Take Home Assignment

- **What strategies did you employ to test this project?**
  To ensure reliability and accuracy for this satellite imaging window builder project, I employed layer testing strategy which includes Unit testing, API testing, and Make based test automation.
  Unit Testing: Implemented comprehensive unit tests in `tests/test_unit.py` covering:
    - Core algorithm functionality (chronological sorting, streaming windows)
    - Edge cases and error conditions
    - Data validation and timestamp parsing
    - State-based window grouping logic

  API Testing : Used pytest with httpx for FastAPI endpoint testing:
    - Request/response validation
    - Error handling scenarios
    - Pydantic model validation
    - HTTP status code verification

  Based Test Automation : Set up `make test` commands for easy test execution and integration into development workflow.

- **How would those strategies differ as the project progressed?**
  As the satellite imaging project scales, testing strategies would evolve to support increasing reliability, complexity and performance expectations.
  Early stage(current): I will focus on unit tests for core algorithms such as time parsing and sorting logic, along with basic API endpoint testing to ensure input validation and response formatting.

  In the Growth stage, I will include integration testing for external API calls, performance testing to simulate thousands of imaging activities, and end-to-end workflow validation from activity ingestion to window generation.

  At the production scale, I will focus on monitoring and observability, implementing real-time testing and validation of satellite data streams.

- **How would you measure the performance of your service?**
  For the satellite imaging window builder service:
  1. Throughput: Activities processed per second which is critical for real-time constellation management
  2. Latency : Response time for window generation which must be under 100ms for real-time scheduling
  3. Memory Usage : To track the Efficiency when processing large satellite datasets
  4. Algorithm Efficiency: Analyzed time complexity for different window sizes (O(n log n) for sorting.

  Monitoring was implemented using FastAPI middleware for request/response timing, along with **Prometheus and Grafana** for real-time dashboard visualization. I will define **custom metrics** such as satellite activity processing rates and window generation success rates.

Benchmarking involved stress-testing the service with realistic datasets exceeding 1,000 imaging activities, measuring performance degradation as the fleet size scaled, and monitoring memory usage patterns during large-scale schedule generation.

- **How would you deploy this project to production?**

  For the deployment of the satellite imaging service, I will implement containerized architecture using **Docker** with multi-stage builds and configured health checks to monitor satellite connectivity. I will also ensure security by running containers as a non-root user.

  The service is orchestrated using **Kubernetes** for high availability and automatic scaling. I will manage **Helm charts** for configuration across environments and integrate a service mesh to enable secure communication between satellites and backend services.

  The deployment pipeline includes Docker image builds with semantic versioning, container security scans, automated test execution, and a two-phase deployment approach **blue-green for staging** and **canary deployments** in production for gradual rollout with traffic control.

  The infrastructure is cloud-native, supporting **AWS EKS**, **GCP GKE**, or **Azure AKS**, and includes **PostgreSQL** for persistent storage, **Redis** for caching, and load-balanced endpoints with active health checks to maintain service resilience under real-time operational demands.

- **Technologies for hosting (K8s, Docker, VMs, App Engine, Lambdas)**

  I would prefer using **Docker/ Kubernetes**, which offer features such as auto-scaling for satellite data bursts, rolling updates, and built-in service discovery. This approach is well-suited for handling variable workloads and managing multiple satellite connections efficiently.

  As an alternative, I will consider **Google Cloud Run or AWS Fargate** for serverless container hosting with automatic scaling to zero, which are cost-effective options for handling intermittent communication windows. However, these platforms come with a cold start latency, which may not meet strict real-time scheduling requirements.

  AWS Lambda was ruled out due to its 15-minute timeout limit and memory constraints, which are inadequate for processing large satellite constellations. Alternatively, we could use a streaming service and call lambda functions for the processing (still this would depend on the payload size).

  The infrastructure will be defined using **Terraform** for multi-cloud provisioning, Helm for Kubernetes templating, and ArgoCD to enable GitOps-based continuous deployment and environment synchronization.

- **Technologies for CI/CD (writing pipelines with automation)**

  For the satellite imaging service, I'll design a CI/CD pipeline to ensure safe, efficient, and mission-aware deployments. The pipeline includes staged steps:

  - lint: Code quality checks (black, pylint, mypy)
  - test: Unit and integration tests
  - security: Container vulnerability scanning

- build: Docker image creation
- deploy-staging: Automated staging deployment
- e2e-tests: End-to-end satellite simulation testing
- deploy-production: Manual approval for production

Tools like **GitHub Actions**, **GitLab CI**, or **Jenkins** can be used, with GitHub Actions preferred for tight repo integration and GitLab CI for enterprise control. For satellite-specific needs, I can integrate simulation environments to run automated tests before deployment. I will also coordinate deployments around scheduled satellite communication windows and set up reliable rollback procedures, since the system is mission-critical.

- **Release process**
  The release process for the satellite imaging service follows **semantic versioning**, where major versions represent breaking changes to imaging algorithms, minor versions introduce new satellite constellation support, and patch versions address bug fixes and performance improvements.

  The release workflow begins with feature branches for individual satellite feature development, followed by pull requests with code reviews using satellite data samples. After successful staging validation with real satellite scenarios, a release candidate is tagged and deployed to production. Post-deployment monitoring is conducted to verify the correctness of satellite activity processing.

  Release gates include a 100% pass rate on automated tests, performance benchmarks to avoid regressions, security scans with no critical vulnerabilities, and manual approval from the operations team. For safety, the rollback strategy includes backward-compatible database migrations, feature flags for toggling new features, and canary deployments with active monitoring of satellite activity processing.

- **What are some strategies you would employ to scale your project?**
  - **Vertical scaling vs. horizontal?**
  - **Microservice architecture vs. Monolith?**
  - **Adding some other infrastructure – queues for decoupling microservices**
  - **Improving the service in some way**

  To scale the satellite imaging window builder service, I will apply a hybrid strategy combining **horizontal and vertical scaling**. Horizontally, the system uses microservice decomposition (e.g., separate services for ingestion, window generation, notifications), load balancing, database sharding by satellite_hw_id, and caching with Redis.
  Vertically, I will optimize memory and CPU usage for large datasets and improved database performance with indexing. Algorithmically, the service supports both **real-time streaming** and **batch processing**, with optional use of approximate algorithms to trade accuracy for speed. The infrastructure leverages auto-scaling groups, regional deployments near ground stations, and CDNs for configuration data.

Compared to vertical scaling (simpler but limited), horizontal scaling provides better **fault tolerance and scalability**. A **hybrid model** is ideal—horizontal scaling for data handling and vertical for intensive algorithm execution.

While a **monolith** simplifies deployment and debugging, moving toward microservices **architecture** offers scalability, team autonomy, and fault isolation. Recommended services include data ingestion, window generation, configuration, notifications, and analytics.

To decouple components, I will introduce **event-driven messaging** with Kafka, Redis Pub/Sub, and AWS SNS/SQS. These queues support real-time updates, scalable buffering, and reliable processing with features like DLQs, message ordering, and retention policies. This architecture enables robust, scalable, and resilient satellite imaging operations.

- **Improving the service in some way**
  To make the satellite imaging service better, I'd enhance the core algorithms with machine learning for smarter window predictions and use optimization techniques to handle complex schedules and conflicts. I'd boost performance through caching, database tuning, parallel processing, and lean data structures. On the feature side, I'd add satellite health checks, weather-aware scheduling, priority handling for VIPs, and smarter resource use. Operationally, I'd set up live dashboards, strengthen automated tests, improve API docs, and tighten security with proper authentication and authorization.

- **How would your design change if you needed to store the uploaded images?**
  Design changes for image storage in satellite imaging system:
  1. The storage architecture: It uses object storage services like AWS S3 or Google Cloud Storage to store satellite images efficiently. To deliver images globally with low latency, a CDN such as CloudFront is employed. A database manages metadata that links imaging activities to the stored images, enabling quick access and organization. Additionally, data tiering will beimplemented, classifying images into hot, warm, or cold storage tiers based on how frequently they are accessed, optimizing both cost and performance.
  2. API Changes: The API will include an extended data model with fields for satellite hardware ID, imaging times, and optional image details like S3 URLs, metadata, and storage tier (hot, warm, cold).
  3. New services: It will support image processing (compression and format conversion), storage management with lifecycle and cost optimization, secure access control, and metadata cataloging with search capabilities.
  4. Storage Considerations: To ensure reliability, critical satellite images use multi-region replication, compression techniques reduce storage needs, and encryption protects data both at rest and in transit. Automated retention policies handle cleanup based on mission needs.
  5. Performance considerations include managing bandwidth impacts from large images, generating cached thumbnails for faster previews, and using asynchronous processing to separate image uploads from imaging window generation.

- **How would your design change if you needed to make the operation asynchronous?**

a. Polling - I'd keep it asynchronous by running the poll loop in a background task (like with asyncio or a worker queue), so the client just kicks it off and gets notified later.
b. WebSocket's - It's already asynchronous by design, the server pushes updates in real time, so the client doesn't have to keep asking. I'd just ensure robust reconnect and backoff to handle dropped connections.

- **What are the cost factors of your scaling choices and your new features? Which parts of your solution would grow in cost the fastest?**
  The fastest growing costs in scaling a satellite imaging system come from data storage, compute resources, and network bandwidth. Storage costs increase linearly with fleet size due to high-resolution images and retention needs, mitigated by tiered storage and compression. Compute costs grow with satellite activity volume, addressed through auto-scaling, spot instances, and algorithm optimization.
  Network costs rise with data transmission and are reduced using CDNs, compression, and edge computing. Cost optimization includes reserved and spot instances, serverless options, and regional deployments. Monitoring costs via resource tagging, budget alerts, and regular analysis ensures efficient satellite operation spending.

- **Where are your critical points of failure and how would you mitigate them?**
  So, the biggest risks in this system are around the database, the core algorithm service, and satellite communication. If the database goes down, we risk losing activity data. I'll rely on multi-AZ replication and restore regular backups to prevent any data loss. I'll also use read replicas to handle traffic spikes and maintain availability. If the core algorithm service fails, I'll monitor health checks closely and use circuit breakers to isolate any failing instances, ensuring scheduling continues smoothly with the remaining healthy nodes. For flaky satellite communications, I'll implement queues and retries to manage unstable connections and use caching to maintain a fallback schedule so the system can keep running even if live data drops.

- **Given a change to the algorithm what issues do you foresee when upgrading your scaled-out solution?**
  When we upgrade the algorithm, the big concerns are data consistency since old and new versions might compute different imaging windows plus any unexpected performance impacts. I'd handle that by doing blue-green or canary deployments, use feature flags to control rollout, and run shadow tests where the new algorithm processes live data but doesn't affect production. That way we can validate correctness and resource impact before fully cutting over.

- **In the age of cloud computing, is there ever a reason to self-host?**

  Definitely, especially in this kind of satellite project. If we're dealing with sensitive or classified data, compliance like ITAR might force us to keep it on-prem. Or if we have ultra-low latency needs near ground stations, self-hosting or hybrid edge compute can be more efficient. But otherwise, cloud still gives us the best balance of scale and cost.

- **If you wanted to migrate your scaled-out solution to another cloud provider (with comparable offerings but different API's) how would you envision this happening? How would deal with data consistency during the transition and rollbacks in the event of failures?**
  If I had to migrate our scaled-out solution to another cloud, I'd first make sure it's cloud-agnostic using containers, Terraform, and abstract APIs. I'd set up the new environment in parallel, start with read-only data migration, then use dual writes or event sourcing to keep data in sync. Traffic would gradually shift over, with replication and thorough monitoring to catch any issues. For consistency and rollback, I'd rely on point-in-time restores, event replays, and smart traffic routing like DNS or load balancers. Overall, it'd be a careful phased approach, running both setups side by side until fully confident, aiming for a smooth handover with minimal risk.

EICD Specific

- **Have you ever worked on a large spec?**
  Yes, I've worked on detailed, multi-team specs—like when building secure healthcare data pipelines at CVS. I broke them down into smaller stories, clarified gray areas early, and kept the team aligned to deliver on time.

- **How would you deal with ambiguity on a spec?**
  I'd first pin down unclear areas by asking targeted questions and mapping edge cases. Then I'd document assumptions and get quick feedback from stakeholders to ensure we're all on the same page before moving forward.

- **How do you communicate that across company boundaries?**
  I keep communication simple and structured sharing clear diagrams, short write-ups, and hosting quick syncs with external partners. I've done this with federal engineers and external vendors to tailor solutions to SLAs.

- **How would you debug a spike in latency in a downstream internal service?**
  I'd start by checking monitoring dashboards and logs to isolate where the latency starts. Then trace API calls, look at DB metrics, and run targeted load tests. Depending on findings, I'd tune queries, scale services, or patch code.

- **Tell me about a time when you/your team received changed requirements shortly before shipping?**
  Right before going-live at American Express, we got a last-minute requirement to add extra fraud checks for compliance. I quickly synced with product and security to nail down what was essential. Then I updated our microservices to call external fraud APIs and adjusted our Temporal workflows to handle new decisions. I also revised our CI/CD pipelines to run added security tests. By prioritizing the critical paths and planning follow-ups for less urgent pieces, we still hit our original launch date without sacrificing quality. The leadership appreciated how smoothly we handled the change under pressure.