
// HIERARCHY NODE DEFINITIONS
//
---------------------------------------------------------------------
-----

/**
 * A generic Value in Chronicles.
 * Can be a string, number, boolean, or null.
 * In MUMPS, everything is technically a string, but we use types for
safety.
 */
type ChroniclesValue = string | number | boolean | null;

/**
 * An Item (Field).
 * Represents a specific data point (e.g., DOB, SSN).
 * Mapped by Item ID (e.g., ".1" for Record ID).
 */
export interface Item {
  id: string;          // The Item Number (e.g., "110")
  name: string;        // The human-readable name (e.g., "Date of
Birth")
  value: ChroniclesValue;
  previousValue?: ChroniclesValue; // For auditing changes across
contacts
  isRequired?: boolean; // Validation metadata
}

/**
 * A Contact (DAT).
 * Represents a date-specific snapshot or event.
```

```
 * In MUMPS, contacts are often indexed by 'Reverse Date' (DAT) for
sorting efficiency.
 */
export interface Contact {
  dat: number;         // Date at Time (Inverse Date format often used
in MUMPS)
  contactType: string; // E.g., "Admission", "Discharge", "Update"
  instant: string;     // ISO 8601 Timestamp for the simulation
  items: Map<string, Item>; // Keyed by Item ID for O(1) lookup
}

/**
 * A Record.
 * The fundamental unit (e.g., Patient John Doe).
 * Base interface extended by specific INI types.
 */
export interface Record {
  ini: INI;
  recordId: string;    // The.1 Item (Unique Identifier)
  name: string;        // The.2 Item (Record Name)
  volatility: Volatility;

  // No-Add setting: Static records often have this to prevent user
creation.
  isNoAdd: boolean;

  // The Contacts tree.
  // For Static files (DEP), often only one contact exists (index 0).
  // For Dynamic files (EPT), this is a history of care.
  contacts: Contact;

  // Linkage for Facility Structure Rule of Specificity
  // These pointers allow the "Bubble Up" logic to traverse the tree.
  parentId?: string;   // Pointer to the parent record (e.g., Dept ->
Loc)
  parentIni?: INI;     // The INI of the parent record
}

//
---------------------------------------------------------------------
-----
// MASTER FILE IMPLEMENTATIONS
//
---------------------------------------------------------------------
-----

/**
 * Patient Master File (EPT)
```

```
 * Dynamic: Grows continuously, never migrated.
 * Represents the clinical human being.
 */
export interface PatientRecord extends Record {
  ini: INI.EPT;
  volatility: Volatility.DYNAMIC;

  // Specific logical links for the "Guarantor Trap"
  guarantorLinks: string; // Array of EAR Record IDs

  // Simulation-specific metadata for Mitosis
  isTemplate?: boolean;     // If true, this is a "Golden Record"
  scenarioDobOffset?: number; // Days offset from Today for Date
Sliding
}

/**
 * Guarantor Master File (EAR)
 * Dynamic: Represents the financial entity.
 * In Pediatrics, this is rarely the patient themselves.
 */
export interface GuarantorRecord extends Record {
  ini: INI.EAR;
  volatility: Volatility.DYNAMIC;

  // Guarantor Type Enum
  guarantorType: "Personal/Family" | "TPL" | "Workers Comp" |
"Research";

  subscriberLink?: string; // If TPL, links to subscriber
  address?: string;        // Used for address matching validation
}

/**
 * Department Master File (DEP)
 * Static: Defined by analysts.
 * The primary login context for users.
 */
export interface DepartmentRecord extends Record {
  ini: INI.DEP;
  volatility: Volatility.STATIC;

  // Hierarchy Links (The Path of Specificity)
  // These link the Department to the Revenue Location (EAF)
  revenueLocationId: string;
  specialty?: string; // e.g., "Pediatrics", "Oncology"
}
```

```
/**
 * Facility/Structure Master File (EAF)
 * Static: Represents Facility, Service Area, Location.
 * This generic file handles the upper levels of the hierarchy.
 */
export interface FacilityStructureRecord extends Record {
  ini: INI.EAF;
  volatility: Volatility.STATIC;
  type: "Facility" | "Service Area" | "Location";

  // Up-tree linkage
  parentStructureId?: string; // e.g., Location -> Service Area
}

/**
 * The Database Container
 * Represents the entire Chronicles environment.
 * Organized by Master File (INI).
 */
export interface ChroniclesDatabase {
 : Map<string, PatientRecord>;
 : Map<string, GuarantorRecord>;
 : Map<string, DepartmentRecord>;
  [INI.EAF]: Map<string, FacilityStructureRecord>;
 : Map<string, any>; // Hospital Accounts
 : Map<string, any>; // Rooms
 : Map<string, any>; // Beds
}
```

## 1.3 The Rule of Specificity Algorithm

The core logic of the simulator is the **Rule of Specificity**. In the Epic ecosystem, configuration is rarely done at the user level. Instead, it is attached to the **Facility Structure**. When the system needs to know a rule (e.g., "What are the visitation hours?", "What is the default lab printer?", "Is this unit male-only?"), it does not simply check a single table. It traverses a hierarchy from the most specific context (the Department) to the most general context (the Facility).
This "inheritance model" is efficient because it allows an organization to set global defaults (e.g., "Visiting hours are 8am-8pm") at the Facility level, while allowing specific units (e.g., "ICU") to override them (e.g., "Visiting hours are 24/7"). For the simulator, implementing this logic is non-negotiable. If we fail to replicate this, the "Inheritance Fail" scenario in Phase 4 will be impossible to generate.

### 1.3.1 The Hierarchy Chain

As defined in Section 1.2 of the audit , the lookup order is rigidly defined. The simulator must respect this precise order:

1. **Department (DEP):** The login context. (Most Specific). The user is physically "in" this unit.
2. **Revenue Location (EAF):** The physical building or campus (e.g., "Main Hospital").
3. **Service Area (EAF):** The business entity or Accounts Receivable (AR) region. Guarantor accounts are scoped here.
4. **Facility (EAF):** The enterprise/System Definitions (LSD). (Most General).

### 1.3.2 The Lookup Function Logic

This logic must be implemented in the simulator's "Kernel" class. Whenever a workflow requests a setting, this function is invoked. We must implement a "Bubble Up" mechanism. The algorithm queries the current level; if the value is null or undefined, it retrieves the ID of the parent level and recursively queries that level.
The logic flow is as follows:

- **Input:** Setting_ID (e.g., "VISITATION_HOURS") and User_Context (The ID of the DEP the user is logged into).
- **Step 1:** Retrieve the DepartmentRecord using the User_Context. Check if the item Setting_ID exists and has a value.
    - *If Yes:* Return the value. Log "Resolved at DEP level."
    - *If No:* Retrieve the revenueLocationId from the Department record.
- **Step 2:** Retrieve the FacilityStructureRecord for the Location. Check if Setting_ID exists.
    - *If Yes:* Return the value. Log "Resolved at LOC level."
    - *If No:* Retrieve the parentStructureId (Service Area) from the Location record.
- **Step 3:** Retrieve the FacilityStructureRecord for the Service Area. Check if Setting_ID exists.
    - *If Yes:* Return the value. Log "Resolved at SA level."
    - *If No:* Retrieve the parentStructureId (Facility) from the Service Area record.
- **Step 4:** Retrieve the FacilityStructureRecord for the Facility. Check if Setting_ID exists.
    - *If Yes:* Return the value. Log "Resolved at FAC level."
    - *If No:* Return the System Default or throw a configuration error.

**Architectural Insight:** The failure to understand this lookup logic is the primary cause of the "Certification Trap" mentioned in the source. Candidates often configure a rule at the Service Area level (expecting it to apply everywhere) but fail to realize that a pre-existing, blank, or contradictory setting at the Department level is overriding it (blocking the inheritance). Our simulator must actively log this "Blocking Event" to teach the user *why* their build failed. In CSS terms, this is analogous to an ID selector overriding a Class selector—specificity wins.

# Chapter 2: The Pediatric "Guarantor Trap" (Phase 2)

## 2.1 The Architecture of Liability in Pediatrics

In the Prelude module (Enterprise Registration), the distinction between the **Patient (EPT)** and the **Guarantor (EAR)** is paramount. The Patient is the clinical entity; the Guarantor is the financial entity responsible for the "Self-Pay" portion of the bill. In an adult context, these are often the same person (Relationship = "Self"). However, in the context of a **Pediatric Health System** (Cook Children's Profile), this default assumption is legally hazardous.
A minor (under 18) generally lacks the legal capacity to enter into a binding financial contract. Therefore, the Patient *cannot* be their own Guarantor. This creates a unique "Hard Stop"

workflow that the simulator must enforce. The simulator must detect the patient's age and, if they are a minor, disable the "Self" relationship option or flag it as a critical error. This is the "Guarantor Trap": trainees accustomed to adult workflows will instinctively select "Self," only to be blocked by the validation engine.

## 2.2 The Registration Validation Engine

The following section outlines the logic driving the "Finish Registration" button in the simulator. The button is effectively a state machine that transitions from DISABLED to ENABLED only when all liability constraints are satisfied. This validation must occur in real-time (on field exit) to provide immediate feedback, mirroring the "Sidebar Checklist" behavior in Epic Hyperspace.

### 2.2.1 Logic Requirements

Based on Section 2.1 of the audit , the validation rules are:

1. **Linkage:** A Guarantor Account must be linked to the Patient. A patient cannot exist in a vacuum; there must be a financial entity attached.
2. **Pediatric Constraint:** If Patient.Age < 18, then Guarantor.ID must not equal Patient.ID. This is the primary pediatric constraint. (Exceptions for "Emancipated Minors" may exist in advanced scenarios, but for the baseline simulation, this is a hard rule).
3. **Type Validity:** The Guarantor Type must be one of the enumerated valid types: "Personal/Family", "Third Party Liability" (TPL), "Workers Comp", or "Research".
4. **TPL Integrity:** If the Guarantor Type is "Third Party Liability" (used for auto accidents/torts), an "Accident Date" is mandatory.
5. **Address Integrity:** The Guarantor's address should match the Subscriber's address. While strictly a soft warning in some systems, in our "Iron Prelude" protocol, we will treat mismatched addresses as a "Compliance Warning" that requires user acknowledgement, as it is a leading cause of claim denials.

### 2.2.2 The Pseudo-Code Implementation (Pythonic Logic)

This pseudo-code demonstrates the RegistrationValidator class. This class would sit on the simulator's application server, processing the JSON payload sent by the frontend whenever the user attempts to save the registration.

```
class RegistrationValidator:
    """
    Validates the registration payload against Pediatric constraints.
    Enforces the 'Guarantor Trap' logic.
    """
    def __init__(self, patient_data, guarantor_list,
insurance_subscriber):
        self.patient = patient_data
        self.guarantors = guarantor_list
        self.subscriber = insurance_subscriber
        self.errors =
        self.warnings =

    def validate_workflow(self):
```

```
        """
        Main validation loop running on every field exit.
        Returns: Tuple (Is_Finish_Allowed (bool), Status_Messages
(list))
        """
        self.errors =
        self.warnings =

        # 1. Check for Guarantor Linkage
        if not self.guarantors:
            self.errors.append("HARD STOP: No Guarantor Account
linked.")
            return False, self.errors

        # Assuming primary guarantor is the first in the list for this
context
        primary_guarantor = self.guarantors

        # 2. Pediatric Constraint (The "Guarantor Trap")
        # Logic: A child cannot be their own guarantor.
        patient_age = self._calculate_age(self.patient['dob'])
        if patient_age < 18:
            # Check if the linked Guarantor ID matches the Patient ID
            if primary_guarantor['id'] == self.patient['id']:
                self.errors.append("HARD STOP: Minor patient cannot be
Self-Guarantor. "
                                   "Please link a Parent or Legal
Guardian.")
                # Insight: This forces the user to create a separate
EAR record.

        # 3. Guarantor Type Logic
        # Logic: Valid types defined in Section 2.1
        valid_types =
        if primary_guarantor['type'] not in valid_types:
             self.errors.append(f"HARD STOP: Invalid Guarantor Type
'{primary_guarantor['type']}'.")

        # 4. TPL (Third Party Liability) Specific Logic
        # If type is TPL (e.g., auto accident), ensure TPL specific
fields are present.
        if primary_guarantor['type'] == "Third Party Liability":
            if not primary_guarantor.get('accident_date'):
                self.errors.append("HARD STOP: TPL Account requires
Accident Date.")

        # 5. Address Match Logic (Soft Stop/Warning)
        # Logic: Guarantor address should match Subscriber address to
```

```
prevent claim denial.
        # This is a 'Warning', not a 'Hard Stop', unless configured
otherwise.
        if primary_guarantor.get('address')!=
self.subscriber.get('address'):
            self.warnings.append("WARNING: Guarantor address mismatch
with Subscriber. "
                                 "This may cause claim denial.")

        # Conclusion
        if len(self.errors) > 0:
            # If ANY hard errors exist, the Finish button is disabled.
            return False, self.errors + self.warnings
        else:
            # If only warnings exist, the button is enabled (but
warnings are shown).
            return True, + self.warnings

    def _calculate_age(self, dob_string):
        """Helper to calculate age from DOB string."""
        from datetime import date, datetime
        dob = datetime.strptime(dob_string, "%Y-%m-%d").date()
        today = date.today()
        return today.year - dob.year - ((today.month, today.day) <
(dob.month, dob.day))
```

### 2.3 Workflow Implication: The "Greyed Out" Button

In the UI of the simulator, the "Finish" button's disabled attribute is bound directly to the boolean output of validate_workflow().

- **Visual Cue:** When errors > 0, the button is Grey (#CCCCCC) and unclickable. Hovering over it displays a tooltip listing the errors.
- **The Trap:** In the exam simulation, the user will be presented with a 16-year-old patient. The system will default to "Self" as the relationship (a common default in poorly configured systems, or a deliberate trap in the exam). The user must actively *remove* the "Self" relationship and search for/create the "Parent" guarantor to enable the button. Failure to do so results in a failed workflow step.

# Chapter 3: The "Mitosis" & "Date Sliding" Engine (Phase 3)

## 3.1 The Static vs. Dynamic Paradox

To simulate a real training environment (Sup/MST), we must address the issue of "Environment Rot." In a real Epic environment, the "Master Patient" (e.g., "ZZZTEST, PEDS") must remain

pristine for the next trainee. However, during a session, the trainee creates specific "Contacts" (Admissions, Orders, Notes) on this patient. If we do not wipe these changes, the next user will see a patient who is already admitted, or who has a cluttered history, breaking the simulation. Furthermore, strict static resets are insufficient for date-sensitive scenarios. Consider a scenario: *"Admit a patient who was born 3 days ago."* If we hard-code the Date of Birth to 01/01/2023, that patient is "3 days old" only on 01/04/2023. By 02/01/2023, they are a month old, and the scenario fails (e.g., the Newborn Admission Navigator won't fire).
This requires the **Mitosis Engine**: a nightly script that performs two critical functions:

1. **The Wipe:** Destroys the user's Dynamic data while preserving the Static infrastructure.
2. **The Date Slide:** Recalculates the dates of the Master Patients to remain relative to current_time.

## 3.2 The run_mitosis Logic

The script distinguishes between **Static (Administrative)** and **Dynamic (Transactional)** files based on the definitions in Section 1.1.1.

- **Keep (Static):** DEP (Departments), ROM (Rooms), BED (Beds), EPM (Payors), EPP (Plans). These are the "walls" of the hospital.
- **Wipe (Dynamic):** User-created EPT (Patients), User-created EAR (Guarantors), HSP (Hospital Accounts). These are the "transactions."
- **Reset & Slide (Template Dynamic):** The "Master Patients" (Templates). These are Dynamic records that act as Static fixtures. We must revert them to their original state but *slide* their dates.

## 3.3 The Python Build Script

This Python script is the heart of the maintenance operation. It would run as a CRON job every night at 02:00 AM.

```
import datetime
import json
import shutil
from enum import Enum

#
------------------------------------------------------------------------
-----
# CONFIGURATION AND CONSTANTS
#
------------------------------------------------------------------------
-----

DB_PATH = "chronicles_db.json"
GOLDEN_PATH = "golden_records.json"

class Volatility(Enum):
    STATIC = "STATIC"   # Do not touch (e.g., Department definitions)
    DYNAMIC = "DYNAMIC" # Wipe nightly (e.g., User-created encounters)
```

```
class MitosisEngine:
    """
    The Mitosis Engine is responsible for the nightly regeneration of
the
    training environment. It enforces the separation of Static and
Dynamic data
    and executes the 'Date Slide' to keep scenarios fresh.
    """
    def __init__(self, database_path, golden_path):
        self.db_path = database_path
        self.golden_path = golden_path
        self.current_date = datetime.date.today()

        # Load the current database (JSON dump of ChroniclesNode)
        # In a real impl, this might connect to a Mongo instance.
        with open(self.db_path, 'r') as f:
            self.db = json.load(f)

    def run_mitosis(self):
        print(f"[{datetime.datetime.now()}] Initiating MITOSIS
protocol...")

        # Step 1: Wipe User Data
        self._wipe_dynamic_transactions()

        # Step 2: Reload and Slide Master Patients
        self._reload_and_slide_golden_records()

        # Step 3: Save State
        print("MITOSIS complete. Environment reset to T-0.")
        self._save_db()

    def _wipe_dynamic_transactions(self):
        """
        Wipe all Dynamic records (EPT, EAR, HAR) created by the user
        in the last session.
        Logic: Iterate through EPT, keep only records flagged as
'TEMPLATE'.
        Delete all HSP (Hospital Accounts) created during the session.
        """
        print(">> Wiping Dynamic User Data...")

        # 1. Clean Patients (EPT)
        # We iterate over a copy of keys to avoid runtime modification
errors
        ept_keys = list(self.db.keys())
        deleted_count = 0
```

```
        for pat_id in ept_keys:
            patient = self.db[pat_id]
            if patient.get('is_template'):
                # This is a master patient (e.g. ZZZTEST). Keep the
record structure.
                # However, we must wipe any contacts added by the user
today.
                # We assume the 'contacts' list in the DB might have
grown.
                # The _reload function will overwrite this anyway, but
good practice to clear.
                pass
            else:
                # This is a patient created by the user (e.g., "Baby
Boy Doe")
                # DELETE ENTIRELY.
                del self.db[pat_id]
                deleted_count += 1

        print(f"   Deleted {deleted_count} user-created Patient
records.")

        # 2. Clean Guarantors (EAR)
        # Similar logic: Keep templates, wipe user data.
        self.db = {k: v for k, v in self.db.items() if
v.get('is_template')}

        # 3. Clean Hospital Accounts (HSP)
        # Wipe all accounts. Users create HARs, but templates don't
usually have active HARs
        # persisting across sessions (unless specifically designed).
        self.db = {}

        print(">> Dynamic Wipe Complete.")

    def _reload_and_slide_golden_records(self):
        """
        Reload the Static 'Master Patient' set from the Golden Copy.
        This ensures if a user accidentally changed a template's name,
it reverts.
        CRITICAL: Applies Date Sliding Logic.
        """
        print(">> Reloading and Sliding Golden Records...")

        with open(self.golden_path, 'r') as f:
            golden_data = json.load(f)

        # We define "Anchor Date" as the relative point 0 (Today).
```

```
        # Scenarios define dates relative to this.

        for pat_id, template_patient in golden_data.items():
            # 1. Deep Copy the template to avoid mutating the Golden
File in memory
            # (In Python, use copy.deepcopy if structure is complex)
            refreshed_patient = template_patient.copy()

            # 2. Check for Date Slide Configuration
            # Example config in JSON: "scenario_dob_offset": -3 (Born
3 days ago)
            offset_days = refreshed_patient.get('scenario_dob_offset')

            if offset_days is not None:
                # Calculate new DOB relative to TODAY
                # If offset is -3, DOB is 3 days ago.
                new_dob = self.current_date +
datetime.timedelta(days=offset_days)

                # Update the DOB Item (.110) in the baseline contact
                # Assuming contact is the demographic baseline
                # Note: We must navigate the JSON structure carefully.
                # In our schema: contacts -> items -> ItemID -> value

                # Check if contacts exist
                if refreshed_patient.get('contacts'):
                    baseline = refreshed_patient['contacts']
                    if baseline['items'].get('110'): # Item 110 = DOB
                         baseline['items']['110']['value'] =
new_dob.isoformat()
                         print(f"   Slid {refreshed_patient['name']}
DOB to {new_dob} (T{offset_days})")

            # 3. Inject back into Live DB
            self.db[pat_id] = refreshed_patient

    def _save_db(self):
        with open(self.db_path, 'w') as f:
            json.dump(self.db, f, indent=2)

#
----------------------------------------------------------------------
-----
# EXECUTION ENTRY POINT
#
----------------------------------------------------------------------
-----
if __name__ == "__main__":
```

```
# Initialize engine
# In production, this would be triggered by the system scheduler.
engine = MitosisEngine(DB_PATH, GOLDEN_PATH)
engine.run_mitosis()
```

**Instructional Note:** This script mimics the nightly "Environment Refresh" that Epic administrators manage. In the certification exam, users are often warned: *"Do not build in the MST environment after 5 PM, or your work will be wiped."* This script enforces that discipline. The Date Sliding logic is particularly sophisticated; it ensures that the "Newborn" scenario is always available, regardless of whether the trainee logs in on Monday or Friday.

# Chapter 4: The "Exam Scenario" Generator (Phase 4)

This section details the "Game Over" scenarios—specifically designed traps based on the "Certification Trap" sections of the audit. These scenarios test the candidate's ability to troubleshoot complex interaction errors between Master Files. Each scenario is a "Fail State" that the simulator actively generates if the user makes a predictable architectural error.

## Scenario 1: The "Inheritance" Fail (The Facility Structure Trap)

- **The Setup:** The user is asked to configure "Visitation Hours" for the new "Pediatric ICU" (PICU). The exam prompt states: *"The PICU requires 24/7 visitation. Configure this at the most efficient level possible."*
- **The Trap:** The candidate, wanting to be efficient, goes to the **Service Area (EAF)** record and sets Visitation Hours to "24/7", assuming it will cascade down to all units.
- **The Blocking Factor:** The existing **Department (DEP)** record for the PICU (which was pre-built in the Golden Data) has a "hard-coded" setting of "9 AM - 5 PM" hidden in a tab they didn't check.
- **The Result:** The Department setting overrides the Service Area setting due to the **Rule of Specificity**. The simulation shows the visitor being denied entry at 8 PM.
- **Game Over Message:** *"FATAL ERROR: Rule of Specificity Violation. The Department-level override (9-5) blocked your Service Area configuration (24/7). You must clear the Department setting to allow inheritance to flow from the Service Area."*

## Scenario 2: The "Filing Order" Fail (The VFO Trap)

- **The Setup:** The user registers a patient with two insurance coverages: **Medicaid** and **Blue Cross (Commercial)**.
- **The Trap:** The user creates the Hospital Account (HAR) and uses the "Filing Order Calculator" incorrectly, or manually overrides it to list **Medicaid as Primary** and **Commercial as Secondary**.
- **The Compliance Violation:** This violates **Medicaid Secondary Payer (MSP)** rules. Medicaid is chemically defined as the "Payer of Last Resort" by federal statute. It must always be last.
- **The Result:** The system allows the HAR creation (Soft Stop), but when the claim is generated in the background simulation, it is instantly denied.
- **Game Over Message:** *"FINANCIAL FATAL: Instant Claim Denial. Medicaid is the Payer*

*of Last Resort. You placed it in Position 1. Private Insurance must be Primary. Resolute HB cannot bill this account. Logic: Commercial -> Medicaid."*

## Scenario 3: The "Bed Logic" Fail (The Gender/Room Trap)

- **The Setup:** The user builds a new room "301" in the "Obstetrics" department. They configure the Room (ROM) as "Semi-Private" (2 beds).
- **The Trap:** They fail to set the **Gender Restriction** on the Room master file to "Female Only."
- **The Simulation:** The Bed Planner bot (a background process) attempts to admit a Male patient with a "GI Bleed" to Bed 301-B because it is "Available" and the logic didn't forbid it.
- **The Result:** A Male patient is placed in a room with a Female OB patient. This is a massive privacy violation and patient safety risk.
- **Game Over Message:** *"CLINICAL FATAL: Privacy Breach. You admitted a Male patient to an un-restricted Obstetrics room occupied by a Female patient. You failed to configure 'Gender: Female' on the ROM master file. Grand Central Bed Logic failed to protect the patient."*

## Scenario 4: The "RTE Mapping" Fail (The Raw Data Trap)

- **The Setup:** The user configures the Real-Time Eligibility (RTE) interface. They successfully trigger a 270 Inquiry and get a 271 Response.
- **The Trap:** The 271 response contains the raw EB code EB*30 (Active Coverage - Health Benefit Plan Coverage). The user has not mapped EB*30 to an Epic **Benefit Group** in the RTE Interface Profile.
- **The Result:** The system shows "Eligibility Verified" in the history, but the **"Co-pay" field in the Registration form remains blank**. The data is stuck in "Raw Data" and didn't parse to the CVG record.
- **Game Over Message:** *"REVENUE FATAL: Point-of-Service Collection Failure. The 271 Response received code 'EB*30', but this code is not mapped to a Component Group (CMG). The registrar was not prompted to collect the co-pay. The data was lost in translation."*

## Scenario 5: The "Harvesting" Fail (The Accommodation Code Trap)

- **The Setup:** The user builds a new "Observation Unit." They create the Department and Beds.
- **The Trap:** They link the Beds to an **Accommodation Code** (fee schedule) for "Inpatient Med/Surg" instead of "Observation Hourly."
- **The Simulation:** A patient is admitted and discharged. The **ADT-Resolute Interface** runs the "Harvest" process to send charges to billing.
- **The Result:** The claim is generated with "Room & Board" charges (Inpatient) for a patient with status "Observation." This is a regulatory mismatch.
- **Game Over Message:** *"COMPLIANCE FATAL: Billing Fraud Risk. You linked an 'Inpatient' Accommodation Code to an 'Observation' bed. This caused the system to bill Room & Board charges for an Observation encounter. Audit triggered by Resolute Harvesting."*

# Chapter 5: Advanced Architect Concepts

## 5.1 Integrated vs. Non-Integrated CMGs (Study Guide)

One of the most complex architectural decisions in the Prelude/RTE build is the choice between Integrated and Non-Integrated Component Groups (CMGs). This distinction is vital for the PT candidate, as it determines how benefit data flows from the 271 response to the patient's file. The following table summarizes the differences, derived from Section 3.3 of the Mission Profile Audit. This table serves as the "Study Guide Flashcard" requested in the prompt.

**Table 1: Integrated vs. Non-Integrated Component Groups (CMGs)**

| Feature | **Integrated CMGs** | **Non-Integrated CMGs** |
|---|---|---|
| **Definition** | A unified set of benefit groups used for *both* RTE Display (Registration) and Backend Adjudication (Resolute Claims). | Separate sets of groups: one for RTE Display (Registration) and one for Backend Adjudication. |
| **Architecture** | **Tightly Coupled.** The data parsed from the 271 response flows directly into the benefit buckets used for claims calculation. | **Decoupled.** The 271 data is mapped to "Display-Only" buckets. The Claims system calculates benefits independently based on the contract build. |
| **Pros** | **High Fidelity.** Ensures that what the registrar quotes ("You owe $20") matches exactly what the system bills. Essential for "Riders" (add-on benefits). | **User Experience.** Allows for "User Friendly" grouping (e.g., grouping "Physical Therapy" and "Occupational Therapy" into one "Rehab" line for the registrar). Simpler to build/maintain. |
| **Cons** | **Complexity.** Very difficult to build and maintain. A change in the claims contract requires a change in the RTE mapping. One rigid structure for two different teams. | **Discrepancy Risk.** High risk that the quote ("$20") differs from the final bill ("$30") because the mappings are not synchronized. |
| **Best Use Case** | Organizations with mature, automated adjudication and complex benefit riders where quote accuracy is paramount. | Organizations prioritizing registrar ease-of-use and speed over 100% quote accuracy, or where contracts are too complex for front-end logic. |

## 5.2 Workflow Engine Rules (The Nervous System)

While the Master Files (EPT, DEP) are the skeleton, the **Workflow Engine (LOR)** is the nervous system. The simulator must implement the distinction between **CER** (Criteria) and **LOR** (Rule). This is a common point of confusion for trainees.

- **CER Rule:** Evaluates to True/False (e.g., Patient.Age < 18). This is the "Sensor."

- **LOR Rule:** Dictates UI behavior based on CER (e.g., IF CER(Under 18) = True THEN SET Navigator = Pediatric_Admission). This is the "Actuator."

In the simulator, this is the logic that swaps the "Adult Admission Navigator" for the "Pediatric Admission Navigator" when the patient's DOB is entered. The LOR effectively "listens" to the CER. If the CER returns "True" (Pediatric Context), the LOR instructs Hyperspace to load the specific Pediatric interface. This dynamic behavior is the hallmark of a well-built Epic environment.

Sources Cited: Mission Profile: Epic Systems Certification Audit MUMPS Global Structure Extreme Database Programming with MUMPS Globals Representing JSON Objects in MUMPS CSS Specificity as Analogy Facility Structure and Inheritance Logic Game Over Scenarios & EAF Hierarchy

## Works cited

1. Extreme Database Programming with MUMPS Globals, https://www.mumps.cz/gtm/misc/ExtremeDatabaseProgrammingWithMumpsGlobals.pdf 2. MUMPS Overview, https://iwayinfocenter.informationbuilders.com/TLs/TL_soa_app_mumps/source/intro9.htm 3. Representing JSON objects in MUMPS - Greg's Health IT Blog, https://gregshealthitblog.com/2016/08/30/representing-json-objects-in-mumps/ 4. Specificity - CSS - MDN Web Docs, https://developer.mozilla.org/en-US/docs/Web/CSS/Guides/Cascade/Specificity