Solving the perception bottleneck for a Morrowind AI agent

The cloud VLM round-trip is unnecessary for ~95% of Morrowind gameplay. OpenMW's Lua scripting API provides structured access to nearly all game state — position, inventory, nearby actors, combat controls, pathfinding, dialogue records, and quest journals — making it the functional equivalent of Mineflayer for Minecraft. By combining this API-first perception with a lightweight local CV pipeline for the remaining edge cases, a fully local agent can achieve **sub-500ms total cycle times** on an M3 Max 128GB with zero cloud API calls. Morrowind's stat-based combat and OpenMW's built-in `world.pause()` OpenMW make this RPG uniquely suited for an LLM-driven agent, even with modest inference speeds.

This guide provides a complete engineering specification for the perception stack: what data to read from the API, when to fall back to vision, which models to run locally, and how to keep the entire perceive→decide→act loop under the latency budget.

OpenMW Lua is the Mineflayer for Morrowind

The critical architectural insight comes from NVIDIA's Voyager: that project never touches screenshots. It reads Minecraft's world through Mineflayer's structured API, feeds text-formatted state to GPT-4, and gets back executable code. 80.lv OpenMW's Lua scripting API (available in 0.49+) provides an equivalent structured interface for Morrowind, running as a **player-attached local script** inside the game client itself. OpenMW

What the API reads without any screen capture

OpenMW Lua exposes virtually everything an agent needs through four core packages:

`openmw.self` provides direct control and state for the player character. Readthedocs `self.controls.movement` sets forward/backward motion (-1 to 1), `self.controls.sideMovement` handles strafing, `self.controls.yawChange` rotates the camera, and `self.controls.use` triggers attacks or spell casts. The `Controls` interface enables full takeover: `Controls.overrideMovementControls(true)` hands all player input to the Lua script. Readthedocs This is the exact equivalent of Mineflayer's `bot.setControlState()`.

`openmw.nearby` provides spatial awareness. `nearby.actors` returns every active actor in loaded cells with full position data. `nearby.items`, `nearby.containers`, and `nearby.doors` enumerate interactable objects. `nearby.findPath(source, destination)` performs navigation-mesh pathfinding OpenMW OpenMW — the agent doesn't need to "see" where to walk; it computes routes algorithmically. `nearby.castRay(from, to)` handles line-of-sight checks. OpenMW

`openmw.types` gives deep state introspection. `Actor.stats(obj)` reads all 27 skills, 8 attributes, health/magicka/fatigue with current and modified values. OpenMW `Actor.inventory(obj)` OpenMW and `Actor.getEquipment(obj)` cover item state. OpenMW `Actor.activeEffects(obj)` lists active magic. `Actor.stance(obj)` reports weapon/spell readiness. OpenMW `Player.quests(obj)` accesses the full quest journal.

`openmw.core.dialogue` is the key perception shortcut. It exposes **every dialogue record** from the game's data files: `core.dialogue.topic.records` (all topics), `core.dialogue.greeting.records` (greetings), `core.dialogue.journal.records` (journal entries). Each record contains `.infos` with the response `.text`, plus filtering conditions like `.actorId`, `.actorFaction`, `.actorRace`, `.gender`, and disposition requirements. Since the agent knows its own faction standings, reputation, race, and quest stages via the API, **it can evaluate dialogue conditions programmatically and predict exactly what an NPC will say** — without reading the screen.

The complete accessibility matrix

Game state	API access	Method
Player position, rotation, cell	✅ Full	`self.position`, `self.rotation`
Health, magicka, fatigue	✅ Full	`Actor.stats(self).dynamic`
All 27 skills, 8 attributes	✅ Full	`Actor.stats(self).skills / attributes`
Full inventory and equipment	✅ Full	`Actor.inventory(self)`, `getEquipment`
Quest journal and stages	✅ Full	`Player.quests(self)`
Faction standings and rank	✅ Full	`NPC.getFactionRank()`
Nearby NPCs with positions	✅ Full	`nearby.actors`

Nearby items, doors, containers	✅ Full	`nearby.items/doors/containers`
Pathfinding between points	✅ Full	`nearby.findPath(src, dst)`
Line-of-sight / raycasting	✅ Full	`nearby.castRay()`
Movement and combat controls	✅ Full	`self.controls.*`
All dialogue text (offline)	✅ Full	`core.dialogue.topic/greeting/journal`
NPC disposition	⚠️ Computable	Formula from base + faction + race + rep
Active dialogue window text	❌ No API	Client-side UI rendering
Character creation screens	❌ No API	Wizard UI flow
Persuasion minigame wheel	❌ No API	Visual-only mechanic
Level-up attribute selection	❌ No API	UI popup
Lockpicking visual feedback	❌ No API	Animation-based

The items marked ❌ account for roughly **5% of total gameplay time**. Everything else — exploration, navigation, combat, inventory management, quest tracking, most dialogue — is fully API-readable.

TES3MP vs OpenMW: choosing the right interface

TES3MP's server-side Lua API is far more limited. It operates on a client-server model where the server sees state-change *events* (player moved, item taken, quest updated) but has **no access to client-side UI, dialogue windows, or real-time AI state**. It cannot read active dialogue, menu states, or send control inputs. TES3MP is only needed if multiplayer interaction is desired.

For a single-player AI agent, use OpenMW directly (0.49+) with a player-attached Lua script. This gives full control and perception. The recommended architecture connects the OpenMW Lua script to an external Python process via file I/O or a local socket:

```
# OpenMW Lua script (player script) writes state to a file/pipe
```

```
-- lua side: runs every frame
local function onUpdate(dt)
    local state = {
        position = {self.position.x, self.position.y, self.position.z},
        health = types.Actor.stats(self.object).dynamic.health.current,
        cell = self.object.cell.name,
        nearby_actors = {},
    }
    for _, actor in ipairs(nearby.actors) do
        table.insert(state.nearby_actors, {
            id = actor.recordId,
            pos = {actor.position.x, actor.position.y, actor.position.z},
            health = types.Actor.stats(actor).dynamic.health.current,
        })
    end
    -- Write JSON to shared file
    io.write_json("/tmp/morrowind_state.json", state)
    -- Read commands from Python agent
    local cmd = io.read_json("/tmp/morrowind_cmd.json")
    if cmd then
        self.controls.movement = cmd.movement or 0
        self.controls.sideMovement = cmd.strafe or 0
        self.controls.yawChange = cmd.yaw or 0
        self.controls.use = cmd.attack or 0
    end
end
```

The five situations that actually require vision

With the API handling 95% of perception, screen reading is needed only for a narrow set of UI interactions. Each has a targeted solution cheaper than a VLM.

Character creation wizard (race, class, birthsign selection): This is a one-time event at game start. The UI layout is completely deterministic — fixed button positions, fixed text. **Solution**: Hard-code the creation flow. The agent knows which race/class/birthsign it wants before the game starts. Use OpenCV template matching to detect which screen is active, then click predetermined coordinates. Total cost: **~15ms** per detection via template matching.

Active dialogue window text: While `core.dialogue` provides all possible dialogue records, the agent can't directly read what text the NPC is *currently displaying*. **Solution**: Implement dialogue condition evaluation in Python. Parse all INFO records from the ESM data (via tes3conv), evaluate conditions against known player state (faction, race, disposition

formula, quest stages), and **predict what the NPC will say**. This eliminates screen reading in ~90% of dialogue encounters. For the remaining cases (complex conditional dialogue), use **PaddleOCR on the cropped dialogue region**: 20-80ms, zero LLM required.

Persuasion wheel minigame: A visual mechanic where the player selects admire/intimidate/taunt/bribe from a rotating wheel. **Solution**: Template match the wheel's four quadrant labels (fixed positions once the wheel is open), then select based on disposition calculation. Alternatively, skip persuasion entirely — Morrowind allows bribing with gold through the API-readable barter system, or using Charm spells.

Level-up attribute selection: A popup showing which attributes can increase. **Solution**: The agent already knows its skill increases from the API (it tracked every skill gain). It can compute which attributes are available for +5 bonuses algorithmically, then use template matching to click the correct attribute buttons at known coordinates. Cost: **~10ms**.

Lockpicking feedback: Visual-only jiggle animation. **Solution**: Lockpicking in Morrowind is purely stat-based (Security skill + lock level + fatigue). The agent can compute success probability from stats and simply attempt repeatedly. No visual feedback interpretation needed for optimal play.

When vision is unavoidable: the sub-500ms local VLM stack

For genuinely ambiguous visual situations — unexpected UI states, spatial reasoning about terrain the navmesh doesn't capture, or verifying that an action succeeded visually — a local VLM provides the fallback. The target: **single screenshot → structured JSON in under 500ms** on M3 Max.

Model recommendations ranked by speed

Tier 1 — Sub-350ms cold inference (rapid triage):

Model	Params	Quant	VRAM	Est. speed (M3 Max)	Best for
FastVLM-0.5B	500M	4-bit	~350MB	~300+ tok/s	Fastest cold inference; Apple-native MLX+CoreML
SmolVLM-500M	500M	4-bit	~400MB	~250+ tok/s	Day-zero MLX support; edge-optimized

FastVLM is the top recommendation for latency. Apple's own model uses **FastViTHD**, a hybrid vision encoder that produces **16× fewer visual tokens** than standard ViT. Albase

This eliminates the vision encoding bottleneck that dominates VLM latency. Apple Machine Learning Research At 50 output tokens with 4-bit quantization: ~10-20ms image encoding + ~15-30ms prefill + ~170ms generation = **~250-300ms total**.

Tier 2 — Sub-600ms with quality (detailed analysis):

Model	Params	Quant	VRAM	Est. speed (M3 Max)	Best for
Moondream 2B	1.86B	4-bit	~816MB	~120-150 tok/s	Excellent UI localization (ScreenSpot 80.4 F1)
Qwen2.5-VL-3B	3B	4-bit	~2GB	~100 tok/s	Best vision understanding at this size

Moondream's architecture enables a critical optimization: separate `encode_image()` and `query()` calls. Encode the frame **once** asynchronously while the previous action executes, then query takes only generation time. SigLIP vision encoder runs in ~100-150ms.

Tier 3 — With prefix caching (repeated frame analysis):

Using **vllm-mlx** with prefix caching, Qwen2.5-VL-7B achieves **28× speedup** on repeated or similar frames. arXiv First frame costs ~1.5-2.5s; subsequent frames with the same system prompt and similar image drop to **~100-200ms**. For Morrowind's relatively static scenes between agent actions, this is extremely effective.

The quantization sweet spot

Research from Q-VLM (NeurIPS 2024) and MBQ (CVPR 2025) establishes that **vision tokens are ~10× less sensitive to quantization than language tokens**. The practical implication: aggressive 4-bit quantization on the vision encoder is nearly lossless, while the language decoder needs more care.

Q4_K_M is the recommended default for all VLMs on this hardware. It preserves ~95% of full-precision quality while delivering 2-2.5× speed improvement. Local AI Zone Codersera With 128GB unified RAM, you could run the 3B routine model at Q8_0 (only ~3.5GB) for maximum quality. Below Q4 (Q3_K_M and lower), structured output reliability degrades unacceptably — IFEval scores drop sharply. Ionio

Keeping output tokens minimal

The biggest controllable factor in VLM latency is output length. Design the JSON schema to require only **30-60 tokens**:

```
{"ui": "gameplay", "enemies": 1, "health": "high", "terrain": "path", "notable":
```

Not:

```
{"description": "The player is in a stone corridor with torchlight. There is one
```

The compact schema cuts generation time by 3-5×.

The hybrid perception architecture: API → CV → VLM decision tree

The perception stack follows a strict escalation hierarchy. Each level is invoked only when the previous level cannot provide sufficient information.

```
Level 0: API Read (~1ms)
    OpenMW Lua → JSON state dump
    Covers: position, stats, inventory, nearby actors,
            quest state, faction, dialogue records
    ↓ insufficient?

Level 1: Deterministic CV (~5-20ms)
    OpenCV template matching for UI state detection
    Covers: which menu is open, is dialogue active,
            is combat engaged (from UI indicators)
    ↓ need text?

Level 2: OCR (~20-80ms)
    PaddleOCR on cropped UI regions
    Covers: dialogue text, journal entries,
            item names, NPC names in UI
    ↓ ambiguous visual state?

Level 3: Local VLM (~250-500ms)
    FastVLM-0.5B or Qwen2.5-VL-3B
    Covers: spatial reasoning, unexpected UI,
            verification of action success
```

Level 1: Template matching at ~15ms per frame

OpenCV's `matchTemplate` with `TM_CCOEFF_NORMED` OpenCV detects Readthedocs Morrowind's fixed UI elements reliably. Pre-capture 5-10 template images (dialogue box

border, inventory header, map header, menu background, health bar frame). Matching 5 templates against a single grayscale 1080p frame takes **10-50ms total** on M3 Max. For Morrowind's fixed-resolution UI, checking only known regions of interest (ROIs) drops this to **sub-5ms**.

```
class MorrowindUIDetector:
    def __init__(self):
        self.templates = {
            'dialogue': cv2.imread('templates/dialogue_border.png', 0),
            'inventory': cv2.imread('templates/inventory_header.png', 0),
            'map': cv2.imread('templates/map_header.png', 0),
            'journal': cv2.imread('templates/journal_header.png', 0),
        }

    def detect(self, frame_gray):
        """~10-15ms for all 4 templates on M3 Max."""
        state = {}
        for name, tmpl in self.templates.items():
            res = cv2.matchTemplate(frame_gray, tmpl, cv2.TM_CCOEFF_NORMED)
            _, max_val, _, max_loc = cv2.minMaxLoc(res)
            state[name] = max_val > 0.85
        return state
```

Level 2: PaddleOCR for game text

PaddleOCR v5 is the fastest Python OCR engine suitable for this task: **12.7 FPS on GPU** benchmarks, TildAlice with only 15MB total model weight (4.4MB detection + 10.5MB recognition). tildalice On Morrowind's clean, high-contrast text rendered at fixed positions, even Tesseract achieves excellent accuracy. tildalice The key optimization: **crop first, then OCR**.

```
from paddleocr import PaddleOCR

ocr = PaddleOCR(use_angle_cls=False, lang='en', use_gpu=False, show_log=False)

def read_dialogue(screenshot, dialogue_bounds=(200, 600, 1720, 880)):
    """~20-40ms on cropped region. Returns dialogue text."""
    x1, y1, x2, y2 = dialogue_bounds
    region = screenshot[y1:y2, x1:x2]
    result = ocr.ocr(region, cls=False)
    if result and result[0]:
        return ' '.join([line[1][0] for line in result[0] if line[1][1] > 0.7])
    return ""
```

Alternative: Apple Vision framework via `VNRecognizeTextRequest` runs on the Neural Engine natively, achieving ~5-10ms on a cropped dialogue region in `.fast` mode. This avoids Python OCR dependencies entirely and leverages M3 Max hardware optimally.

Level 3: VLM only when truly needed

The VLM fires in a small minority of cases — when the agent encounters a visual state it can't classify through templates or OCR. Practical triggers include: unknown UI overlay, need to verify navigation succeeded visually, spatial reasoning about terrain not captured by the navmesh, or truly novel screen content the template library doesn't cover.

Making decisions fast: constrained decoding and tiered inference

Once perception is solved cheaply, the decision layer still needs an LLM. Two techniques make this fast: **GBNF grammar constraints** that force valid output and a **three-tier model hierarchy** that uses the smallest adequate model for each decision.

GBNF grammar eliminates wasted tokens

Constrained decoding via llama.cpp's GBNF grammars forces the model to output only syntactically valid game commands. GitHub This has a counterintuitive benefit: **it can actually speed up inference** by skipping scaffolding tokens. Aidan Cooper aidancooper Benchmarks from JSONSchemaBench show the Guidance framework produces constrained output at 6-10ms per token vs 15-17ms unconstrained. arXiv

```
# Morrowind agent command grammar (GBNF)
root ::= "{" ws "\"action\":" ws action "," ws "\"params\":" ws params ws "}"
action ::= "\"move\"" | "\"interact\"" | "\"attack\"" | "\"cast\""
         | "\"use_item\"" | "\"dialogue\"" | "\"wait\"" | "\"look\""
params ::= "{" ws param ("," ws param)* ws "}"
param ::= "\"" key "\":" ws value
key ::= [a-z_]+
value ::= string | number | "true" | "false" | "null"
string ::= "\"" [^"]* "\""
number ::= "-"? [0-9]+ ("." [0-9]+)?
ws ::= [ \t\n]*
```

With this grammar, a Q4_K_M model produces valid JSON **100% of the time** — the grammar makes invalid output physically impossible. Let's Data Science The semantic quality of values may degrade slightly at aggressive quantization, but structure is guaranteed.

Grammar compile time is a one-time **50ms** cost at startup. arXiv

Three-tier metabolic protocol

The agent runs three models simultaneously in memory (total ~28GB, well within 128GB):

Tier	Model	Quant	RAM	Speed (M3 Max)	Use case	% of calls
T1: Routine	Qwen3-3B	Q8_0	~3.5GB	~80 tok/s	Move, attack, loot, wait	**90%**
T2: Medium	Llama-3.1-8B	Q4_K_M	~5GB	~40 tok/s	Dialogue choice, inventory, buying	**8%**
T3: Complex	Qwen3-32B	Q4_K_M	~20GB	~15 tok/s	Quest planning, novel strategy	**2%**

Routing is **deterministic, not LLM-based**:

```
def classify_complexity(task_type, game_state):
    """Zero-cost routing — no inference needed."""
    if task_type in ('move_to', 'attack_nearest', 'loot', 'wait', 'equip'):
        return 1  # T1: pattern-matched routine action
    if task_type in ('dialogue_response', 'buy_sell', 'spell_choice'):
        return 2  # T2: requires some reasoning
    if task_type in ('quest_planning', 'stuck', 'unknown_situation'):
        return 3  # T3: genuine novelty
    return 2  # default to medium
```

At T1 (90% of calls), a 30-token command with GBNF constraint takes ~50-100ms. The agent makes **10-20 routine decisions per second** without breaking a sweat.

Behavior trees handle the other 90%

Most Morrowind gameplay doesn't need any LLM at all. Walking to a waypoint, following a path grid, attacking the nearest enemy, looting a corpse — these are deterministic behaviors. A behavior tree executes them at near-zero cost: Medium

```
# Simplified behavior tree for routine gameplay
class MorrowindBT:
    def tick(self, state):
        if state.enemy_nearby and state.in_combat:
```

```
        return self.combat_sequence(state)    # Rule-based: attack, potion if
    if state.dialogue_active:
        return self.dialogue_handler(state)    # OCR + T2 LLM for response
    if state.has_waypoint:
        return self.navigate(state)            # API pathfinding, zero LLM
    if state.has_plan:
        return self.execute_plan_step(state)  # Deterministic step from LLM p
    return self.request_plan(state)            # T3 LLM: "what should I do nex
```

The **plan-execute pattern** is critical: the T3 model generates a 5-10 step plan once Wollenlabs (e.g., "go to Balmora → find Caius Cosades → give him the package → ask about orders"), then the behavior tree executes steps 2-10 deterministically. The LLM is only re-invoked when the plan fails or completes.

Morrowind's latency budget is surprisingly generous

Morrowind is not a twitch game. **Combat is stat-based dice rolls**, not skill-based dodging. Quora The hit-chance formula is `(Weapon_Skill + Agility/5 + Luck/10) × (0.75 + 0.5 × Fatigue_Ratio) + Fortify_Attack`. Steam Community Player damage is determined by how long the attack button is held, not by reaction time. Steam Community Steam Community Attack animations take 0.5-1.5 seconds per swing. There is no dodge mechanic, no parry timing, no skill-based blocking. UESPWiki

A 1-second decision cycle is fully adequate for competent Morrowind combat. A 2-second cycle is acceptable for all non-combat gameplay (exploration, dialogue, inventory management, quest tracking) which constitutes ~90% of play time. For comparison, CRADLE's agent in Red Dead Redemption 2 operated at **5-15 seconds per action** and still completed objectives (though it failed at real-time combat).

The pause advantage

OpenMW provides `world.pause(tag)` and `world.unpause(tag)` in Lua. The agent can freeze the game during complex reasoning: OpenMW

```
-- Pause for LLM inference on complex decisions
local world = require('openmw.world')
world.pause("agent_thinking")
-- ... Python process runs T3 inference (1-2 seconds) ...
world.unpause("agent_thinking")
```

`world.simulationTimeScale` can also slow the game to any fraction of real-time. OpenMW

Setting it to 0.25 gives the agent 4× more wall-clock time for every game-second. This is the same strategy Voyager uses (pausing the Minecraft server) Hacker News and VideoGameBench Lite uses (pausing emulators). Substack

Realistic cycle time budget

Phase	Routine (BT)	Standard (T1 LLM)	Complex (T3 LLM, paused)
API state read	1ms	1ms	1ms
Template matching	—	15ms	15ms
OCR (if dialogue)	—	40ms	40ms
LLM inference	0ms	80ms (T1+GBNF)	1500ms (T3, paused)
Action execution	1ms	1ms	1ms
Total	**~2ms**	**~137ms**	**~1557ms (game paused)**

The weighted average across all decision types: roughly **~150ms per action cycle** with 90% BT, 8% T1, 2% T3. This allows **~7 decisions per second** for routine gameplay.

Pre-computing the world: the knowledge base that eliminates guesswork

The agent should **know** Morrowind's world before it launches the game. Pre-parsing the game's data files builds a complete spatial and narrative database.

ESM data extraction pipeline

tes3conv (Rust, by Greatness7) converts Morrowind.esm to JSON in seconds. GitHub The output contains every record type: NPCs, dialogue, quests, cells, items, path grids, doors, creatures, spells, factions.

```
# One-time conversion (~80MB JSON output)
tes3conv Morrowind.esm morrowind_data.json
```

```
import json

with open('morrowind_data.json') as f:
```

```
    data = json.load(f)

# Extract all NPCs with locations
npcs = [r for r in data if r['type'] == 'Npc']
# Extract path grids for navigation
pathgrids = [r for r in data if r['type'] == 'PathGrid']
# Extract all dialogue with conditions
dialogues = [r for r in data if r['type'] == 'Dialogue']
# Extract cell references (placed objects including NPCs)
cells = [r for r in data if r['type'] == 'Cell']
```

Building the spatial navigation graph

Path grid records (PGRD) contain walkable nodes and edges for every cell. Door records link cells together. Combined with exterior cell coordinates (each cell = 8192 × 8192 game units), this builds a **complete walkable graph of Morrowind's world**.

```
import networkx as nx

G = nx.Graph()

# Add intra-cell path grid nodes and edges
for pgrd in pathgrids:
    cell = pgrd['cell']
    for i, point in enumerate(pgrd['points']):
        node_id = f"{cell}::{i}"
        G.add_node(node_id, x=point['x'], y=point['y'], z=point['z'], cell=cell)
    for edge in pgrd['edges']:
        G.add_edge(f"{cell}::{edge[0]}", f"{cell}::{edge[1]}",
                   weight=euclidean_dist(pgrd['points'][edge[0]], pgrd['points'][

# Add inter-cell door connections
for door in door_records:
    src_node = nearest_pathgrid_node(G, door['src_cell'], door['src_pos'])
    dst_node = nearest_pathgrid_node(G, door['dst_cell'], door['dst_pos'])
    G.add_edge(src_node, dst_node, weight=10, type='door')

# Add fast travel (silt strider, boats, mages guild, intervention spells)
for route in fast_travel_routes:
    G.add_edge(route['from_node'], route['to_node'], weight=1, type='fast_travel'

# Route finding: Balmora to Vivec
path = nx.shortest_path(G, source="Balmora::0", target="Vivec, Foreign Quarter::0
```

The agent computes routes **before** moving, using pure graph algorithms at ~1-5ms per

query. No vision, no LLM.

Dialogue tree pre-evaluation

Morrowind's dialogue system is fully deterministic: each INFO record has conditions (speaker race, class, faction, player faction rank, disposition threshold, quest stage checks). GitLab UESPWiki The agent evaluates these conditions against known player state to **predict NPC responses without reading the screen**:

```
def evaluate_dialogue_options(npc_id, player_state, dialogue_db):
    """Predict available topics and responses for this NPC."""
    available = []
    for topic in dialogue_db['topics']:
        for info in topic['infos']:
            if matches_conditions(info['conditions'], npc_id, player_state):
                available.append({
                    'topic': topic['name'],
                    'response': info['text'],
                    'result_script': info.get('script', None)
                })
                break  # First matching info wins (Morrowind dialogue priority)
    return available
```

UESP knowledge base via RAG

UESP's MediaWiki API provides quest walkthroughs in structured form. Scrape the ~480 Morrowind quest pages, chunk by quest stage, and embed with **nomic-embed-text** (137M params, runs locally via Ollama at ~2-5ms per embedding) into ChromaDB:

```
import chromadb
from ollama import embed

client = chromadb.Client()
collection = client.create_collection("morrowind_quests")

# Embed and store quest walkthrough chunks
for chunk in quest_chunks:
    embedding = embed(model="nomic-embed-text", input=chunk['text'])
    collection.add(
        documents=[chunk['text']],
        metadatas=[{"quest": chunk['quest_name'], "stage": chunk['stage']}],
        embeddings=[embedding['embedding']],
        ids=[chunk['id']]
    )
```

```
# Query at runtime: "What do I do after delivering the package to Caius?"
results = collection.query(query_texts=["deliver package Caius next step"], n_res
```

Apple Silicon optimization: using every co-processor

The M3 Max has four compute engines. The perception stack should use each for what it does best.

Neural Engine (16 cores, 18 TOPS): Train a lightweight Core ML image classifier (via Create ML) on ~500 Morrowind screenshots labeled by UI state (gameplay, dialogue, inventory, map, menu, character_creation). Export as a ~2MB .mlmodel. Inference runs at **sub-5ms per frame** on the Neural Engine, providing instant UI state detection without template matching.

GPU (40 cores, Metal): All LLM/VLM inference via MLX. The **mlx-vlm** library (v0.3.11, 2.1k GitHub stars) supports Qwen2.5-VL, SmolVLM, FastVLM, and 20+ other architectures natively. Use **vllm-mlx** for production serving with prefix caching (28× speedup on repeated frames). MLX is **21-87% faster than llama.cpp** on Apple Silicon due to unified memory zero-copy operations and lazy evaluation fusion.

CPU (16 cores): Game orchestration, pathfinding (NetworkX), dialogue condition evaluation, ESM data queries, behavior tree execution. These are all lightweight serial operations.

Screen capture: Use **ScreenCaptureKit** for hardware-accelerated window capture. Configure `SCContentFilter` to capture only the OpenMW window. Frame delivery latency is **2-8ms** with `UserInteractive` QoS. Capture at 30fps (sufficient for the agent's decision rate); crop to specific UI regions before any processing.

```
let config = SCStreamConfiguration()
config.width = 1280  // Lower resolution sufficient for the agent
config.height = 720
config.minimumFrameInterval = CMTime(value: 1, timescale: 30)
config.pixelFormat = kCVPixelFormatType_32BGRA
config.showsCursor = false
let filter = SCContentFilter(desktopIndependentWindow: morrowindWindow)
```

The complete perception stack specification

Memory budget (128GB unified RAM)

Component	RAM	Notes
OpenMW game client	~2GB	Game + loaded assets
Qwen3-3B Q8_0 (T1 routine)	3.5GB	Always loaded
Llama-3.1-8B Q4_K_M (T2 medium)	5GB	Always loaded
Qwen3-32B Q4_K_M (T3 complex)	20GB	Load on demand
FastVLM-0.5B 4-bit (vision fallback)	350MB	Always loaded
nomic-embed-text (RAG)	300MB	Always loaded
ChromaDB + knowledge base	~500MB	Pre-loaded at startup
Pre-parsed ESM databases	~200MB	SQLite + NetworkX graph
PaddleOCR models	15MB	Loaded on demand
Core ML UI classifier	2MB	Neural Engine
macOS + system overhead	~8GB	—
Total	**~40GB**	**88GB headroom**

Heartbeat loop pseudocode

```
class PerceptionStack:
    """Sub-500ms perceive→decide→act on M3 Max, zero cloud calls."""

    def __init__(self):
        self.lua_bridge = OpenMWLuaBridge()          # File/socket IPC
        self.ui_detector = CoreMLUIClassifier()       # Neural Engine, <5ms
        self.ocr = PaddleOCR(lang='en')               # On-demand
        self.vlm = FastVLM("mlx-community/FastVLM-0.5B-4bit")  # GPU
        self.llm_t1 = MLXModel("qwen3-3b-q8")        # GPU, routine
        self.llm_t2 = MLXModel("llama-8b-q4km")      # GPU, medium
        self.llm_t3 = MLXModel("qwen3-32b-q4km")     # GPU, complex
        self.bt = MorrowindBehaviorTree()
        self.knowledge = MorrowindKnowledgeBase()     # ESM + UESP + RAG
        self.change_detector = FrameChangeDetector()
        self.command_grammar = load_gbnf("morrowind_commands.gbnf")
```

```
def tick(self):
    # Phase 1: API perception (~1ms)
    state = self.lua_bridge.read_state()  # Position, stats, nearby, quests

    # Phase 2: Check if BT can handle this (~0ms)
    bt_action = self.bt.evaluate(state)
    if bt_action and bt_action.confidence > 0.9:
        self.lua_bridge.send_command(bt_action)
        return  # Total: ~2ms

    # Phase 3: Screen capture + CV (only if needed, ~20ms)
    frame = self.capture_frame()
    if not self.change_detector.has_changed(frame):
        self.lua_bridge.send_command(self.last_action)
        return  # Total: ~5ms

    ui_state = self.ui_detector.classify(frame)   # CoreML, ~3ms

    # Phase 4: Text extraction if dialogue (~40ms)
    text = None
    if ui_state == 'dialogue':
        # Try API-based dialogue prediction first
        predicted = self.knowledge.predict_dialogue(
            state.interacting_npc, state.player_state)
        if predicted.confidence > 0.8:
            text = predicted.text  # Zero vision cost
        else:
            text = self.ocr.read_region(frame, DIALOGUE_BOUNDS)

    # Phase 5: VLM only for genuinely ambiguous visual state (~300ms)
    visual_context = None
    if ui_state == 'unknown' or (ui_state == 'gameplay' and state.is_stuck):
        visual_context = self.vlm.analyze(frame,
            "What is visible? JSON: {ui, enemies, terrain, notable}")

    # Phase 6: LLM decision with appropriate tier (~50-1500ms)
    prompt = self.build_prompt(state, ui_state, text, visual_context)
    tier = classify_complexity(state.current_task, state)

    if tier == 3:
        self.lua_bridge.pause_game()  # world.pause("agent")

    action = [self.llm_t1, self.llm_t2, self.llm_t3][tier-1].generate(
        prompt, grammar=self.command_grammar, max_tokens=40)

    if tier == 3:
        self.lua_bridge.unpause_game()
```

```
        self.lua_bridge.send_command(action)
        self.last_action = action
```

Expected performance profile

Scenario	Frequency	Latency	Cloud calls
BT-handled routine (walk, attack, loot)	70%	~2ms	0
T1 LLM routine (simple choice)	20%	~100-150ms	0
T2 LLM + OCR (dialogue, inventory)	7%	~300-500ms	0
T3 LLM + VLM (quest planning, stuck)	3%	~1.5-2.5s (paused)	0
Weighted average	—	**~70ms**	**0**

Conclusion

The path from a $50/day cloud VLM bill and multi-second latency to a sub-500ms local perception stack requires one fundamental shift: **stop treating Morrowind like a black-box screen to be interpreted, and start treating it like an API to be queried**. OpenMW Lua provides the structured game state interface that makes this possible — it is Morrowind's Mineflayer. The remaining visual edge cases (5% of gameplay) are handled by a cascade of increasingly expensive but decreasingly frequent techniques: Core ML classification at 3ms, PaddleOCR at 40ms, and a local FastVLM at 300ms as the absolute last resort.

Three design decisions dominate the performance envelope. First, **pre-computation eliminates runtime discovery**: the agent loads into the game already knowing every NPC location, every dialogue tree, every path grid connection, and every quest walkthrough. Second, **behavior trees handle 70% of actions at zero inference cost**, with LLMs reserved for genuinely novel decisions. Third, **OpenMW's pause mechanism** converts the hardest decisions from a latency problem into a correctness problem — the agent can take 2 seconds on a complex quest-planning inference without missing a single game frame.

The entire stack fits in ~40GB of the M3 Max's 128GB unified memory, leaving 88GB of headroom for experimentation with larger models. The weighted-average cycle time of ~70ms means the agent is making decisions faster than a human player reads dialogue text. Morrowind's stat-based combat system — where outcomes are determined by character builds rather than reaction time — means this is more than fast enough for expert-level play.