From LLM-Style to VLM-Style Agents: FrozenLake as a Case Study

Prime-Intellect-Inspired Agent Architecture

Prime Intellect Research

January 24, 2026

Difficulties with LLM-Style Agents in FrozenLake

The "Blindness" Problem

- **Perfect State Assumption**: LLMs rely on the environment providing exact coordinates or descriptions.
- **No Perception**: The agent cannot "see"; it only reads textual logs.
- **Grounding Issues**: If the text description is ambiguous or missing, the agent fails immediately.
- **Sim-to-Real Gap**: Real-world environments (robotics, web UI) do not provide structured text states.

"LLM agents are powerful planners but blind."

Advantages of Vision-Language Models (VLMs)

Why Switch to VLMs?

- **Visual Grounding**: The agent perceives the environment directly (pixels), just like a human.
- **Implicit State Understanding**: No need for the environment to "confess" its state in text.
- **Robustness**: Better handling of unstructured environments where text descriptions are hard to generate.
- **Universal Interface**: Pixels are a universal API for games, robotics, and tools.

VLMs bridge the gap between high-level reasoning and raw sensory data.

Ideological Shift: LLM $\rightarrow$ VLM

Aspect	**LLM Style**	**VLM Style**
Observation	Text Description	Image / Video Frame
Knowledge	Explicitly Given	Visually Inferred
Intelligence	Pure Planning	Perception + Planning
Input	Prompt	Prompt + Image

"The agent must see before it can think."

What is VLM-Style FrozenLake?

- **Environment Invariance**:
  - Same physics (‘frozenlake_world.py‘).
  - Same actions (UP, DOWN, LEFT, RIGHT).
  - Same evaluation criteria (+1 Goal, -1 Hole).
- **Visual Observation**: Instead of text coordinates, the agent receives a rendered image (‘frame_x.png‘).
- **Prime Intellect Constraints**:
  - **No Training**: Weights are frozen.
  - **No Fine-Tuning**: No gradient updates.
  - **No RL**: Learning via selection.

Architecture of VLM-Style FrozenLake

Components

- **Environment + Renderer**: The symbolic world now pipes state to a renderer to generate pixels.
- **Wrapper (Multimodal)**: Constructs a prompt combining the image and task instructions.
- **VLM**: Processes the visual field to identify the agent (Red Dot) and hazards (Holes) before reasoning.
- **Client**: Manages the selection-based memory system (‘memory.json‘).

Note: The architecture remains identical to LLM style; only the observation pipeline changes.

Detailed Environment Execution Flow

1. **Step 0: Initialization**:
   - Agent starts at (0,0).
   - Renderer saves initial view as frame_0.png.
2. **Step t: Observation**:
   - Wrapper reads frame_t.png.
   - Constructs prompt: *"Current view is frame_t.png..."*
3. **Step t: Action**:
   - VLM sees frame → Decides <action>RIGHT</action>.
   - Environment executes move.
4. **Step t+1: Update**:
   - Agent moves to new tile.
   - Renderer generates frame_{t+1}.png.
   - Loop repeats with new frame.

frame_t.png → VLM → Action → Env → frame_{t+1}.png

Memory Architecture (‘memory.json‘)

The memory is no longer a list of strings. It is a structured **Lookup Table** (Q-Table) that maps states to action values.

Structure of the Q-Table

```
{
  "(0, 0)": {            // State (Coordinate)
    "UP": -0.1,          // Bad Action
    "DOWN": 0.0,
    "LEFT": 0.0,
    "RIGHT": 0.8         // Highly Recommended
  },
  "(0, 1)": {
    ...
  }
}
```

Key Idea: We do not just replay history. We **aggregate** history into

Memory: The Implicit Q-Table

- **Structure (Q-Table)**:
  - Instead of storing full lists of steps, we store the aggregate **value** of actions.
  - `memory.json`: Maps **Visual State** $\rightarrow$ {**Action: Score**}.
- **Update Mechanism (Online Learning)**:
  - After every step, the system performs a **Q-Learning Update**:
  - $Q(s, a) \leftarrow Q(s, a) + \alpha[R + \gamma \max Q(s', a') - Q(s, a)]$
  - This happens "outside" the VLM, updating the context for future episodes.
- **Utilization (In-Context RL)**:
  - The VLM receives the scores for its **current location** as a "Hint".
  - Prompt: *"History suggests: UP(-0.1), RIGHT(1.2)."*
  - The VLM combines this **symbolic advice** with its **visual perception** to decide.

Crucial Distinction

The Q-Table is **not** the agent; it is just a tool (a "cheat sheet").

- **Standard RL**: The table dictates the action ($\arg\max Q$).
- **VLM Agent**: The VLM **sees** the image, considers the table's advice

LLM Style vs VLM Style Comparison

Criterion	LLM Style	VLM Style
Performance	High (Easy)	Lower (Harder)
Cost	Low (Tokens only)	High (Image processing)
Grounding	Weak	Strong
Generalization	Poor	Better
Need for Training	No	No
Sim. to Reality	Low	High

Is VLM Needed for FrozenLake?

× **Not needed for solving FrozenLake**:

- The state space is small and discrete.
- Symbolic solvers or simple LLMs solve it efficiently.

✓ **Useful for Research**:

- **Architectural Validation**: Verifies the multimodal pipeline works.
- **Perception Grounding**: Tests if the model can map pixels to concepts ("Hole", "Safe").
- **Research Realism**: Simulates constraints of real-world robotics.

"VLM FrozenLake is for research, not performance."

When Should You Use VLMs?

Use LLMs when...

- State is natively text or code.
- High precision logic is required.
- Low latency/cost is critical.
- Representation is symbolic (Database, CLI).

Use VLMs when...

- State is unstructured (Pixels, Camera).
- Environment details are not pre-parsed.
- Spatial reasoning is required.
- Interaction involves GUI or Physical World.

FrozenLake is Symbolic $\rightarrow$ LLM is natively better.
Real World is Visual $\rightarrow$ VLM is required.

Final Takeaways

- **Universal Loop**: The Agent-Environment loop remains invariant regardless of modality.
- **Ideology Preserved**: Evolution happens via memory selection, not parameter updates.
- **Stepping Stone**: FrozenLake verifies the VLM architecture before scaling to complex visual tasks (e.g., Minecraft, WebAgent).

"LLM agents evolve thoughts.
VLM agents evolve perception."