Assignment Title — "Argos: A Federated, Adaptive Smart Campus Orchestration Platform"

Overview (short)

Design and implement Argos, a large-scale, extensible, object-oriented platform that orchestrates campus services (academics, facilities, security, analytics). The system must be modular, distributed, support real-time data, policy-driven access, adaptive behavior (runtime reconfiguration), formal verification for critical modules, and machine-learning–assisted decision components. Implementation language: choose one of Python — but design must be language-agnostic and demonstrate cross-language API boundaries (e.g., via gRPC/REST).

Core Learning Goals

- Advanced OOP: deep inheritance, composition, interfaces, generics/templates, reflection/meta-programming.
- Design patterns: plugin architecture, dependency injection, observer, strategy, factory, adapter, facade, mediator.
- Concurrency & distribution: thread safety, actor model or task queues, distributed consensus.
- Formal methods: model-checking or static proofs for critical invariants.
- Security & privacy: role-based access control, secure persistence, audit trail, encryption, privacy-preserving analytics.
- Data engineering: streaming events, time-series storage, aggregation, durable persistence.
- DevOps: containerization, CI/CD, automated tests, performance benchmarking, chaos testing.
- Research extension: adaptive policies, continual learning, fairness and explainability.

System Description (big-picture)

Argos manages:

- **Users** (Students, Staff, Lecturers, Admins, Guests) with dynamic roles.
- **Courses & Academics**: registration, waitlists, adaptive timetabling.

- **Facilities**: room booking, energy optimization.
- **Security**: access control to doors, cameras (simulated), incident logging.
- **Analytics**: personalized recommendations (courses, study groups), anomaly detection in access patterns.
- **Integration**: third-party services (payment gateway, external LMS), pluggable modules.

It must run as multiple services (microservices or modular monolith), communicate via RPC, and scale horizontally.

Mandatory Components & Requirements

1) Object Model & Core Classes (Design + Implement)

Design a **rich** class hierarchy with the following mandatory types (plus your own):

- AbstractEntity (base): universal ID, lifecycle, versioning.
- Person (abstract) → Student, Lecturer, Staff, Guest
    - Dynamic role attachments at runtime (a Student can be a TA).
- Credential & AuthToken: pluggable auth strategies (password, OAuth, certificate).
- Course, Section, Syllabus, Assessment, Grade (immutable grade objects).
- Facility, Room, Resource (sensors, actuators).
- EnrollmentPolicy (strategy pattern): multiple policies (prereq-check, quota, priority).
- Scheduler subsystem with Constraint objects (soft/hard), Timetable snapshotting.
- Event and EventStream classes for publish/subscribe.
- Plugin interface and PluginManager for hot-loading modules at runtime.
- AuditLogEntry, ComplianceChecker for immutable audit trail.
- Policy and PolicyEngine (policy evaluation in rules or DSL).
- MLModel wrapper (abstract) supporting train(), predict(), explain().

Requirements:

- Use encapsulation, well-documented public interfaces, and immutability where appropriate.
- Support versioning of entity schemas.
- Include at least **5 layers of inheritance** in one area, and **multiple orthogonal composition relationships** (has-a, uses-a, contains).

2) Concurrency & Distribution

- At least two services must run concurrently and communicate (e.g., EnrollmentService + SchedulerService).
- Implement **optimistic concurrency control** and **pessimistic locking** where suitable.
- Design an event-driven architecture using event sourcing for key subsystems (e.g., enrollment).
- Use thread-safe collections/structures. Provide a ConcurrencyStressTest module that spawns N clients with mixed operations and reports correctness under load.

3) Persistence & Data Model

- Persist data to disk in both a relational-like format (simulate with SQLite/Postgres or files) and an append-only event store (for Event Sourcing).
- Implement snapshotting and event replay to rebuild state.
- Data migration support: define a migration DSL and implement at least two migrations between schema versions.

4) API & Interoperability

- Expose APIs via gRPC (proto) and REST simultaneously (same business logic).
- Provide client SDKs (one minimal client in a second language — e.g., if main impl is Java, provide a Python client) that authenticate and perform complex flows.
- Implement API versioning and backward compatibility tests.

5) Security, Privacy & Compliance

- Role-Based Access Control (RBAC) + Attribute-Based Access Control (ABAC) for fine-grained rules.
- End-to-end encryption for sensitive fields (e.g., grades).
- Audit logs must be tamper-evident (append-only hash chain).
- Implement GDPR-like data-erasure: demonstrate safe deletion or pseudonymization of a Student while preserving analytics integrity.
- Penetration-test (automated tests simulating attacks: replay, injection, privilege escalation) and report results.

6) Reports & Policy Engine (Polymorphism & Interfaces)

- Implement a Reportable interface (or abstract class) with a generateReport(format, scope) method.
- Provide at least three report implementations: AdminSummaryReport, LecturerCoursePerformanceReport, ComplianceAuditReport.
- Reports must support output formats: JSON, CSV, PDF (PDF generation optional but extra credit).
- Use runtime polymorphism to plug new report types at runtime.

7) Exception Handling & Fault Tolerance

- Implement rich domain exceptions and a global error-handling strategy.
- Graceful degradation: if ML service is down, fall back to rule-based decision.
- Circuit-breaker pattern for failing external services.

8) Machine Learning Integration (Research-grade)

- Implement two ML components:
  - EnrollmentPredictor: predicts dropout probability per student (trained on synthetic dataset you produce).
  - RoomUsageOptimizer: suggests room assignments to minimize energy + travel time (formulate as optimization).
- Provide training pipelines, model versioning, and explainability hooks (explain() method).
- Ensure ML components expose deterministic behavior for unit tests (seed RNGs).

9) Formal Verification & Critical Invariants

- Pick a critical invariant (e.g., "no student can be enrolled in overlapping sections that are scheduled at the same time with the same seat allocation") and formally verify it using one technique:
  - Model checking with a tool (specify model and properties) or
  - Use assertions + invariants + runtime monitors + proof sketches.
- Provide the model/spec and demonstration of the verification (counterexample-free).

10) Testing, CI/CD & DevOps

- Unit tests, integration tests, property-based tests (for core invariants).
- A test harness to simulate 10,000 enrollment operations with concurrent users.
- CI pipeline: automated lint, tests, build, container image.
- Provide Dockerfile(s) and compose/k8s manifests for local deployment.
- Performance benchmark: measure throughput and latency under 1000 concurrent clients; present results.

Extra Challenges (pick at least two)

1. **Distributed Consensus:** Implement a replicated state machine for a small critical service using Raft or Paxos simulation.
2. **Hot Code Reload:** Allow plugin updates without restart, preserving existing sessions.
3. **Formal Contract:** Write and check design-by-contract annotations (pre/post/invariants) and demonstrate violations and fixes.
4. **Differential Privacy:** Add noise to analytics so aggregate queries are differentially private; measure utility vs privacy.
5. **Explainable AI:** Provide per-decision explanations for EnrollmentPredictor with LIME/SHAP-like logic.

Deliverables (concrete)

1. **Code repository** with clear package structure, README, build scripts. (Preferably on Git)
2. **UML diagrams:**
    - Class diagrams (detailed)
    - Component & deployment diagrams
    - Sequence diagrams for 3 critical flows (enrollment, grade assignment, emergency lockdown)
3. **Design Document (8–12 pages)**:
    - Architecture, patterns used, trade-offs, data model, deployment notes.
4. **Test Report**:
    - Unit/integration coverage, stress test outputs, CI results.
5. **Formal Verification Artifacts**:
    - Model/spec + results (counterexamples, proofs).
6. **Datasets**:
    - Synthetic dataset generator plus sample dataset (≥100k events).
7. **Performance Report**:
    - Benchmark methodology and raw/processed results.
8. **Security Report**:
    - Threat model, attack simulations, mitigations.
9. **Demo**:
    - A script that brings up the system locally and runs a demo scenario (end-to-end).
10. **Optional**: Docker/K8s deployment, GUI, mobile client.

Evaluation Rubric (total 200 points — intentionally huge)

- Architecture & OOP design — 30
- Correctness & invariants (incl. formal verification) — 30
- Concurrency, distribution & event sourcing — 20

- Persistence, migrations & snapshotting — 15
- Security, audit & privacy — 20
- ML components & explainability — 15
- Testing & CI/CD — 15
- Performance & scalability — 15
- Documentation & UML — 10
- Bonus: GUI, plugin hot-reload, distributed consensus — up to 30 extra

Grading Note (meta)

This assignment is intentionally open-ended and interdisciplinary — it requires systems design, software engineering, formal reasoning, and research. There is no single correct implementation; evaluation focuses on clarity of design, correctness of critical parts, test coverage, and the rigor of proofs/benchmarks. Candidates must justify design choices and demonstrate trade-offs.