Palantir Data Engineer Specialist Certification Exam Guide

1. Exam Coverage

The exam tests your ability to **build, operate, secure, and defend production data products** in Palantir Foundry, and own the decisions a pipeline depends on: ingestion pattern, storage layout, contract, orchestration, cost, and the trade-offs between them. You are graded on **design, decision, and diagnosis** (not mere recall), and on defending your approach in view of constraints.

You should be able to:

- Choose the ingestion pattern for a source, a volume, and a freshness requirement, and defend it against the adjacent one.
- Design a storage layout — keys, grain, format, partitioning, retention.
- Determine why a transform produced different rows on a rerun, or ran as a snapshot instead of incrementally.
- Define complex data expectations in code.
- Diagnose and fix a slow, failing, or expensive build.
- Change a live pipeline without breaking downstream consumers, and recover one that has already broken.

Format & Grading

During this **three-hour**, **hands-on** exam, you will build, extend, and stress-test real Foundry pipelines and data products in Ontologize's dedicated test environment. Question prompts include task guidance, resources, and acceptance criteria. You will be asked to plan, build, modify, and reflect on your solutions, with corresponding artifacts required for each of these activities. **AI FDE** will be unavailable during portions of the exam.

Ontologize administers and proctors the exam live over Zoom. After registering at ontologize.com/exams, you will receive additional instructions for participating in the exam.

Your work is graded on whether the required design components are present, whether what you build behaves correctly against functional expectations, and whether you can explain and defend your decisions. Points are weighted by domain (not all are equal) and task effort.

If you do not pass, your report shows your rubric performance by domain. Use your weakest domains to focus your next preparation, then return to the relevant sections of this guide and practice those skills in Foundry.

Role boundary

You own	You do not primarily own
Ingestion, transformation, and export pipelines	Ontology design
Storage layout, table format, keys, grain, and retention	Application UX, write-back, and automations
Orchestration, schedules, dependencies, and backfills	Enterprise policy ownership and shared IAM, secrets, and network operations
Shaping datasets that map to ontology requirements	Advanced BI storytelling and statistical interpretation
Data quality gates and expectations	AI and model lifecycle work
Pipeline monitoring, SLOs, incident response, and recovery	Operating shared observability platforms

2. Exam domains

The exam is organized around ten domains described in greater detail below. Each domain has an explicit boundary, and several exam traps sit exactly on those boundaries.

#	Domain
1	Source Connectivity, Ingestion, and Export
2	Storage, File Formats, and Table Design
3	Transformation and Pipeline Construction
4	Orchestration, Scheduling, and Dependency Management
5	Data Modeling, Semantic Models, and Basic Ontology Design
6	Data Quality, Testing, and Contract-Aware Publication
7	Data Product Observability, SLAs/SLOs, Incident Response, and Recovery
8	Governance, Security, Privacy, Metadata, and Lineage
9	Performance, Scalability, and Cost Diagnosis
10	Engineering Practice, Deployment, Documentation, and Stateful Change Management

Domain 1: Source Connectivity, Ingestion, and Export

Manage and justify the design of data flows in and out of Foundry:

- Choose among batch, incremental, CDC, and streaming syncs, virtual tables, exports, outbound webhooks, inbound Listeners, push streams, external transforms, functions, and Compute-Module-backed integration.
- Distinguish an outbound webhook from an inbound Listener or push stream.
- Design JDBC ingestion for large tables: monotonic incremental columns, query parallelization, full-sync feasibility, transaction type, fetch size, and compression.
- Checkpoint, replay, and recover a feed after a failure.

Self-check: Given a source system, a volume, and a freshness requirement, can I identify and justify the ingestion pattern?

Domain 2: Storage, File Formats, and Table Design

Optimize data storage for queries while remaining flexible enough to support schema changes:

- Decide the keys, data types, row grain, and retention at each layer.

- Choose among Foundry datasets, managed and virtual Iceberg tables, views, and filter-optimized projections.
- Partition and cluster for how the data will be read, since that is what queries pay for.
- Leave room for new columns, and account for incremental reads, row-level writes, and snapshots.

Self-check: When a new column arrives next quarter, does my process elegantly account for it?

Domain 3: Transformation and Pipeline Construction

Write transformations that produce expected outputs when rerun:

- Clean, parse, join, deduplicate, conform, and aggregate in modular, documented steps.
- Write incremental transforms using `added`, `current`, and `previous` reads, and the `snapshot_inputs`, `semantic_version`, `require_incremental`, `strict_append`, and `allow_retention` settings.
- Make a rerun produce the same result, or make any extra effects deliberate and visible.

Self-check: If this transform runs twice on the same input, what changes the second time?

Domain 4: Orchestration, Scheduling, and Dependency Management

Run pipelines on time and recover when they fail:

- Define schedules, task dependencies, retries, timeouts, and parameterized runs.
- Handle late-arriving data and time-zone issues.
- Run operational backfills and rerun windows without corrupting downstream state.
- Record what each run did, and send failures to someone who will act on them.

Self-check: When an upstream feed lands four hours late, what does my schedule do, and what do consumers see?

Domain 5: Data Modeling, Semantic Models, and Basic Ontology Design

Turn curated data into ontology types:

- Build dimensional and semantic models: entities, relationships, facts, dimensions, grain, keys, and slowly changing attributes.

- Model several object types together: how many rows each link joins, whether it uses a foreign key, a join table, or another object, and whether to materialize it.
- Model time series: decide between a time-series property and a sensor object, set the grain, and keep series IDs unique across syncs.

Self-check: Can I create object and link types based on well-scoped backing data pipelines and optimize ontology storage patterns?

Domain 6: Data Quality, Testing, and Contract-Aware Publication

Stop bad data before it publishes:

- Write checks that run before publication: column types, missing values, duplicates, broken references, allowed values and ranges, row counts, freshness, and rules the business depends on.
- Agree what consumers can rely on: stable keys, how late data is handled, duplicates, schema changes, retention, and what happens on failure.
- Decide the behavior on failure: warn, stop publication, quarantine, serve prior-good output, or notify consumers.

Self-check: When this check fails at 3am, what reaches consumers, and who finds out?

Domain 7: Data Product Observability, SLAs/SLOs, Incident Response, and Recovery

Run the pipeline in production, and deal with bad data the checks did not catch:

- Set targets for whether jobs run, how fresh the data is, and whether it is complete, then monitor against them.
- Notice when a schema changes, the row count jumps, or the values start looking different.
- Configure Monitoring Views, Data Health coverage, and freshness alerting.
- Use lineage to see who is affected, tell them, and decide whether to restore, rerun, or fix forward.
- Write post-incident notes that service project documentation.

Self-check: For a breached freshness SLO, can I say who is affected, what to restore, and in what order?

Domain 8: Governance, Security, Privacy, Metadata, and Lineage

Apply security requirements inside the assets you deliver:

- Label who owns each asset, how sensitive it is, and how long it is kept.
- Use markings and restricted views so people see only the rows and columns they should.
- Mask or remove identifying fields, and use Sensitive Data Scanner without creating extra copies of the data.
- Model so people get the least access that works, and send policy questions to whoever owns the policy.

Self-check: What are my options for mandatory and discretionary data access control?

Domain 9: Performance, Scalability, and Cost Diagnosis

Find and explain bottlenecks:

- Pick the compute engine and the right Spark profile for the data volume, the join, how much fits in memory, and how much is written out.
- Diagnose Spark precisely: driver and executor memory, broadcast joins, skewed keys, oversized tasks, and pulling too much back to the driver.
- Push work down to the source system where that avoids copying data.
- Weigh cost against speed, freshness, and reliability.

Self-check: Can I point at the signal that demonstrates the bottleneck, and identify where the fix should be applied?

Domain 10: Engineering Practice, Deployment, Documentation, and Stateful Change Management

Change a live pipeline without breaking its downstream consumers:

- Use version control, peer review, documentation, and automated release checks.
- Use the approved release pipelines, and package a product through Marketplace when others will install it.
- Change a schema without breaking processes that read it, adding the new before removing the old.
- Check lineage before changing anything, use Global Branching to change several resources together, and retire old workflows safely.
- Backfill history when a schema or a definition changes.

Self-check: Who/what consumes this asset today, and how do my changes affect them?

3. Working Methods

This section describes the “meta-domains” of the exam. It suggests strategies for building, design decisions, and troubleshooting. Practice using these frameworks when preparing for the exam.

The data engineering lifecycle

1. **Understand the consumer.** What decision or product depends on this data, and at what freshness?
2. **Choose the ingestion pattern** against source capability, volume, and freshness.
3. **Design the storage layout:** keys, grain, format, partitioning, and retention, planned for schema change.
4. **Build the transformation** in small steps that give the same result when run again.
5. **Write the contract:** keys, nullability, uniqueness, freshness, and failure behavior.
6. **Gate publication** with executable checks and a defined action on failure.
7. **Model** read-only object and link types.
8. **Schedule and orchestrate** with real dependencies, retries, and late-data handling.
9. **Instrument it:** SLIs, SLOs, alerts, and lineage for impact analysis.
10. **Change it safely:** branch, analyze consumers, migrate, and backfill.

Core decision areas

Use the source, the consumer, and the cost of getting it wrong to choose the simplest, defensible, safe design.

Decision	Choose based on	Preferred approach	Main risk or evidence to check
Ingestion pattern	Source capability, volume, freshness requirement, and whether the source can push	Batch or incremental sync for periodic loads; CDC when row-level change history matters; streaming when freshness is sub-minute	Confirm the source supports a monotonic column or change feed before promising incremental
Virtual table or materialize	Source load tolerance, latency, lineage needs, security,	Query virtually when the source can carry the load and lineage stays intact; materialize when you	A virtual table moves the failure into someone

Decision	Choose based on	Preferred approach	Main risk or evidence to check
	pushdown support, and downstream transform needs	need stability, speed, or heavy transformation	else's system, at their availability
Full or incremental transform	Input growth, whether history is immutable, and rerun cost	Incremental when inputs append and history is stable; full rebuild when late corrections rewrite the past	Check what invalidates the increment. A silent full rebuild at scale is a cost incident
Storage format	Read pattern, write pattern, schema-change frequency, and engine support	Foundry datasets for pipeline-internal work; Iceberg tables where snapshots, row-level writes, or external engines matter	Verify the features you rely on are supported before designing around them
Check placement	Whether bad data must never publish, or must be caught after it does	Pre-publication gates for contract violations; runtime monitoring for behavior that only appears in production	A check that only warns is not a gate. Decide the failure behavior explicitly
Serving model	How the data will be queried, how many rows a link joins, and who reads it	Model object types and link types against the read pattern, and materialize where join cost warrants	Shared properties, value types, and interfaces are outside this track's scope
Change strategy	Number of consumers and whether the change is compatible	Expand and contract for stateful change; Global Branching to coordinate across resources	Check lineage for consumers before, not after. A compatible change is cheaper than a migration

Troubleshooting playbook

1. **Classify the symptom:** stale output, failed build, wrong values, missing rows, slow query, cost spike, or permission error.
2. **Establish when it last worked.** Use Time Travel across dataset transactions or Iceberg snapshots to locate the change.
3. **Separate data change from code change.** Compare the input transaction and the `semantic_version` before reading the logic.
4. **Check the schedule and its dependencies:** stale upstreams, skipped runs, retry exhaustion, and partial writes or aborts.
5. **Read the contract:** schema drift, nullability, key collisions, duplicates, and late arrivals against what the transform assumed.
6. **For performance, get the signal before the theory:** skew, shuffle, broadcast, memory pressure, and driver collection.
7. **Assess impact through lineage** before repairing, and identify affected consumers.
8. **Recover:** restore, rerun, or roll forward, then reconcile row counts and notify consumers.
9. **Write the root-cause note,** naming the domain that owns the permanent fix.

Avoid the impulse to simply add memory, widen a schema, or disable a check. These mask defects, deferring the cost further downstream.

4. Practice scenarios

1. **An incremental transform silently rebuilds in full each night.** Inspect what invalidates the incremental build, including a changed `semantic_version` or a non-append upstream write.
2. **A downstream object shows duplicate records after a source migration.** Check primary-key stability in the contract; the source likely reissued keys that were previously stable.
3. **A Spark job fails only on month-end volumes.** Look for skew and broadcast-join thresholds before increasing executor memory.
4. **A freshness SLO breaches because an upstream feed lands late twice a week.** Handle late arrival in the schedule and contract; alerting on the symptom does not deliver the data.
5. **A quality check has been warning for three weeks and nobody acted.** Decide the failure behavior: stop publication, quarantine, or serve prior-good output. A permanent warning is not a gate.

6. **A schema change breaks two of nine downstream consumers.** Analyze lineage first, then run expand and contract.
7. **Costs triple after a virtual table is pointed at a busy source.** Re-examine the virtualize-or-materialize call against source load and pushdown support.
8. **A restricted dataset gets copied into an unclassified derived table.** Trace lineage, apply marking-based policy at the source, and remove the copy. Classifying it after the fact leaves the exposure in place.

Additional hands-on practice:

Build a pipeline from an external source through to an object type. Ingest incrementally with a monotonically increasing column, land it in a deliberate storage layout, transform it in modular steps, and gate a successful build with checks for uniqueness, nullability, and freshness. Configure and link two object types. Then break it by duplicating a primary key, adding a column upstream, or by skewing the data until a join fails. For each one, diagnose, decide whether the fix is in storage, transformation, schedule logic, or elsewhere, and recover using Time Travel and a rerun.

Capability review checklist

- ☐ I can choose an ingestion pattern and say why the adjacent one is wrong.
- ☐ I can design JDBC ingestion for a large table with a monotonic incremental column.
- ☐ I can decide virtual table versus materialize on source load, lineage, security, and pushdown.
- ☐ I can design a storage layout that absorbs schema evolution.
- ☐ I can write an incremental transform and state what triggers a full rebuild.
- ☐ I can build a streaming pipeline and pick the right transaction semantics.
- ☐ I can define a data contract covering keys, nullability, freshness, and failure behavior.
- ☐ I can model and configure ontology object and link types.
- ☐ I can place a quality check and choose its behavior on failure.
- ☐ I can set an SLO and run a recovery when it breaches.
- ☐ I can diagnose a Spark failure to a named cause.
- ☐ I can run an expand-and-contract migration across multiple consumers.
- ☐ I can use Time Travel to locate when data or schema changed.