Palantir Applied AI Engineer Specialist Certification Exam Guide

1. Exam Coverage

The exam tests whether you can **design, build, evaluate, secure, and defend production AI systems** in Palantir Foundry & AIP, and own the decisions an AI system depends on: architecture, placement, risk, scale, cost, and the trade-offs between them. It goes beyond building to the routine: you are graded on **design, decision, and diagnosis** (not mere recall), and on defending your approach in light of constraints. You build working systems, evaluate them, and defend your decisions.

You should be able to:

- Choose the model, the adaptation approach, and the architectural placement for a workload, and defend the choice.
- Architect retrieval and grounding that return relevant, current, authorized evidence with citations.
- Design tool use and bounded autonomy with least-privilege permissions and staged actions.
- Build evaluation suites and release gates that verify quality.
- Diagnose failures from traces, retrieval results, evaluations, logs, and observed behavior.
- Keep humans in control of high-consequence actions, and know when a simpler solution is the correct one.

Format & Grading

During this **three-hour**, **hands-on** exam, you will build, extend, and stress-test real Foundry AI workflows 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
The hard AI-system decisions: architecture, placement, risk, scale, and trade-offs	Core ontology design and data pipelines
Advanced retrieval, agents, evaluation, security, and cost/performance at scale, end to end	Training or fine-tuning foundation models from scratch
Extending an existing ontology for a workflow: vector properties, function-backed Actions, agent tools	Redefining the core object and link contracts for the enterprise
What ships and defending the trade-offs	Unbounded authority for an automated system

2. Exam domains

The exam is organized around nine domains described in greater detail below.

#	Domain
1	Foundation Model Selection and Adaptation Strategy

#	Domain
2	Prompt, Context, and Output Contract Engineering
3	Retrieval, RAG, and Grounded Generation
4	Tool Use, Workflow Orchestration, and Bounded Autonomy
5	AI-Augmented Data Pipeline Integration
6	AI Application and Runtime Integration
7	Conversational Applications, State, and Memory
8	Evaluation, Release, and Change Control
9	Production Observability, Safety, Cost, and Latency Stewardship

Domain 1: Foundation Model Selection and Adaptation Strategy

Own the model and adaptation decision, and defend it against the nearest alternative:

- Compare models on capability, latency, cost, context window, modality, and structured-output reliability.
- Decide among prompting, retrieval, long context, and lightweight adaptation for a given task.
- Map task requirements to the models available in the Model Catalog and to the authoring surface, such as AIP Logic.
- Decide when a workload should require structured output, and express it as a typed Struct contract.

Self-check: Given a feature, can I choose the model and the adaptation approach and defend it against alternatives on capability, cost, and latency?

Domain 2: Prompt, Context, and Output Contract Engineering

Engineer the model call and its output contract to work reliably in production:

- Design system and task prompts, examples, and refusal guidance in AIP Logic.
- Design a typed Struct output. In AIP Logic the platform coerces the model to the Struct schema or fails the node, so the engineering focus is not format repair but

semantic validation: check that the returned values are in range, grounded, and consistent, and fall back or flag when they are not.
- Assemble and trim non-retrieved context to fit token limits without losing signal.

Self-check: When a function returns a well-formed but wrong or ungrounded value, can I add a validation step that catches it and falls back, rather than assuming the schema alone makes the output trustworthy?

Domain 3: Retrieval, RAG, and Grounded Generation

Build honest grounding that returns relevant, authorized, cited evidence and abstains or flags a conflict when it cannot answer:

- Design chunking, embeddings, a vector property, and semantic search, with hybrid (keyword plus vector) retrieval, reranking, and metadata filtering.
- Recognize when source content is not plain text: a value locked in a chart, figure, or scanned image needs a vision model to extract at embed time, because text extraction and text embeddings will miss it.
- Enforce authorized, permission-aware retrieval and expose provenance and citations.
- Define no-answer behavior and diagnose where retrieval quality breaks down.
- Flag a conflict when retrieved guidance contradicts the live data, and trust the current telemetry over a stale document.

Self-check: Given irrelevant or incomplete retrieval, can I identify the failing stage (chunking, embedding, ranking, threshold, context selection, or generation) and say how I would measure the fix?

Domain 4: Tool Use, Workflow Orchestration, and Bounded Autonomy

Build tool-using, multi-step systems in AIP Logic and AIP Chatbot Studio, and grant no more autonomy than the task earns:

- Design tool and argument schemas, permissions, timeouts, retries, routing, chaining, and evaluator loops.
- Decide between a predefined workflow and agentic autonomy, and scope each call to the minimal context and tools.
- Stage high-consequence actions for human approval, for example a function-backed Action set to ask for approval before execution.

Self-check: Can I explain why this workflow needs these tools and this level of autonomy, and what it should do when evidence is insufficient?

Domain 5: AI-Augmented Data Pipeline Integration

Insert model calls into batch, streaming, and event-driven workflows, such as pipelines in Pipeline Builder:

- Configure extraction, classification, enrichment, and summarization, including visual document processing.
- Preserve prompts, raw responses, and provenance for audit and debugging.
- Decide whether computation belongs south of the ontology for reusable batch enrichment or north for interactive reasoning, and handle rate limits, concurrency, and review queues.

Self-check: Given a workload, can I justify the processing layer using consumer breadth, freshness, reusability, scale, and cost?

Domain 6: AI Application and Runtime Integration

Invoke models securely from applications and services:

- Design synchronous, asynchronous, and streaming patterns with timeouts, cancellation, and idempotent retries.
- Choose the client-facing versus backend or confidential-service boundary, and keep secrets and orchestration server-side.
- Serve cost-aware and degrade gracefully under failure.

Self-check: Can I design an invocation path that stays correct and safe when a model call, Action, or external dependency fails or is retried?

Domain 7: Conversational Applications, State, and Memory

Design multi-turn conversation, session state, and memory in AIP Chatbot Studio:

- Design session state with application variables, history trimming, and summarization.
- Decide what state is ephemeral and what must persist, and where each belongs.
- Architect durable, cross-session memory as an ontology pattern, because AIP Chatbot Studio has no built-in cross-conversation memory. Persist selectively (extract the durable facts, drop the noise), avoid duplicates, supersede entries that have gone stale, and require human confirmation on high-consequence writes.
- Design clarification, correction, uncertainty display, and human handoff.

Self-check: Can I keep a multi-turn assistant coherent across a session and hand off cleanly when it reaches the limit of what it should do?

Domain 8: Evaluation, Release, and Change Control

Produce durable evidence that the system works. Build the eval suite, gate the release, and treat evaluation as a requirement:

- Build AIP Evals suites, test cases, evaluators, and metrics covering normal, edge, adversarial, unauthorized, and high-consequence cases.
- Know the built-in evaluator types and choose the right one per output field: exact string match, regex, and keyword checks for enums, IDs, and citations; LLM-as-a-judge for grounded free text; and length or similarity measures (string length, Levenshtein, ROUGE) where they fit. Reserve LLM-as-a-judge for what a deterministic check cannot verify, and configure the results dataset so runs are recorded.
- Distinguish signal from non-deterministic variance and detect regressions across prompts, models, and versions.
- Track versions and configuration, gate releases, and design feedback loops that turn user corrections into new evaluation cases.

Self-check: If a result changed between versions, can I determine whether it is a genuine regression, expected variance, or a change in the test distribution, and gate the release accordingly?

Domain 9: Production Observability, Safety, Cost, and Latency Stewardship

Steward the running system, and defend it against adversarial and second-order failures:

- Use AIP Observability (distributed traces, execution history, logs, token usage, and P95 latency) to find bottlenecks.
- Apply safety filters, prompt-injection defenses, privacy-aware logging, and audit evidence, and defend against injection carried in retrieved content or tool output as well as in the user's message.
- Treat cost and latency as a go/no-go gate, using caching, batching, and routing before reaching for a larger model.

Self-check: Given a trace and a cost or latency trend, can I identify the bottleneck and recommend a targeted optimization rather than simply choosing a larger model?

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 AI engineering lifecycle

1. **Define the task.** What outcome is required, who is the user, and what counts as a correct answer or action?
2. **Assess risk.** Is the output informational, advisory, or capable of changing external state?
3. **Choose the pattern and placement.** Function, retrieval workflow, agentic workflow, or deterministic orchestration; north or south of the ontology; batch or interactive.
4. **Assemble context.** Retrieve relevant, current, authorized evidence with provenance.
5. **Constrain tools.** Give the system only the tools, data, and Actions the task requires.
6. **Generate or act.** Produce an answer, recommendation, draft, or controlled state change.
7. **Evaluate.** Measure correctness, groundedness, refusal behavior, tool selection, latency, and cost.
8. **Operate.** Monitor real behavior, regressions, failure modes, and feedback.
9. **Improve safely.** Change prompts, retrieval, tools, or models with regression checks and appropriate review.

A polished demo is not evidence of a production-ready AI system. The exam rewards designs that are elegant, secure, testable, and observable.

Core decision areas

Use the task, the evidence available to the system, and the consequence of error to choose the simplest, defensible, safe design.

Decision	Choose based on	Preferred approach	Main risk or evidence to check
Model and adaptation	Capability, cost, latency, modality, structured-output need	Match the model and adaptation approach to the task; start capable, then optimize down	Justify against the nearest alternative; verify structured-output reliability

Decision	Choose based on	Preferred approach	Main risk or evidence to check
Architectural placement	Consumer breadth, freshness, reusability, scale, cost	Put reusable enrichment south of the ontology; keep context-dependent reasoning north	Wrong side causes stale batch outputs or redundant per-user compute
Deterministic logic or AI	Whether the task is fixed or requires interpretation	Use logic for explicit rules, AI for bounded interpretation, and combine where useful	Narrow the AI step and validate its output; ask whether errors are tolerable or dangerous
Retrieval and grounding	Relevance, coverage, freshness, provenance, authorization, token budget	Retrieve current, authorized evidence and expose provenance	Inspect retrieved context, not only the final answer; more context can add noise or unauthorized data
Tool use and autonomy	Read versus write, side effects, consequence of error	Give the workflow only the tools it needs; separate recommend, prepare, and execute	Check authorization, idempotency, retries, and duplicate handling; stage high-consequence actions
Evaluation and change control	Quality criteria, non-determinism, drift	Use representative and adversarial cases; gate releases; feed corrections back	Measure groundedness, refusal behavior, safety, latency, and cost; distinguish regression from variance
Cost and latency	Interaction needs, call count, context size, scale	Simplify calls, trim context, cache where safe, use asynchronous	Profile the trace before choosing a larger model or more autonomy

Decision	Choose based on	Preferred approach	Main risk or evidence to check
		execution for long work	
Engineering restraint	Whether the extra machinery earns its keep	Prefer a plain function, prompt change, or existing pattern when it suffices	Justify an agent, custom framework, or compute module against cost, maintainability, and risk

Troubleshooting playbook

1. **Classify the failure:** wrong answer, unsupported claim, missing evidence, irrelevant context, unsafe action, refusal, timeout, or cost regression.
2. **Inspect the trace:** prompt and input, retrieved context, tool calls, model output, validation, and final action.
3. **Check authorization:** confirm what the user and execution identity were allowed to retrieve or change.
4. **Separate stages:** retrieval, reasoning, tool selection, execution, and presentation can fail independently.
5. **Compare to a baseline:** use a known-good prompt, retrieval configuration, model, or evaluation version.
6. **Make the smallest change:** fix filtering, metadata, prompt constraints, tool schema, validation, or workflow control.
7. **Run regression evaluations:** check quality, safety, latency, cost, and affected use cases.
8. **Escalate when needed:** involve an ontology, data, or security owner for changes outside your boundary.

4. Practice scenarios

1. **An assistant gives a confident answer unsupported by the source material.** Inspect retrieved context and require grounded evidence or an explicit uncertainty and refusal path; do not merely raise temperature or prompt length.

2. **Retrieval returns a record the user is not authorized to see.** Fix access control at the retrieval and data boundary and verify the execution context; do not rely on the model to redact it.
3. **An agent repeats a write Action after a timeout.** Add idempotency, operation status, and execution safeguards before enabling retries.
4. **Quality drops after adding more retrieved context.** Inspect relevance, ranking, chunking, freshness, and conflicting evidence; more context is not automatically better.
5. **A workflow is accurate but too slow for an interactive user.** Profile retrieval and model calls, simplify or parallelize safe steps, trim context, or move long work to an asynchronous flow with visible status.
6. **A request asks the system to take a high-consequence action on ambiguous evidence.** Constrain or refuse execution, surface uncertainty, and route to human review rather than maximizing autonomy.
7. **A stakeholder asks for an agent where a single function would do.** Recommend the simpler design and defend it on cost, maintainability, and risk; reserve the agent for where its autonomy is warranted.

Additional hands-on practice:

Build a small retrieval-backed assistant in AIP Logic and AIP Chatbot Studio over secured business data, with one read-only tool and one controlled Action. Ground its answers in a source document you chunk and embed yourself, and give it an abstain path for when the evidence is weak. Then stress it with stale, irrelevant, conflicting, or unauthorized evidence and an ambiguous high-consequence request, and build an AIP Evals suite that measures the system before and after each fix. Use a plain function wherever an LLM is not needed. Throughout, practice explaining your evidence, controls, metrics, and trade-offs.

Capability review checklist

- ☐ I can choose a model and adaptation approach and defend it against alternatives.
- ☐ I can make the north-versus-south-of-the-ontology placement call and justify it.
- ☐ I can diagnose whether a bad answer came from retrieval, reasoning, tool selection, or execution.
- ☐ I can design permission-aware retrieval and expose provenance and citations.
- ☐ I can design least-privilege tools, bounded autonomy, and staged high-consequence actions.
- ☐ I can design evaluations for normal, ambiguous, adversarial, unauthorized, and high-consequence cases, and gate a release.
- ☐ I can treat cost and latency as a gate and optimize before reaching for a larger model.

- [ ] I know when to require human confirmation or escalation.
- [ ] I can recognize when a simpler solution is correct and defend not building the heavier one.
- [ ] I can explain why the nearest alternative is weaker under the stated constraints.