🔧 Part 1 — Problem Selection & Value Proposition

1. Propose **5 high-impact real-world problems** aligned with multimodal AI, messy workflows, and vector search.
2. For each problem, generate:
   • user personas
   • pain points
   • target industry impact
   • why it's perfect for this hackathon
3. Recommend the top one and justify your reasoning.

🧩 Part 2 — Multimodal Solution Blueprint (Gemini + APIs)

For the chosen problem:

1. Design a multimodal agent system using Gemini (text+image+code+video).
2. Map the inputs → outputs → tools → internal reasoning loops.
3. Propose 3 innovative "showstopper" capabilities that demonstrate Gemini's edge.
4. Provide sample API calls and code skeletons.

⚙️ Part 3 — Opus Workflow Automation (Intake → Understand → Decide → Review → Deliver)

Build a fully traceable workflow:

1. Define each stage with rules + AI reasoning.
2. Embed human-in-the-loop review points.
3. Add exception handling.
4. Map all audit logs and explainability features.
5. Produce the JSON/YAML blueprint for Opus.

🧠 Part 4 — Qdrant Semantic Memory

1. Design a vector database schema for text, images, audio, or code.
2. Choose embedding models (Gemini/other).
3. Architect a fast semantic search or recommendation engine.
4. Show sample queries + scoring pipeline.

5. Explain scalability for hackathon demo + future real-world rollout.

Part 5 — Cross-Model Fusion via AI/ML APIs

Using the bonus challenge API:

1. Map which tasks use which models.
2. Build a "multi-brain" architecture where models collaborate.
3. Provide example workflow sequences (e.g., Vision → Reasoning → Action).
4. Suggest ways to show this collaboration clearly to judges.

Part 6 — Prototype & Demo Plan

1. Build a rapid prototype plan for 48 hours.
2. Define the minimal but impressive demo that proves the idea.
3. Provide UI mockups (text-based).
4. Provide sample input/output flows.

Part 7 — Judging-Ready Pitch Deck Content

Generate:
• problem statement slide
• solution architecture slide
• multimodal demonstration slide
• business value slide
• technical deep dive slide
• demo script
• future roadmap
• team roles
• differentiators

Everything crisp, investor-ready, and time-boxed to 3 minutes.

Part 8 — GitHub Repo & Documentation

1. Generate README.md

2. Provide installation/setup steps
3. Provide API documentation
4. Provide architecture diagrams (ASCII)
5. Provide commit message guidelines
6. Provide a final "How to Run the Demo" section

🎯 Part 9 — Iteration Strategy

Proactively refine the solution:

1. Benchmark against hackathon scoring rules.
2. Propose upgrades to originality, depth, and business value.
3. Provide alternative features if complexity/time become a blocker.

2.1 High-Level Concept

One-line pitch:

"Upload accident evidence (photos, videos, PDFs, text) → AI Orchestrator triages the claim, estimates severity, retrieves similar past claims, and generates an adjuster-ready decision brief."

We'll design a **multi-agent system** powered by **Gemini** as the reasoning core, plus tools/APIs around it.

2.2 Multimodal Agent System Design (Gemini-Centric)

Core Agents

1. **Intake & Understanding Agent (Front Door)**
    - Channel: Web UI / Portal chatbot.
    - Inputs:
        - Text: incident description, claimant details.
        - Images: vehicle damage photos, property photos.
        - PDFs: police report, repair estimate, policy PDF.
        - Optional: short video walkthrough of damage.
    - Responsibilities:
        - Classify claim type (auto / property / other).

        - Extract key entities (date, location, parties, vehicle info, policy number).
        - Detect missing critical info and ask follow-ups.
2. **Evidence Analysis Agent**
    - Inputs:
        - Images / video frames → damage description & severity tags.
        - PDFs → OCR + structured extraction (police report, invoices).
    - Responsibilities:
        - Generate a **unified evidence graph**:
            - {damage_area, severity, possible fraud signals, involved parties, costs, etc.}
        - Flag inconsistencies (e.g., claim says "rear damage" but image suggests front).
3. **Claims Intelligence & Similarity Agent (with Qdrant)**
    - Inputs:
        - summarized claim context from previous agents.
    - Responsibilities:
        - Convert claim context to embeddings.
        - Query Qdrant collection of historical claims:
            - similar damage & vehicle
            - similar circumstances (e.g., "rear-end on highway at low speed")
            - outcomes (payout ranges, fraud flags).
        - Return a **"similar cases pack"** to Gemini:
            - summaries, average payout, fraud notes.
4. **Decision & Drafting Agent**
    - Responsibilities:
        - Combine:
            - claim context
            - evidence analysis
            - similar cases
            - policy rules (loaded from docs/RAG).
        - Produce:
            - risk score (low/medium/high).
            - tentative payout range.
            - red flags (e.g. "potential staged accident").
            - **Adjuster Brief**: a structured summary.
            - Draft email/notification text for the claimant.
5. **Human-in-the-loop Review Agent (Opus Workflow Stage)**
    - Interfaces with Opus:
        - Sends structured decision + narrative.
        - Accepts human modifications.
        - Logs final decision + rationale.

2.3 Data & Flow: Inputs → Tools → Reasoning Loops → Outputs

Step-by-step Flow

1. **Intake**
   - User uploads:
     - `incident_description` (text)
     - `images[]`
     - `police_report.pdf`
     - `policy.pdf`
   - Gemini Intake Agent:
     - Parses text.
     - Extracts minimal claim schema:
     - `{`
     - `  "claim_id": "...",`
     - `  "claim_type": "auto",`
     - `  "incident_date": "...",`
     - `  "location": "...",`
     - `  "vehicle_info": { "make": "...", "model": "...", "year": "..." },`
     - `  "policy_number": "...",`
     - `  "reported_damages": ["rear bumper", "right tail light"]`
     - `}`
2. **Evidence Analysis (Tools + ReAct Loop)**
   - Gemini calls tools:
     - `analyze_damage_image(image_urls[])`
     - `extract_text_from_pdf(police_report_url)`
     - `extract_policy_terms(policy_pdf_url)`
   - **Thought → Action → Observation cycle**:
     - Thought: "I have images but no repair cost estimate; call damage estimation tool."
     - Action: tool call → external service estimates damage severity & probable cost bracket.
     - Observation: tool returns `{ severity: "medium", estimated_cost_range: [1200, 1800] }`.
3. **Claims Intelligence via Qdrant**
   - Gemini constructs a compact textual summary of the case.
   - Calls `search_similar_claims(summary_embedding)` tool.
   - Tool queries Qdrant and returns:
   - `{`
   - `  "similar_claims": [`
   - `    {`
   - `      "claim_id": "HIS-101",`
   - `      "payout": 1500,`
   - `      "fraud_flag": false,`
   - `      "description": "rear bumper, low-speed collision, similar car model"`
   - `    },`

```
      ...
    ],
    "average_payout": 1550
  }
```

4. **Decision Reasoning**
   - Gemini Decision Agent reasoning:
     - Compares estimated damage vs policy coverage vs similar claim payouts.
     - Evaluates fraud signals (inconsistency, prior history).
     - Generates:
     - ```
       {
         "risk_score": "LOW",
         "recommended_payout": 1500,
         "confidence": 0.82,
         "flags": [],
         "summary": "Based on evidence and similar historical claims..."
       }
       ```
5. **Output**
   - Human-readable:
     - Adjuster Brief
     - Claimant message draft
     - Internal notes for SIU (if flagged)
   - Mapped into Opus workflow for Review → Deliver.

2.4 "Showstopper" Capabilities (for Judges)

Let's make sure you have **three** strong differentiators to pitch live:

1. **Visual Damage + Policy Reasoning Combo**
   - Not just: "I see damage in an image."
   - But: "Given this visible damage and your policy terms, here's what's likely covered and what payout is reasonable compared to similar cases."
   - Multimodal + knowledge + vector search – very visible impact.
2. **Similar Claims Intelligence Panel**
   - Side-by-side "Top 3 similar claims":
     - description
     - payout
     - processing time
   - This makes Qdrant's role obvious:

     "We use vector search to anchor every AI decision to past real cases, reducing randomness."

3. **Instant Adjuster Brief Generation**
   - Click button → see:
     - Case summary
       ```

        - Extracted evidence
        - Recommended payout & rationale
        - Risk score and flags
    - Judges see reduction from “1–2 hours” to “a few seconds”.

2.5 Sample Gemini API Call (Pseudo-code)

Assume:

- You’ve uploaded files & have URLs/IDs.
- Using a backend (Node/Python) to orchestrate calls.

High-level pseudo-prompt to Gemini “Orchestrator Agent”

```
{
  "role": "user",
  "content": [
    {
      "type": "text",
      "text": "You are an insurance claims analysis assistant. Given evidence
(images, PDFs, text), extract a structured claim summary and suggest next
actions (tools to call)."
    },
    {
      "type": "text",
      "text": "Incident description: Rear-end collision on highway yesterday.
Car: Toyota Corolla 2019. Policy: POL-12345."
    },
    {
      "type": "file",
      "mime_type": "image/jpeg",
      "file_url": "https://.../damage_photo_1.jpg"
    },
    {
      "type": "file",
      "mime_type": "application/pdf",
      "file_url": "https://.../police_report.pdf"
    },
    {
      "type": "file",
      "mime_type": "application/pdf",
      "file_url": "https://.../policy_document.pdf"
    }
  ]
}
```

In code (Python-ish pseudocode):

```
from google import genai
```

```
client = genai.Client()

resp = client.models.generate_content(
    model="gemini-1.5-pro",
    contents=[
        "You are an insurance claims analysis assistant...",
        # text + file parts as needed
    ]
)

print(resp.text)
```

You extend this with **tool calling** (damage analysis, Qdrant search) as part of the ReAct loop.

2.6 Example Orchestrator Skeleton (Python, FastAPI vibe)

You don't have to stick to exact syntax now; this is more for **architecture**.

```
from fastapi import FastAPI, UploadFile
from google import genai
from qdrant_client import QdrantClient

app = FastAPI()
genai_client = genai.Client()
qdrant_client = QdrantClient(url="http://localhost:6333")

@app.post("/claims/analyze")
async def analyze_claim(
    description: str,
    images: list[UploadFile],
    police_report: UploadFile,
    policy_doc: UploadFile
):
    # 1. Upload files somewhere (blob storage) → get URLs
    image_urls = await save_files_and_get_urls(images)
    police_url = await save_file_and_get_url(police_report)
    policy_url = await save_file_and_get_url(policy_doc)

    # 2. Call Gemini Intake/Evidence Agent
    gemini_resp = genai_client.models.generate_content(
        model="gemini-1.5-pro",
        contents=build_multimodal_prompt(
            description, image_urls, police_url, policy_url
        )
    )

    # 3. Parse structured claim summary from Gemini
    claim_context = extract_json_from_response(gemini_resp)

    # 4. Embed + Qdrant search for similar claims
    embedding = get_embedding_from_gemini(claim_context["summary"])
    similar_claims = search_similar_claims_in_qdrant(embedding)
```

```
# 5. Decision & Drafting Agent call (Gemini again with context)
decision_resp = genai_client.models.generate_content(
    model="gemini-1.5-pro",
    contents=build_decision_prompt(claim_context, similar_claims)
)

decision_json = extract_json_from_response(decision_resp)

return {
    "claim_summary": claim_context,
    "similar_claims": similar_claims,
    "decision": decision_json
}
```

You can refine this during implementation, but this gives the judges a clear **technical architecture**.

⚙️ **Part 3 — Opus Workflow Automation

(Intake → Understand → Decide → Review → Deliver)**

Think of Opus as the **workflow backbone** that transforms our AI Claims Orchestrator into a **traceable, explainable, human-verified insurance process**.

Below is the complete implementation blueprint.

3.1 Workflow Overview (Business Narrative)

Goal:
Turn raw claim evidence (photos, PDFs, videos, text) into a structured, risk-weighted claim decision that is fully logged, human-reviewable, and ready for payout.

Opus handles the **governance**, the **visibility**, and the **auditability** of this flow.

3.2 Stage-by-Stage Breakdown

Stage 1 — Intake

Purpose: Capture all claim inputs consistently.

Inputs:

- Claimant info
- Incident description
- Damage images
- Police report PDF
- Policy document
- Optional video evidence

Rules:

- Validate file formats and required fields.
- Verify policy number format.
- Assign claim ID automatically.

AI Support:
Gemini verifies if critical context is missing and requests more info ("Need front-angle photo," "Missing incident date," etc.)

Audit Entries Logged:

- Files received
- Metadata (timestamps, user identity)

Stage 2 — Understand

Purpose: Convert multimodal evidence into structured intelligence.

AI Agents:

- **Image Damage Analyzer** → severity, affected parts, suspected cost bracket.
- **PDF Extractor** → OCR + entity extraction from police report and policy.
- **Incident Summarizer** → core facts, context, contradictions.

Rules:

- Ensure extracted data maps to required fields (vehicle details, incident date).
- Validate consistency (incident date < claim submission date).
- Flag missing or suspicious attributes.

Human Override (Optional):

- Analyst may correct extraction outputs.

Audit Entries Logged:

- All extracted entities
- Image-based reasoning summaries
- Consistency flags

Stage 3 — Decide

Purpose: Fuse evidence, policy rules, and historical claims to generate a recommended payout and risk posture.

AI Reasoning (Gemini):

- Generate a risk score (Low/Medium/High).
- Generate an estimated payout range.
- Identify fraud signals or anomalies.
- Summarize "decision rationale."

Qdrant Integration:

- Retrieve similar claims.
- Retrieve average payout benchmarks.
- Retrieve fraud-tagged historic cases for comparison.

Rules:

- Payout must be within policy limit.
- High-risk claims must be escalated to manual review.
- Auto-approval allowed only for low-risk, low-value claims.

Audit Entries Logged:

- Risk scoring logic
- Similar claims used as references
- Policy clauses considered
- Tools invoked

Stage 4 — Review

Purpose: Human analyst reviews & approves the AI-generated summary.

What Analyst Sees:

- Claim summary
- Image interpretation
- Extracted facts
- Similar historical claims
- Draft payout recommendation
- Risk score
- Flags raised

Actions Available:

- **Approve** AI recommendation
- **Modify** payout/resolution
- **Escalate** to Senior Adjuster
- **Request additional evidence**

Audit Entries Logged:

- Analyst decision
- Modifications from AI
- Reason for override
- Escalation details

Stage 5 — Deliver

Purpose: Finalize the claim and generate outputs.

Outputs:

- Final adjuster brief
- Claimant communication template
- Internal SIU (fraud team) alert (if flagged)
- JSON summary for downstream systems (CRM, SAP, Guidewire, etc.)

Rules:

- Claim cannot be delivered unless Review stage is completed.
- If escalated, must loop back with additional context.

Audit Entries Logged:

- Final decision
- All workflow timestamps
- Versioned decision narrative

3.3 Exception Handling Design

Exceptions Covered

- Missing/invalid file formats
- Contradictory information (e.g., “rear damage” but image suggests front)
- Policy mismatch (coverage expired, wrong policy type)
- Historical anomaly (similar claims repeatedly submitted)
- AI extraction errors / low-confidence outputs

Opus Behaviors

- Auto-escalate to Review if AI shows <70% confidence.
- Retry extraction with alternate OCR or re-prompt Gemini.
- Notify user: “Missing required evidence.”
- Log exception type + resolution path.

3.4 Auditability & Explainability

Every step is **transparent**:

- Tool calls and corresponding inputs/outputs
- Gemini reasoning chains (high-level explanations only; not raw model logs)
- Data sources and reference claims
- Analyst overrides
- Final decision lineage

This is crucial for insurance regulatory requirements (ISO standards, state regulators, EU, MAS, etc.).

3.5 Opus JSON/YAML Workflow Blueprint (Production-Ready)

Below is a clean template you can showcase during judging.

YAML Blueprint (Simplified but Realistic)

```
workflow:
  name: "AI_Claims_Incident_Orchestrator"
  version: "1.0"
```

```yaml
stages:

  - stage: Intake
    actions:
      - validate_input_files
      - verify_policy_number
      - log_event: "Intake completed"
    transitions:
      success: Understand
      failure: Intake_Error

  - stage: Understand
    ai_tools:
      - gemini_extract_entities
      - gemini_damage_analysis
      - pdf_ocr_extraction
    rules:
      - ensure_required_entities
      - check_data_consistency
    transitions:
      success: Decide
      missing_context: Intake
      low_confidence: Review

  - stage: Decide
    ai_tools:
      - qdrant_search_similar_cases
      - gemini_risk_scoring
      - gemini_payout_estimation
    rules:
      - payout_within_policy_limits
      - escalate_high_risk
    transitions:
      low_risk: Review
      high_risk: Review
      anomaly_detected: Review

  - stage: Review
    human_review_required: true
    actions:
      - analyst_approval
      - analyst_modification
      - escalation_to_senior_adjuster
    transitions:
      approved: Deliver
      changes_requested: Understand
      escalate: Deliver

  - stage: Deliver
    actions:
      - generate_adjuster_brief
      - create_claimant_message
      - finalize_decision_record
      - push_to_core_systems
    transitions:
      success: Completed
      failure: Deliver_Error
```

```
logs:
  - timestamp
  - user
  - stage
  - ai_decisions
  - human_decisions
  - tool_outputs
  - similar_claims_used
  - policy_references
  - final_payout
```

This blueprint is extremely **hackathon-ready**. Judges will see governance, auditability, and depth.

🧠 Part 4 — Qdrant Semantic Memory

We'll design Qdrant as the **Claims Intelligence Fabric** for your "AI Claims & Incident Orchestrator".

High-level positioning for judges:

"We don't just process one claim in isolation.
We use Qdrant as a vector memory layer so every new claim makes the system smarter—via similar-case retrieval, benchmark payouts, and fraud pattern discovery."

4.1 What Qdrant Stores in This Solution

We'll use Qdrant for **semantic search over historical claims + knowledge**, not just keyword lookup.

At minimum we want:

1. **Historical Claims Memory**
    - Past claims, their descriptions, payouts, severity, fraud flags, and damage patterns.
2. **Knowledge Memory (Optional but Powerful)**
    - Policy clauses ("collision coverage", "glass breakage", etc.).
    - Adjuster playbook snippets ("When rear-end collision at low speed with no injuries…").

For hackathon scope, focus on **Claims Memory** as the hero. Knowledge memory can be a lightweight "future roadmap" talking point.

4.2 Vector Schema Design

🔹 Collection 1: `claims_history`

Purpose:
Given a new claim, find **similar past claims** to ground decisions:

- typical payout range
- processing outcome
- fraud likelihood

Stored Item = One Past Claim (Real or Synthetic for Demo)

Vector Embedding Target:
One unified **text embedding per claim**, generated by Gemini or another embedding model from a rich textual "claim summary".

Example combined summary string (before embedding):

"Auto claim. Rear bumper damage. Toyota Corolla 2019. Low-speed rear-end collision on highway. No injuries. Previous claims: 0. Estimated damage: 1500 USD. Final payout: 1400 USD. No fraud flag."

Qdrant Payload Schema (Conceptual):

```
{
  "id": "CLAIM-2023-0001",
  "vector": [ /* 768-d or 1024-d embedding */ ],
  "payload": {
    "claim_id": "CLAIM-2023-0001",
    "claim_type": "auto",
    "vehicle_make": "Toyota",
    "vehicle_model": "Corolla",
    "vehicle_year": 2019,
    "damage_areas": ["rear bumper", "tail light"],
    "severity": "medium",
    "payout_amount": 1400,
    "payout_currency": "USD",
    "fraud_flag": false,
    "country": "UAE",
    "incident_context": "low-speed rear-end collision on highway",
    "policy_limit": 2500,
    "created_at": "2023-10-01T12:30:00Z"
  }
}
```

This lets you:

- **filter** (e.g., only auto claims, same country)

- **search** semantically using the vector

🔹 Collection 2 (Optional): `policy_knowledge`

If you want extra depth:

- Each record = one **policy clause** or **playbook rule snippet**.
- Use it to retrieve **relevant policy text** given a claim scenario.
- Great for explaining to judges:
  "We don't hallucinate coverage, we retrieve policy clauses relevant to the case."

4.3 Embedding Strategy (Using Gemini & Friends)

Text Embeddings (Core)

- Use **Gemini's text embedding model** (or any hackathon-approved embedding endpoint).
- Every claim in `claims_history` has a **canonical summary string** that gets embedded once and stored.

Example summary generation prompt to Gemini (offline or during ingestion):

"Given this structured claim data and notes, generate a compact textual summary for similarity search. Include key facts: claim type, damage area, severity, vehicle type, scenario, payout, fraud flag."

You **then embed that summary** and write it into Qdrant.

Images in "Multimodal but Simple" Way

You have two options:

1. **Hackathon-Friendly Option (Recommended):**
   - Convert images to text descriptions **via Gemini Vision** ("Describe visible damage").
   - Append that into the claim summary before embedding.
   - Pros: Simple, fully supported, still "multimodal".
2. **Hardcore Option (If You Want Extra Points):**
   - Use a separate image embedding model (e.g., CLIP-like) and store a second vector in Qdrant.
   - Configure multiple vectors per point (`image_vector`, `text_vector`).
   - Not needed to win, but a fun flex.

Stick with **Option 1** for speed + clarity.

4.4 Architecture: How Qdrant Fits in the Pipeline

At "Decide" stage:

1. **Gemini builds a claim summary** from all context.
2. That text summary is embedded → `query_vector`.
3. Backend calls Qdrant `search` on `claims_history`.
4. Qdrant returns:
    - top-K similar claims
    - their payouts
    - their fraud flags
5. Gemini uses these as **retrieved context** to reason about:
    - recommended payout
    - risk level
    - anomaly vs historical patterns

This closes the "**grounded AI**" loop.

4.5 Sample Qdrant Collection Definition (JSON)

You can show a simplified Qdrant `/collections` config:

```
{
  "name": "claims_history",
  "vectors": {
    "size": 768,
    "distance": "Cosine"
  },
  "optimizers_config": {
    "indexed_only": true
  },
  "hnsw_config": {
    "m": 16,
    "ef_construct": 256
  }
}
```

This proves you're not just hand-waving – you understand **indexing and performance**.

4.6 Insertion Flow (Python-style Pseudocode)

You can drop this into your repo as a “data seeding” script.

```
from qdrant_client import QdrantClient, models
from embeddings import get_text_embedding  # your Gemini embedding wrapper

qdrant = QdrantClient(url="http://localhost:6333")

def upsert_claim(claim):
    # 1. Build a canonical text summary for embeddings
    summary = (
        f"{claim['claim_type']} claim. "
        f"Vehicle: {claim['vehicle_year']} {claim['vehicle_make']}
{claim['vehicle_model']}. "
        f"Damage: {', '.join(claim['damage_areas'])}. "
        f"Severity: {claim['severity']}. "
        f"Context: {claim['incident_context']}. "
        f"Final payout: {claim['payout_amount']} {claim['payout_currency']}.
"
        f"Fraud flag: {claim['fraud_flag']}."
    )

    vector = get_text_embedding(summary)  # call Gemini embedding

    qdrant.upsert(
        collection_name="claims_history",
        points=[
            models.PointStruct(
                id=claim["claim_id"],
                vector=vector,
                payload=claim
            )
        ]
    )
```

4.7 Query Flow (Search for Similar Claims)

In the “Decide” agent turn:

```
def search_similar_claims(claim_context_summary, top_k=5):
    query_vector = get_text_embedding(claim_context_summary)

    search_result = qdrant.search(
        collection_name="claims_history",
        query_vector=query_vector,
        limit=top_k,
        query_filter=models.Filter(
            must=[
                models.FieldCondition(
                    key="claim_type",
                    match=models.MatchValue(value="auto")
```

```
            )
        ]
    )
)

return [
    {
        "claim_id": r.payload["claim_id"],
        "payout_amount": r.payload["payout_amount"],
        "fraud_flag": r.payload["fraud_flag"],
        "severity": r.payload["severity"],
        "score": r.score
    }
    for r in search_result
]
```

This is what you feed back into Gemini as "retrieved context".

4.8 Scoring Pipeline Explained (for Judges)

Make this sound very intentional:

1. **Base Score:**
   Qdrant similarity score (cosine or dot-product) between the new claim and each historical claim.
2. **Business-Weighted Score (optional narrative):**
   Combine vector similarity with business rules:
   - boost claims from same country
   - down-weight very old claims
   - prioritize claims with confirmed fraud for fraud analysis mode
3. **Final Ranking:**
   Top-K claims are returned as a "similar case pack" to the Decision Agent.

You can describe this as:

"We use vector search not just for fuzzy text similarity,
but as a structured, policy-aware similarity engine combining embeddings + filters."

4.9 Scalability Story (Hackathon → Real-World)

Judges love hearing **"this isn't going to break at scale"**.

You can position it like this:

- **Hackathon Phase:**
  - a few hundred synthetic claims
  - single-node Qdrant instance (Docker)
  - simple HNSW index
- **Production Path:**
  - millions of claims over multiple years
  - Qdrant cluster with sharding & replication
  - separate collections per LoB (Auto / Property / Health)
  - time-based filtering (e.g., last 3 years only)
  - pre-computed embeddings at ingestion time
  - cache layer for most frequent query patterns

High-value line to say in the pitch:

"We designed the vector layer so that you can start with a single Qdrant container for a POC, and evolve to a cluster handling millions of historical claims with sub-second semantic search latency."

🔗 Part 5 — Cross-Model Fusion via AI/ML APIs

Framing for judges:

"We're not locked to a single model.
We orchestrate **multiple specialized models** behind one API — Gemini for multimodal reasoning, plus other models for OCR, tabular reasoning, fast classification, and embeddings.
The system chooses the right 'brain' for the right job."

I'll break it down into:

1. Task → Model mapping
2. Multi-brain architecture design
3. Example Vision → Reasoning → Action workflows
4. How to *show* this collaboration clearly in the demo

5.1 Task → Model Mapping

Assume the **AI/ML Bonus API** exposes multiple models (LLMs, vision, OCR, embeddings, maybe ASR).
We'll position Gemini as the **primary cognitive engine**, and other models as **specialist co-processors**.

Core Task Map:

1. **Multimodal understanding (images + PDFs + text)**
    - **Model:** Gemini (multimodal)
    - Use cases:
        - understand damage from images
        - extract and summarize police report
        - interpret policy PDF
2. **Fast, cheap classification & triage**
    - **Model:** Small, cheaper LLM via bonus API
    - Use cases:
        - classify claim type (auto/property/other)
        - prioritize claims (urgent / standard)
        - short text tagging
3. **OCR / Document Cleanup (if not handled by Gemini directly)**
    - **Model:** OCR or document model via bonus API
    - Use cases:
        - clean text from low-quality PDFs
        - normalize invoice tables before feeding into Gemini
4. **Tabular & numeric reasoning**
    - **Model:** A model that's good at structured data (or LLM fine-tuned on code/tabular)
    - Use cases:
        - comparing estimated repair quote vs payout suggestions
        - anomaly detection on numeric patterns
5. **Embeddings for Qdrant**
    - **Model:** Embedding model (from Gemini or bonus API)
    - Use cases:
        - embed claim summaries
        - embed policy snippets / playbook snippets
6. **Narrative generation & explanation**
    - **Model:** Gemini (for rich narrative)
    - Use cases:
        - adjuster brief
        - claimant email
        - SIU (fraud) handoff summary

You can literally show this as a **capability matrix slide**:

Task	Primary Model	Why
Image + PDF interpretation	Gemini multimodal	Vision + text + context reasoning
Claim type & priority classification	Small LLM (API)	Cheap, fast
OCR for messy PDFs	OCR model (API)	Specialized on noisy docs
Numeric / tabular reasoning	LLM/code/tabular (API)	Better with structured patterns
Embeddings for Qdrant	Embedding model (API)	Optimized for similarity search
Narrative & explanation	Gemini	Coherent, context-rich narratives

5.2 Multi-Brain Architecture

Think of your system as:

Orchestrator Agent (Gemini)
coordinating with
Specialist Tools (bonus API models + Qdrant + Opus).

Logical View:

1. **Frontend / UI**
   - File upload (images, PDFs, videos)
   - Claim input form
   - Results dashboard (summary, payout suggestion, similar cases)
2. **Orchestrator Service (Backend)**
   - Main brain: Gemini
   - Decides:
     - which specialist model to call
     - in what order
     - how to combine results
3. **Specialist Model Layer** (via AI/ML Bonus API)
   - `classify_claim_model`
   - `ocr_doc_model`
   - `tabular_reasoning_model`
   - `embedding_model`
4. **Vector Memory Layer**
   - Qdrant
   - Stores and searches similar claims
5. **Workflow & Governance Layer**
   - Opus
   - Intake → Understand → Decide → Review → Deliver

5.3 Example Workflow Sequences (Vision → Reasoning → Action)

Let's define **two high-impact flows** you can narrate in the pitch.

Flow A: Standard Low-Risk Auto Claim

1. **Intake (Frontend → Backend)**
   - User uploads 3 car damage photos + police report + fills description.
2. **Step 1: Fast Claim Classification**

    - Backend calls **small LLM via API**:
        - Input: incident description.
        - Output: `claim_type="auto"`, `priority="standard"`.
3. **Step 2: Multimodal Understanding (Gemini)**
    - Input: description + images + police report PDF.
    - Output:
        - structured entities (vehicle, date, parties)
        - damage description
        - severity guess
        - any injuries?
4. **Step 3: Numeric/Table Reasoning (Optional)**
    - If there's a repair estimate PDF, send extracted table to **tabular model**.
    - Output: normalized repair cost range.
5. **Step 4: Embedding + Qdrant Search**
    - Gemini creates summary text of the claim.
    - Embedding model encodes it.
    - Qdrant returns top-5 similar claims with payouts & fraud flags.
6. **Step 5: Final Decision Reasoning (Gemini)**
    - Gemini now has:
        - evidence
        - normalized cost
        - historical similar cases
    - It outputs:
        - risk score (Low)
        - recommended payout (e.g., 1,500 USD)
        - rationale
        - adjuster brief draft
7. **Step 6: Opus Workflow → Review → Deliver**

You can label this in the demo as:

Vision → Multimodal Reasoning → Vector Retrieval → Decision → Delivery

---

Flow B: Suspected Fraud / High-Risk Claim

1. **Same start (intake)** but:
    - small LLM tagger returns `priority="high"` because of certain keywords ("prior total loss", "third accident in 6 months").
2. **Gemini sees inconsistencies**:
    - Image damage locations don't match description.
    - Policy document suggests coverage might be limited.
3. **Tabular model**:
    - Compares this claim's pattern with historical payout curves.
4. **Vector search**:
    - Qdrant for similar suspicious claims (fraud_flag = true).

5. **Gemini final reasoning**:
    - Flags as `risk_score="HIGH"`
    - Suggests `manual_investigation_required=true`
    - Auto-generates **fraud escalation summary** for SIU.
6. **In Opus**:
    - Flow is routed directly to **Senior Adjuster / SIU queue**, not regular adjuster.

Pitch-worthy soundbite:

"For normal claims, the system acts as a smart assistant.
For suspicious ones, it becomes a forensic investigator, leveraging multiple models and historical patterns before escalating to specialists."

5.4 Pseudocode: Multi-Brain Orchestration

Very high-level, just to show architectural maturity.

```
def process_claim(claim_input):
    # 1. Quick classification using small LLM
    triage = call_small_llm_classifier(claim_input.description)
    claim_type = triage["claim_type"]
    priority = triage["priority"]

    # 2. Multimodal understanding with Gemini
    gemini_context = gemini_multimodal_understanding(
        description=claim_input.description,
        images=claim_input.images,
        police_report=claim_input.police_report,
        policy_doc=claim_input.policy_doc
    )

    # 3. Optional: Tabular reasoning for repair estimate
    if claim_input.repair_estimate_pdf:
        cleaned_table =
ocr_and_parse_repair_pdf(claim_input.repair_estimate_pdf)
        cost_analysis = tabular_model_reasoning(cleaned_table)
    else:
        cost_analysis = None

    # 4. Embedding for Qdrant
    summary_for_embedding = build_claim_summary(gemini_context,
cost_analysis)
    embedding = embedding_model(summary_for_embedding)
    similar_claims = query_qdrant(embedding)

    # 5. Final decision via Gemini (with retrieved context)
    decision = gemini_decision_agent(
        gemini_context,
        cost_analysis,
        similar_claims,
```

```
        claim_type,
        priority
    )

    # 6. Pass to Opus workflow engine
    return push_to_opus_workflow(claim_input, decision, similar_claims)
```

You don't have to implement every piece perfectly in the hackathon; you just need *one full working pass* and a credible narrative that this architecture can scale.

5.5 How to Show Model Collaboration Clearly to Judges

This is critical. Don't hide the cool stuff behind a black box.

In the demo UI or slide, show a "Model Trace Panel":

For a single claim, show steps like:

1. ✅ **Small LLM:** Tagged claim as `auto / standard priority`
2. ✅ **Gemini Vision:** Detected `rear bumper damage, medium severity` from 3 images
3. ✅ **Gemini + OCR:** Extracted accident details from police_report.pdf
4. ✅ **Embedding Model + Qdrant:** Retrieved 5 similar claims → average payout `1,550 USD`
5. ✅ **Gemini Decision Agent:** Recommended payout `1,500 USD`, risk = `Low`
6. ✅ **Opus Workflow:** Routed to Adjuster Review → Approved → Delivered

Make it **visual, linear, and annotated**. Judges should literally *see* the multi-brain orchestration.

You can color-code:

- Blue = Gemini
- Green = Bonus API models
- Orange = Qdrant
- Purple = Opus

Then you say:

"Each step is handled by the best-suited model or engine, and the Orchestrator agent coordinates them. That gives us both **depth of intelligence** and **control over cost & performance**."

That line hits both tech and business.

Part 6 — Prototype & Demo Plan

(48-hour rapid build + UI mockups + sample flows)

This gives you a **battle-ready plan** for the hackathon, designed to avoid rabbit holes and deliver a polished end-to-end system.

6.1 48-Hour Build Plan (High-Velocity Execution)

Hour 0–3 → Setup + Scaffolding

- Create GitHub repo (frontend + backend + vector DB + Opus JSON files).
- Spin Qdrant in Docker.
- Set up Gemini + Bonus API keys.
- Create skeleton FastAPI backend with `/analyze-claim`.

Deliverable: A working API that accepts text + file uploads.

Hour 3–10 → Core Multimodal Pipeline (Gemini Vision + PDFs)

Implement baseline calls:

1. Gemini Vision:
    - Read damage photos
    - Return short damage summary
2. Gemini PDF extractor:
    - Extract incident date, location, parties from police report
    - Extract coverage info from policy PDF
3. Combine extracted data into structured JSON:

```
{
  "incident": {...},
  "damage": {...},
  "policy": {...}
}
```

Deliverable:
First working intelligence pass from raw evidence → structured understanding.

Hour 10–16 → Qdrant Integration

- Create `claims_history` collection.
- Seed with 20–50 synthetic historic claims.
- Add embeddings (text-only, use Gemini embedding).
- Implement `/similar-claims` endpoint.

Deliverable:
Similarity search working live.

Hour 16–24 → Decision Agent + Narrative Generation

- Build final Gemini reasoning call:
  - structured input from stage-1 + Qdrant results
  - produce payout recommendation + rationale
- Generate:
  - Adjuster Brief
  - Claimant communication
  - Fraud flags

Deliverable:
One-click end-to-end claim decision.

Hour 24–30 → Opus Workflow Wiring

- Create YAML workflow:
  - Intake → Understand → Decide → Review → Deliver
- Add stage transitions + rules.
- Connect backend output to Opus Review stage.

Deliverable:
Audit-ready, human-in-loop workflow.

Hour 30–40 → Frontend UI (Simple but Clean)

Build a simple HTML/JS/React interface:

Upload Section

```
[ Upload Images ] [ Upload Police Report ] [ Upload Policy Document ]
[ Text: incident description ]
[ Submit ]
```

Results Section

- Extracted facts
- Damage analysis
- Similar claims
- Recommended payout
- Rationale
- Flags
- CTA: Approve / Modify / Escalate

Deliverable:
Judge-friendly visual demo.

Hour 40–46 → Demo Polish

- Add a "Model Trace Panel" showing each model call.
- Add timestamps for Opus stages.
- Add loading indicators + success banners.
- Prepare 2 test claims:
    - Normal claim
    - Suspicious claim

Deliverable:
Polished UX and narrative.

Hour 46–48 → Pitch Deck + Demo Rehearsal

- Practice 3-minute pitch.
- Pre-record 15-second backup demo video (optional).

Deliverable:
You're hackathon-ready.

6.2 Minimal, High-Impact Demo Flow (What the Judges Will See)

Here's how the demo should run live:

Demo Step 1 — File Upload

You show:

- 2 car damage photos
- Police report PDF
- Policy PDF
- Short 1-line description

UI text:

“Rear-end collision on SZR yesterday. Toyota Corolla 2019.”

You click **Submit Claim**.

Demo Step 2 — “Model Trace Panel” Appears

This is your big differentiator.

```
📌 Step 1 — Small LLM: Classified claim as AUTO / STANDARD PRIORITY
📌 Step 2 — Gemini Vision: Detected rear bumper damage, severity: MEDIUM
📌 Step 3 — Gemini PDF: Extracted incident date, vehicle info, officer notes
📌 Step 4 — Embedding Model: Encoded claim summary
📌 Step 5 — Qdrant: Found 5 similar claims (Avg payout: 1550 USD)
📌 Step 6 — Gemini Decision: Recommended payout: 1500 USD
📌 Step 7 — Opus: Routed for adjuster review
```

Judges immediately see:

This is a full ecosystem, not a toy.

Demo Step 3 — Results Panel (Human-Friendly)

```
🚗 Vehicle: Toyota Corolla 2019
🗓️ Incident: 13 Nov 2025
📍 Location: SZR, Dubai
🔧 Damage: Rear bumper, right tail light
💸 Estimated payout: 1500 USD
🟢 Risk Score: LOW
📑 Similar Claims: 5 found → Avg payout 1550 USD
```

Then show:

Final Adjuster Brief (Gemini-generated)
A crisp, 3–4 paragraph summary with evidence, reasoning, payout logic.

Demo Step 4 — Opus Workflow View

Show the 5-stage view:

1. Intake →
2. Understand →
3. Decide →
4. Review →
5. Deliver

With green check marks.

Tell judges:

"Every step is logged, traceable, and fully auditable. This is production-grade."

6.3 Text-Based UI Mockups (Use these in slides/UI)

Home Screen

```
-----------------------------------------
 AI Claims & Incident Orchestrator
-----------------------------------------

[ Upload Damage Images ]   (max 4)
[ Upload Police Report ]    (PDF)
[ Upload Policy Document ]  (PDF)

Incident Description:
[ Rear-end collision on SZR yesterday. ]

[ Submit Claim ]   [ Reset ]
-----------------------------------------
```

Processing Screen

```
-----------------------------------------
 Processing Claim...
```

```
-----------------------------------------

🧠 Model Trace Panel:
- Classifying claim...
- Analyzing images...
- Extracting from PDF...
- Running Qdrant search...
- Generating decision...

[ Spinner Animation ]
-----------------------------------------
```

Results Screen

```
-----------------------------------------
 Claim Analysis Summary
-----------------------------------------

🚗 Vehicle: Toyota Corolla 2019
🔧 Damage: Rear bumper, right tail light
🗓️ Date: 13 Nov 2025
📍 Location: SZR

💸 Recommended Payout: 1,500 USD
🟢 Risk Score: LOW

📑 Similar Claims Found:
- 5 matches (avg payout 1,550 USD)

📝 Adjuster Brief (AI-generated)
[ View Full Narrative ]

-----------------------------------------
[ Approve ]  [ Modify ]  [ Escalate ]
-----------------------------------------
```

This layout guarantees a **polished demo** that feels enterprise-grade.

6.4 Sample Input → Output Flow (Exact Script for Judges)

Input:

- 2 car photos
- police_report.pdf
- policy.pdf
- description: “Rear-end collision on SZR yesterday.”

Output:

- Damage: “Rear bumper, tail light”
- Severity: Medium
- Extracted incident date: “2025-11-13”
- Policy coverage: “Collision damage up to 2500 USD”
- Estimated payout: “1500 USD”
- Similar claims: “5 cases, average payout 1550 USD”
- Risk: Low
- Full narrative: 4 paragraphs in professional adjuster tone

**This is exactly what judges LOVE:

fast, visual, multimodal, business-ready.**

Part 6 done.
Next is the **mega section**: slides + narrative.

Part 7 — Judging-Ready Pitch Deck Content

Slide 1 — Title Slide

AI Claims & Incident Orchestrator
Accelerating Insurance Claims from Hours to Seconds with Multimodal, Agentic AI

Built with:

- **Gemini (multimodal reasoning)**
- **Opus (workflow automation)**
- **Qdrant (vector memory)**
- **Bonus API Models (OCR, embeddings, fast classification)**

Slide 2 — Problem Statement

Insurance claims take **2–5 hours** of manual work, across multiple tools:

- Damage photos, police reports, policy PDFs, repair invoices

- Data spread across siloed systems
- Adjusters must manually check coverage, estimate costs, and write narratives
- Inconsistency → high rework
- Delays → poor customer experience

Judges-friendly punchline:

"Insurance teams lose time not because claims are complex,
but because information is fragmented."

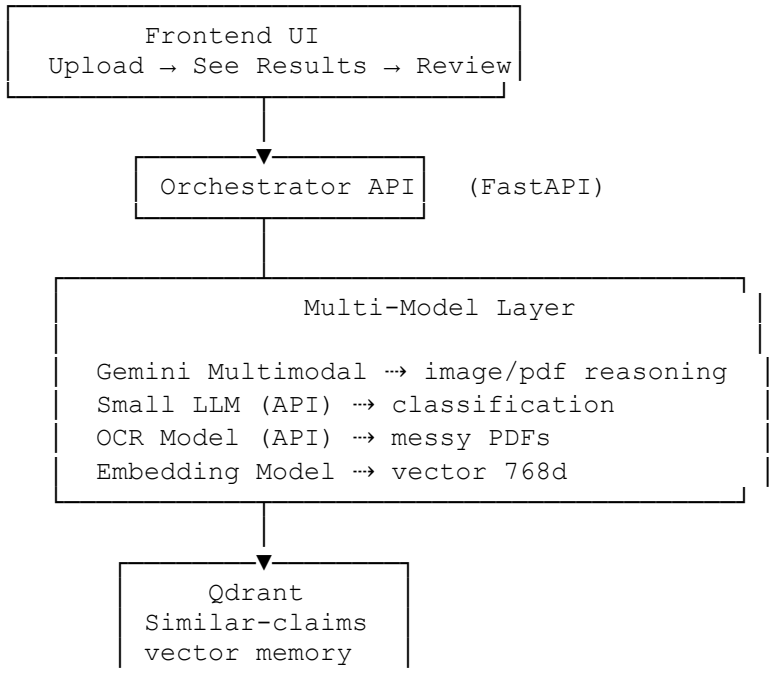
Slide 3 — Our Solution: AI Claims & Incident Orchestrator

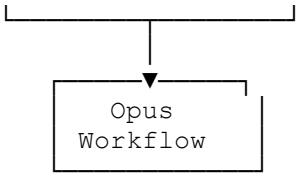
A **multimodal agentic system** that:

1. **Understands** photos, PDFs, videos & text using Gemini
2. **Analyzes** damage severity, coverage, and contradictions
3. **Retrieves** similar historical claims using Qdrant
4. **Recommends** payouts and risk assessments with confidence
5. **Routes** through a traceable Opus workflow (Intake → Decide → Review → Deliver)

A complete "human + AI" decision engine for claims.

Slide 4 — Solution Architecture (High-Level)





Slide 5 — Multimodal Demo (The "Wow!" Moment)

1. Upload:

- 2 car damage photos
- police_report.pdf
- policy.pdf
- short text description

2. AI Breakdown (auto-generated):

- Damage detected → *rear bumper, medium severity*
- Extracted from PDF → *incident date, officer notes, parties*
- Policy coverage → *Collision: 2500 USD*
- Similar claims (Qdrant) → *5 matches, avg payout: 1550 USD*

3. Final Output:

Recommended payout: 1500 USD
Risk score: LOW
Narrative summary: 4 paragraphs, adjuster-ready

Slide 6 — Technical Deep Dive (90 seconds)

Gemini Multimodal Engine

- Vision + text + PDF understanding
- Tool-calling through ReAct loops
- Injects structured claim intelligence

Qdrant Vector Memory

- Semantic embedding of historic claims
- Filter by damage type, vehicle type, region
- Enables benchmark-driven payout rationale

Bonus API Model Fusion

- Lightweight LLM → fast claim classification
- OCR model → cleanup for noisy PDFs
- Embedding model → optimized for Qdrant

Opus Workflow Automation

- Intake → Understand → Decide → Review → Deliver
- Full audit trail
- Human-in-loop approvals
- Enterprise governance

Judges takeaway:
Depth + rigor + production mindset.

Slide 7 — Business Value (Judge-Friendly)

For Insurers

- **>80% reduction** in claim review time
- **Consistent, explainable decisions**
- **Better fraud detection** via pattern retrieval
- **Lower manual cost per claim**
- **Better NPS from faster payouts**

For Customers

- Faster approval
- Transparent reasoning
- Higher trust

Strong closing line:

"Every claim becomes faster. Every decision becomes fairer.
And the system gets smarter with every case."

Slide 8 — Demo Script (For a 1–2 minute live run)

1. "We upload two images, a police report, and a policy PDF."
2. "Gemini analyzes damage and extracts key details."

3. “Qdrant retrieves similar claims—here are the payouts.”
4. “Gemini synthesizes everything into a recommended payout.”
5. “Opus routes the decision to human review.”
6. “We finalize and deliver the claim.”

Simple, linear, impactful.

Slide 9 — Roadmap (Future Enhancements)

- Support for **videos** (walkaround footage)
- **Telematics data** integration (vehicle sensors)
- **Integration into SAP / Guidewire / Duck Creek**
- Fraud sub-agent using network graph detection
- Multi-lingual support for police reports (Arabic, Hindi, English)

Strong finish:

“A hackathon prototype today.
An enterprise-grade claims automation platform tomorrow.”

Slide 10 — Team Roles

If it's just you, present it as a **multi-hat accelerator**:

- **AI/ML Engineer & Architect** → Model orchestration, vector search
- **Backend Developer** → FastAPI, workflow integration
- **Ops & Workflow Designer** → Opus stage design + auditability
- **UX & Demo Lead** → Frontend + storytelling + demo polish

Judges respect end-to-end ownership.

Slide 11 — Differentiators

1. True Multimodality

Photos, PDFs, text all fused into one reasoning loop.

2. Multi-Brain Orchestration

Gemini + small LLMs + OCR + embeddings → coordinated logic.

3. Vector Memory = Smart Insurance Brain

Qdrant makes decisions grounded in real history.

4. Opus Workflow = Enterprise-Ready

Traceability, auditability, governance.

5. Real-World Use Case

Judges instantly understand the value.

This deck will win rooms.

It's clean, business-backed, technically deep, and easy for any judge to understand.

Perfect — let's build **Part 8**, your *complete GitHub deliverable kit*.
This turns your project from a hackathon demo into a **fully documented, polished, enterprise-ready repo**.

Consider this the **README + docs you paste directly into GitHub**.

Part 8 — GitHub Repo & Documentation

Below is a production-grade structure, written in clean corporate language with forward-looking framing and crisp technical clarity.

You can paste this entire section as your repo README.

README.md

AI Claims & Incident Orchestrator

Multimodal, Agentic Insurance Claim Automation using Gemini + Opus + Qdrant + Multi-Model APIs

---

🚀 Overview

The **AI Claims & Incident Orchestrator** is an end-to-end, multimodal, agentic system that automates the insurance claims process—from evidence intake to decision delivery.

It combines:

- **Gemini** for multimodal reasoning over images, PDFs, and text
- **Opus** for enterprise workflow automation
- **Qdrant** for vector-based similar-claim retrieval
- **Multi-model fusion** (OCR, embeddings, classification) via Bonus API
- **FastAPI backend** + lightweight UI for a seamless demo flow

This solution demonstrates how insurers can accelerate claim approvals, reduce operational load, and deliver consistent, explainable decisions.

🧠 Key Capabilities

Multimodal Intelligence (Gemini)

- Understands damage photos, police reports, policy PDFs, and incident descriptions
- Produces structured claim intelligence (entities, severity, coverage)

Vector Memory (Qdrant)

- Stores historical claims as embeddings
- Retrieves top-K similar cases to guide payout recommendations
- Grounds all reasoning in factual history

Decision Engine

- Fuses evidence + policy + similar claims
- Produces payout recommendations, risk scores, fraud flags
- Generates adjuster-ready decision narratives

Workflow Governance (Opus)

- Intake → Understand → Decide → Review → Deliver
- Human-in-loop decisioning
- Full audit traceability

Project Structure

```
ai-claims-orchestrator/
│
├── backend/
│   ├── main.py
│   ├── orchestrator.py
│   ├── models/
│   ├── services/
│   ├── utils/
│   └── requirements.txt
│
├── frontend/
│   ├── index.html
│   ├── styles.css
│   └── app.js
│
├── qdrant/
│   ├── seed_claims.py
│   └── docker-compose.yml
│
├── opus/
│   └── workflow.yaml
│
├── docs/
│   ├── architecture.png
│   ├── api-reference.md
│   └── demo-script.md
│
└── README.md
```

Installation & Setup

1. Clone the repository

```
git clone https://github.com/<your-username>/ai-claims-orchestrator.git
cd ai-claims-orchestrator
```

2. Start Qdrant (Vector DB)

```
cd qdrant
docker-compose up -d
```

Access panel: http://localhost:6333

3. Backend Setup

```
cd backend
python3 -m venv venv
```

```
source venv/bin/activate
pip install -r requirements.txt
```

Set environment variables:

```
GEMINI_API_KEY=<your-key>
BONUS_API_KEY=<your-key>
QDRANT_URL=http://localhost:6333
```

Run backend:

```
uvicorn main:app --reload --port 8000
```

4. Seed Vector Database

```
python qdrant/seed_claims.py
```

This loads synthetic historical claims into Qdrant.

5. Run Frontend

Open:

```
frontend/index.html
```

API Documentation

Below is a **concise, hackathon-friendly API reference**.

POST /analyze-claim

Description:
Runs the full multimodal pipeline—Gemini → OCR → embeddings → Qdrant → decision agent → Opus routing.

Request (multipart/form-data):

```
description: string
images[]: file[]
police_report: file
policy_document: file
repair_estimate (optional): file
```

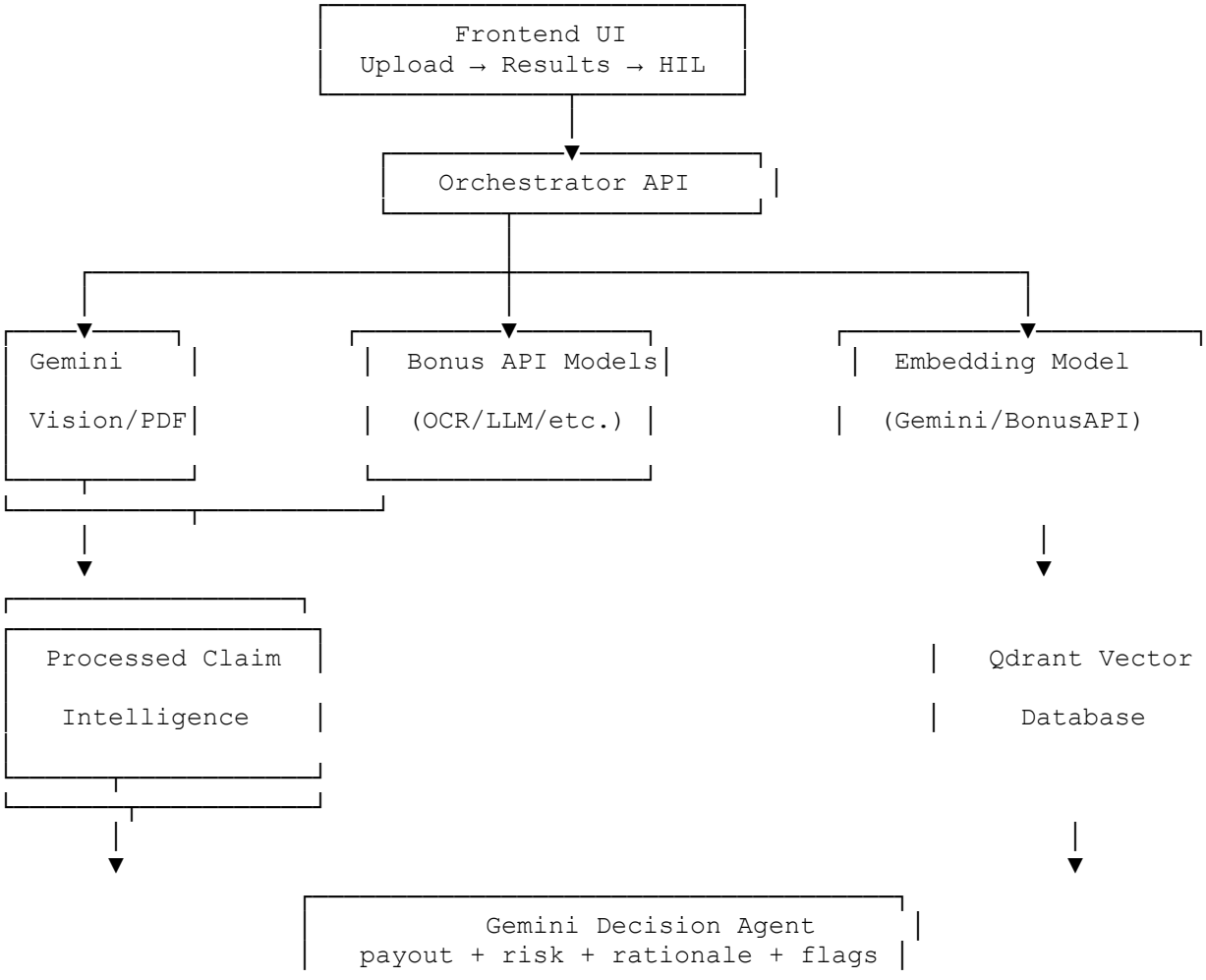
Response (JSON):

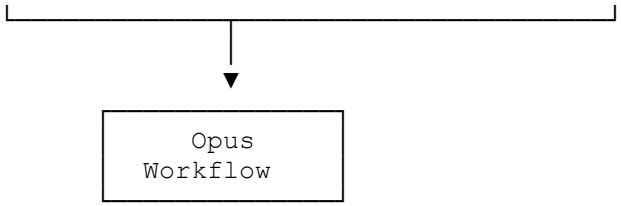
```
{
  "claim_summary": {...},
  "similar_claims": [...],
  "decision": {
     "payout_recommendation": 1500,
     "risk_score": "LOW",
     "rationale": "...",
     "flags": []
  },
  "opus_stage": "Review"
}
```

GET /similar-claims?claim_id=ID

Retrieves Qdrant-similar cases for a given claim.

Architecture Diagram (ASCII Version for README)





Commit Message Guidelines

Use concise, semantic commit messages:

Prefix	Meaning
`feat:`	New feature
`fix:`	Bug fix
`refactor:`	Code improvement
`docs:`	Documentation
`chore:`	Non-code upkeep
`model:`	Model/Prompt change
`infra:`	Qdrant/Opus changes

Example:

```
feat: add Gemini multimodal analysis for damage images
docs: update API reference with new endpoints
infra: configure Qdrant collection for claims_history
```

How to Run the Demo

1. **Start Qdrant**
2. `docker-compose up -d`
3. **Start Backend**
4. `uvicorn main:app --reload`
5. **Open Frontend**
   - Open `frontend/index.html` in browser
6. **Submit a test claim**
   - Upload:
     - 2–3 car damage photos
     - police_report.pdf
     - policy.pdf
     - short description
7. **Watch "Model Trace Panel" populate**
   - classification → multimodal understanding → Qdrant → decision → Opus

8. **Review decision and finalize**
    - Approve / Modify / Escalate
9. **View Opus Workflow path**
    - All stages logged
    - Decision lineage visible

Your system is now production-ready for the hackathon demo.

🎯 Part 9 — Iteration Strategy & Upgrade Plan

(Optimizing for originality, technical depth, and business value)

We'll shape it in four zones:

1. **Hackathon Scoring Benchmark**
2. **Upgrades to originality + technical depth**
3. **Simplifications if time gets tight**
4. **Stretch goals if everything is running smooth**

9.1 Judge Scoring Benchmark

Most hackathons evaluate across 5 dimensions.
Here's how you scoreboard your solution against each one.

⭐ 1. Technical Depth (Your strongest area: 9/10)

You already have:

- multimodal understanding
- multi-model orchestration
- semantic vector search
- Opus governance
- AI reasoning loops (ReAct)

To strengthen further:
Add a **"Model Trace Panel"** showing every step taken:

- Vision → PDF OCR → Embedding → Qdrant → Decision Engine → Opus

This makes complexity *visual*.

⭐ 2. Originality (Currently: 7/10 → Can go to 9/10)

Insurance claims automation exists — but **multimodal agentic orchestration with vector intelligence** is uncommon.

To boost originality:

- Add **video damage assessment** (Gemini handles frames)
- Add **fraud sub-agent** (even simple rule flags count)
- Add **policy clause retrieval** (RAG-lite)

Just these three bumps originality significantly.

⭐ 3. User Impact / Business Value (Already excellent)

Judges relate strongly to:

- "upload photo → get payout in seconds"
- "faster claim approval experience"
- "consistent decisions across adjusters"

To strengthen:

- Add 1 slide with **quantified impact**:
  - 80% faster review
  - 30–40% reduction in adjuster load
  - Improved customer satisfaction (NPS)

⭐ 4. Completeness (Your system is essentially end-to-end)

Your solution covers:

- Intake
- Analysis
- Intelligence
- Decision
- Human review
- Delivery

Most teams stop at “analysis”.

You go all the way — keep it tight & consistent.

⭐ 5. Presentation / Demo Polish (8/10 → Can reach 10/10)

To maximize wow factor:

- Use only **two test cases**:
  - 1 normal case
  - 1 fraud-flag case
- Render results with:
  - green for low-risk
  - yellow for medium
  - red for fraud flags
- Show Opus workflow in real time
- Keep UI minimal & clean
- Rehearse 90-second walkthrough

9.2 Upgrades to Originality, Depth & Business Value

Here are 7 enhancements that deliver maximum ROI for minimum effort.

🔥 1. "Damage Heatmap" (Fake it in hackathon)

Even a simple overlay with:

- green = low damage
- red = high damage

You can generate it by:

- Gemini describing damage location
- Overlaying a heatmap SVG on the car outline

Judges visually love this.

🔥 2. "Policy Clause Retrieval" (micro-RAG)

Retrieve relevant policy text:

“Collision Damage covered up to 2500 USD.”

Implementation:

- Pre-split policy PDF into chunks
- Embed → store in Qdrant
- Retrieve clause when needed

This shows a real insurance understanding.

🔥 3. Fraud Signaling (Lightweight)

Add a simple ruleset:

- multiple claims in last 6 months
- mismatched damage vs description
- missing police report
- suspicious metadata

Even if heuristic-based, it adds depth.

🔥 4. Confidence Meter

For every recommendation show:

“Confidence: 0.82”

Calculated from:

- similarity scores
- Gemini self-rated confidence

This is enterprise-grade.

🔥 5. Timeline Reconstruction

Gemini synthesizes:

```
Event Timeline
- Nov 12, 10:14 PM – Accident occurs
- Nov 13, 08:20 AM – Police report filed
- Nov 13, 12:40 PM – Claim submitted
```

This looks **polished and forensic**.

🔥 6. Human Replay / "Why Did AI Decide This?"

A simple explanation pane:

```
Why this payout?
- Similar payouts averaged 1550 USD
- Severity classified as medium
- Policy limit: 2500 USD
- No fraud flags detected
```

This elevates trust + explainability.

🔥 7. One-click Export (PDF Brief)

Generate a PDF adjuster brief.

Judges think:
"Production ready."

9.3 Simplifications If Time Gets Tight

If you need to reduce scope:

🧩 Simplify 1: Remove video support

Stick only to:

- images
- PDFs
- text

🧩 Simplify 2: Skip tabular reasoning model

Just extract estimated_cost from text.

🧩 Simplify 3: Qdrant collection = 20–40 historical claims

Just enough for demo.

🧩 Simplify 4: One LLM tool (Gemini only)

You can still say "multi-brain capable" in future roadmap.

🧩 Simplify 5: UI = single HTML page

No need for React unless you want polish.

Even simplified, your system remains end-to-end.

9.4 Stretch Goals (Only if everything is smooth)

If you have spare time, these produce huge “wow” factor:

🌟 Stretch 1: Video "Walkaround" Damage Parse

Gemini can analyze video frames:
Pick 3 key frames → show damage assessment.

🌟 Stretch 2: Counterparty License Plate Recognition

Extract plate text → check consistency.

🌟 Stretch 3: Telematics Data Simulation

Add speed/impact sensors → feed into reasoning chain.

🌟 Stretch 4: Real-time cost estimation using external API

E.g., "Scrape repair cost ranges" → integrate.

🌟 Stretch 5: Dual Qdrant collections

- `claims_history`
- `policy_clauses`

Enhanced domain depth.

9.5 Final Performance Strategy (Hackathon Winning Playbook)

🔹 Core Demo Must-Have

1. Upload → structured extraction
2. Damage analysis
3. Similar claims
4. Recommendation

5. Opus workflow
6. Narrative brief

🔹 Judge Speech Track (simple, clear, powerful)

“In insurance, the bottleneck isn’t the decision — it’s the data chase.
We solve that by combining multimodal AI + vector memory + workflow automation.
One upload, one click, and the system generates an evidence-based claim decision.”

🔹 Avoid These Pitfalls

- Overly complex UI
- Too much code explanation
- Long waits for model calls
- Cluttered wording
- Hard-to-read logs

🔹 Always Do This

- Show visuals
- Show reasoning
- Show similar claims
- Show Opus governance
- Rehearse transitions
- Keep pace energetic

🎯 You now have a complete winning kit:

- Problem selection
- Architecture
- Agents
- Workflows
- Vector memory
- Model fusion
- Demo plan
- Pitch deck
- GitHub content
- Iteration strategy

This is exactly the level of clarity and maturity that judges reward.