CAMPUS RECRUITMENT 2026

AI-Assisted Technical Assessment

NexaPay Payment Gateway Engineering Challenge

Duration	4 Hours (240 minutes)
Total Marks	200
AI Tools Allowed	Yes - ChatGPT, Claude, Copilot, any AI assistant
Internet Access	Allowed for documentation only (no copying solutions)
Languages	Elixir (mandatory for Sections B, C, D)
Source Document	NexaPay Technical Specification (provided, 500+ pages)
Submission	Git repository with code + decision_log.md

IMPORTANT INSTRUCTIONS:

1. You are expected to use AI tools throughout this assessment. This is not a test of whether you can code from memory - it is a test of whether you can **think, decompose, verify, and architect solutions** using AI as a tool.
2. You MUST maintain a **decision_log.md** file documenting: (a) what you asked AI, (b) what AI got wrong and how you fixed it, (c) why you chose your approach over alternatives. **This file carries 30 marks.**
3. The source document contains **intentional contradictions, hidden dependencies, and implicit rules**. Discovering and resolving these is part of the assessment.
4. You cannot complete all sections perfectly in 4 hours. **Prioritization is being evaluated.** Show working progress over polished incompleteness.
5. A 10-minute live walkthrough will follow submission. You must explain your approach verbally.

MARKING SCHEME OVERVIEW

Section	Title	Marks	Focus Area	Time Guidance
A	Document Analysis & Pattern Discovery	40	Analytical thinking, contradiction detection	45 min
B	Data Modeling & Ecto Schemas	30	System design, Elixir/Ecto knowledge	30 min
C	Web Server with Multilayer Validation	50	Elixir web dev, validation architecture	75 min
D	Async Pub/Sub & Event System	30	Event-driven design, concurrency	40 min
E	Mutex & Concurrency Control	20	Distributed systems, race conditions	30 min
F	Decision Log & Walkthrough	30	Problem-solving process, communication	20 min
	TOTAL	**200**		**240 min**

SECTION A: DOCUMENT ANALYSIS & PATTERN DISCOVERY

[40 Marks | Recommended: 45 minutes]

This section tests your ability to extract, synthesize, and resolve information from a large, complex technical document. You will need to navigate the 500+ page NexaPay specification and identify key patterns, contradictions, and hidden rules.

Question A.1: Transaction Validation Rules Discovery [15 marks]

From the source document, identify ALL validation layers that a transaction must pass through before being authorized. For each layer, provide:

(a) The layer name and its purpose [2 marks per layer]

(b) The execution order and WHY they are ordered that way [3 marks]

(c) The error code prefix for each layer [2 marks]

Deliverable: A structured summary in your decision_log.md under heading "## Validation Layer Analysis"

Question A.2: Contradiction Detection [15 marks]

The source document contains at least THREE intentional contradictions or inconsistencies. Find and document them.

For each contradiction found:

(a) State what the contradiction is, citing the relevant sections [3 marks each]

(b) Propose a resolution strategy explaining which value/approach you will use in your implementation and why [2 marks each]

> ***Hints:*** *Look at timeout values, KYC status enumerations, and entity relationship definitions across different chapters. The contradictions are NOT typographical errors - they represent real-world scenarios where different teams documented different behaviors.*

Deliverable: A structured list in decision_log.md under heading "## Contradictions Found"

Question A.3: Hidden Dependency Discovery [10 marks]

Identify the many-to-many relationship that is described only in prose (never as a formal table or ERD) in the source document.

(a) Which two entities have this relationship? [3 marks]

(b) What additional fields does the junction/association carry beyond simple foreign keys? List at least 4. [4 marks]

(c) How does this relationship affect transaction validation? Cite the specific validation rule. [3 marks]

Deliverable: Documented in decision_log.md under "## Hidden Dependencies"

SECTION B: DATA MODELING & ECTO SCHEMAS

[30 Marks | Recommended: 30 minutes]

Question B.1: Core Ecto Schemas [20 marks]

Using the entity specifications discovered in Section A, implement Ecto schemas for the following entities in Elixir:

1. **Merchant** - with all fields from the source document, proper types, and validations [5 marks]
2. **Transaction** - with state machine status field, all relationships, and generated fields [5 marks]
3. **PaymentMethod** - with proper enum types and constraints [3 marks]
4. **MerchantPaymentMethod** - the junction table you discovered in A.3, with all association fields [5 marks]
5. Ecto migrations for all schemas [2 marks]

Requirements:

- Use Ecto changesets with proper validation (cast, validate_required, validate_inclusion, etc.)
- Handle the KYC status contradiction from A.2 (accept both legacy and new enum values)
- Include @doc annotations explaining design decisions

Question B.2: State Machine Validation [10 marks]

Implement a module **NexaPay.Transaction.StateMachine** that:

(a) Defines all valid state transitions from the source document [4 marks]
(b) Provides a function **valid_transition?(from_state, to_state)** that returns true/false [3 marks]
(c) Provides a function **transition!(transaction, new_state, params)** that validates and applies the transition, returning {:ok, updated_txn} or {:error, reason} [3 marks]

```
# Expected behavior:
StateMachine.valid_transition?(:initiated, :processing) # => true
StateMachine.valid_transition?(:captured, :processing) # => false
StateMachine.valid_transition?(:initiated, :settled) # => false
```

Deliverable: Elixir source files in lib/nexapay/schemas/ and lib/nexapay/transaction/

SECTION C: WEB SERVER WITH MULTILAYER VALIDATION

[50 Marks | Recommended: 75 minutes]

This is the core implementation section. Build a Phoenix (or Plug-based) web server that implements the NexaPay transaction creation API with the multilayer validation framework described in the source document.

Question C.1: API Endpoint Implementation [15 marks]

Implement **POST /api/v1/transactions** that:

(a) Accepts the request body format specified in the source document (Chapter 62) [3 marks]
(b) Requires X-Api-Key and Idempotency-Key headers [2 marks]
(c) Returns the standard NexaPay JSON response envelope for both success and error cases [5 marks]
(d) Returns appropriate HTTP status codes: 201 (created), 400 (validation error), 401 (auth error), 409 (idempotency conflict), 422 (business rule violation), 429 (rate limit), 500 (server error) [5 marks]

Question C.2: Multilayer Validation Pipeline [25 marks]

Implement the 5-layer validation pipeline. Each layer must:

(a) Run in the correct order as specified in the source document [5 marks]
(b) Short-circuit on first failure (do not run subsequent layers) [3 marks]
(c) Return structured error information including the layer name, error code, and details [5 marks]

Minimum implementation per layer:

Layer	Minimum Validations Required	Marks
1. Schema	Required fields check, type validation, amount > 0, format validation for email/phone	4
2. Entity	Merchant exists + active, PaymentMethod exists + active, Merchant-Method association active	4
3. Business Rule	Amount within min/max for payment method, per-txn limit check, KYC tier limit check	4
4. Compliance	At least 1 compliance check (e.g., amount reporting threshold of 200,000 INR)	2
5. Risk	At least 1 risk check (e.g., velocity check - reject if > 10 txns in 5 minutes)	2

Question C.3: Error Response Format [10 marks]

The error response MUST follow this exact JSON structure:

```json
{
  "success": false,
  "error": {
    "code": "SCHEMA_MISSING_FIELD",
    "message": "Required field 'amount' is missing",
    "layer": "schema",
    "details": [
      {"field": "amount", "rule": "required", "message": "is required"}
    ]
  },
  "metadata": {
    "request_id": "req_abc123",
    "timestamp": "2026-01-15T10:30:00.000Z",
    "version": "v1"
  }
}
```

Marks are awarded for: correct structure [3], layer identification in error [3], detailed field-level errors [2], proper metadata [2]

SECTION C: TEST CASES

The following test cases will be used to evaluate your implementation. Your code must handle ALL of these scenarios correctly.

Test Case C.TC1: Happy Path - Successful UPI Transaction

```
# Request
POST /api/v1/transactions
Headers: X-Api-Key: "test_key_merchant_001", Idempotency-Key: "idem_001"
Body: {
  "amount": 1500.00, "currency": "INR", "payment_method": "upi",
  "reference_id": "ORDER-001",
  "customer": {"email": "test@example.com", "phone": "+919876543210"}
}

# Expected: 201 Created with success: true, status: "processing"
```

Test Case C.TC2: Schema Validation Failure - Missing Amount

```
# Request: same headers, body missing "amount" field
Body: {"currency": "INR", "payment_method": "upi", "reference_id": "ORDER-002"}

# Expected: 400 Bad Request
# error.code: "SCHEMA_MISSING_FIELD", error.layer: "schema"
```

Test Case C.TC3: Schema Validation - Negative Amount

```
Body: {"amount": -500.00, "currency": "INR", "payment_method": "upi", ...}

# Expected: 400, error.code: "SCHEMA_INVALID_AMOUNT", error.layer: "schema"
```

Test Case C.TC4: Entity Validation - Inactive Merchant

```
# X-Api-Key maps to a merchant with onboarding_status: "review" (not "activated")

# Expected: 403, error.code: "ENTITY_MERCHANT_INACTIVE", error.layer: "entity"
```

Test Case C.TC5: Entity Validation - KYC Not Approved

```
# Merchant with kyc_status: "pending" (legacy) or "not_started" (new system)

# Expected: 403, error.code: "ENTITY_MERCHANT_KYC_INVALID", error.layer: "entity"
# NOTE: Both "verified" (legacy) AND "approved" (new) should be accepted as valid
```

Test Case C.TC6: Business Rule - Amount Exceeds Payment Method Max

```
# UPI max is 200,000 INR (from source doc)
Body: {"amount": 250000.00, "payment_method": "upi", ...}

# Expected: 422, error.code: "RULE_AMOUNT_ABOVE_MAX", error.layer: "business_rule"
```

Test Case C.TC7: Business Rule - Amount Below Minimum

```
# Credit card min is 100.00 INR
Body: {"amount": 50.00, "payment_method": "credit_card", ...}

# Expected: 422, error.code: "RULE_AMOUNT_BELOW_MIN", error.layer: "business_rule"
```

Test Case C.TC8: Compliance - Large Transaction Flagging

```
Body: {"amount": 250000.00, "payment_method": "netbanking", ...}

# Expected: 201 (transaction succeeds but is FLAGGED, not blocked)
# Response should include metadata.compliance_flags: ["AMOUNT_REPORTING"]
```

Test Case C.TC9: Idempotency - Duplicate Request

```
# Send same request twice with same Idempotency-Key
# First request: 201 Created
# Second request with SAME key and SAME body: 200 OK (return cached response)
# Second request with SAME key but DIFFERENT body: 409 Conflict
```

Test Case C.TC10: Authentication - Missing API Key

```
# Request without X-Api-Key header

# Expected: 401 Unauthorized
```

SECTION D: ASYNC PUB/SUB & EVENT SYSTEM

[30 Marks | Recommended: 40 minutes]

Question D.1: Event Publisher [10 marks]

Implement a module **NexaPay.Events.Publisher** that:

(a) Publishes domain events when transactions change state (use Phoenix.PubSub) [4 marks]

(b) Events follow the envelope schema from the source document (event_id, event_type, version, timestamp, source, correlation_id, data, metadata) [3 marks]

(c) Supports hierarchical topic patterns as defined in the source document [3 marks]

```
# Expected usage:
Publisher.publish("transaction.authorized", %{
  transaction_id: txn.id,
  merchant_id: txn.merchant_id,
  amount: txn.amount
})
# This should broadcast to topics: "txn:transaction:authorized:*" and "txn:*"
```

Question D.2: Event Subscriber with Error Handling [10 marks]

Implement a module **NexaPay.Events.NotificationHandler** that:

(a) Subscribes to transaction.authorized and transaction.failed events [3 marks]

(b) Implements the NexaPay.EventHandler behaviour with handle_event/1 and handle_error/2 callbacks [3 marks]

(c) Simulates sending a webhook notification to the merchant's webhook_url (log the payload, don't actually make HTTP calls) [2 marks]

(d) Implements back-pressure: if the GenServer mailbox exceeds 100 messages, drop non-critical events and log a warning [2 marks]

Question D.3: Dead Letter Queue Pattern [10 marks]

Implement a module **NexaPay.Events.DeadLetterQueue** that:

(a) Captures events that fail processing after 3 retry attempts [3 marks]

(b) Uses exponential backoff for retries (1s, 5s, 30s as specified in source doc) [3 marks]

(c) Stores failed events with error details and retry metadata in an ETS table [2 marks]

(d) Provides a function list_dead_letters/0 that returns all failed events for inspection [2 marks]

Deliverable: Elixir source files in lib/nexapay/events/

SECTION E: MUTEX & CONCURRENCY CONTROL

[20 Marks | Recommended: 30 minutes]

Question E.1: Distributed Mutex Implementation [12 marks]

Implement a module **NexaPay.Concurrency.DistributedMutex** that provides distributed locking. Since we don't have a Redis cluster in the test environment, simulate it using an ETS-backed or Agent-backed lock store.

Required functions:

(a) **acquire(resource_type, resource_id, opts)** - Acquires a lock with configurable TTL. Returns {:ok, lock_ref} or {:error, :lock_held}. Must support timeout for waiting. [4 marks]

(b) **release(lock_ref)** - Releases a lock. Must verify the caller is the lock owner (prevent releasing another process's lock). [3 marks]

(c) **with_lock(resource_type, resource_id, opts, fun)** - Convenience function that acquires lock, executes fun, and ensures lock is released even if fun raises an exception. [3 marks]

(d) Automatic lock expiry after TTL (stale lock detection) [2 marks]

```
# Expected usage:
DistributedMutex.with_lock(:merchant_balance, merchant_id, [ttl: 30_000], fn ->
  # Check balance
  # Update balance
  # These operations are serialized per merchant
end)
```

Question E.2: Race Condition Prevention [8 marks]

Demonstrate that your mutex implementation prevents race conditions by writing a test:

(a) Spawn 10 concurrent processes that each attempt to increment a shared counter by 1 [3 marks]

(b) WITHOUT the mutex, show that the final value can be incorrect (demonstrate the race condition) [2 marks]

(c) WITH the mutex, show that the final value is always exactly 10 [3 marks]

Deliverable: Elixir source files in lib/nexapay/concurrency/ and test files in test/

SECTION F: DECISION LOG & LIVE WALKTHROUGH

[30 Marks | 20 marks for document + 10 marks for live walkthrough]

Question F.1: Decision Log Document [20 marks]

Your **decision_log.md** must contain the following sections:

Section	Content Required	Marks
## Approach & Prioritization	How you decided what to tackle first and time allocation strategy	4
## AI Interaction Log	At least 5 examples of AI prompts you used, what AI got right, and what you had to fix/override	6
## Validation Layer Analysis	Output from Section A.1	(counted in A)
## Contradictions Found	Output from Section A.2	(counted in A)
## Hidden Dependencies	Output from Section A.3	(counted in A)
## Architecture Decisions	Key design choices: why Phoenix vs Plug, how you structured modules, why you chose specific data structures	5
## What I Would Do Differently	Honest reflection on what you would change with more time	3
## Known Limitations	Bugs you know about but did not have time to fix	2

Question F.2: Live Walkthrough [10 marks]

After submission, you will have a 10-minute live session with evaluators where you must:

(a) Demo your working API with at least 3 test cases [3 marks]
(b) Explain how your validation pipeline processes a request end-to-end [3 marks]
(c) Show your pub/sub system in action (publish an event, show subscriber receiving it) [2 marks]
(d) Explain one thing AI got wrong and how you identified and fixed it [2 marks]

SUBMISSION INSTRUCTIONS

Repository Structure

```
nexapay/
+-- decision_log.md            # REQUIRED: Your decision log
+-- README.md                  # Setup & run instructions
+-- mix.exs
+-- config/
+-- lib/
|   +-- nexapay/
|   |   +-- schemas/           # Ecto schemas (Section B)
|   |   |   +-- merchant.ex
|   |   |   +-- transaction.ex
|   |   |   +-- payment_method.ex
|   |   |   +-- merchant_payment_method.ex
|   |   +-- transaction/
|   |   |   +-- state_machine.ex
|   |   +-- validation/        # Validation pipeline (Section C)
|   |   |   +-- pipeline.ex
|   |   |   +-- schema_validator.ex
|   |   |   +-- entity_validator.ex
|   |   |   +-- business_rule_validator.ex
|   |   |   +-- compliance_validator.ex
|   |   |   +-- risk_validator.ex
|   |   +-- events/            # Pub/Sub system (Section D)
|   |   |   +-- publisher.ex
|   |   |   +-- notification_handler.ex
|   |   |   +-- dead_letter_queue.ex
|   |   +-- concurrency/       # Mutex system (Section E)
|   |       +-- distributed_mutex.ex
|   +-- nexapay_web/            # Web layer
|       +-- controllers/
|       +-- router.ex
+-- test/                      # Tests
+-- priv/repo/migrations/      # Database migrations
```

How To Run

Your README.md must include commands to:

1. Install dependencies: `mix deps.get`

2. Setup database: `mix ecto.setup`
3. Seed test data: `mix run priv/repo/seeds.exs`
4. Run server: `mix phx.server`
5. Run tests: `mix test`

GOOD LUCK! Remember: we are evaluating your THINKING PROCESS, not just your code. A well-documented partial solution with clear reasoning scores higher than a complete but unexplained solution.