Vijay, Nipun, Nisarg Unified Sprint: “BHIV Core + Reinforcement Intelligence Layer” 6 Day Plan

Goal: Build the full multimodal, agentic BHIV pipeline and embed a lightweight reinforcement layer that logs decisions, tracks rewards, and enables learning from Day 1 — with no friction to normal flow.

What Changes?

- We don’t replace anything — we augment it.
- RL acts like a learning observer, recording every decision (agent, model, retry) and tagging it with rewards (clarity, success, cost, etc.).
- At first, the RL layer only tracks and logs. Later, it starts making suggestions — like a junior intern learning from experience.

Final Unified 6-Day Sprint Plan with RL Integration

Da	Focus	Key Outcomes	RL Layer
Day 1	Final Registry + RL Logging Infra	Agent/LLM registries, standard run interface, RL stubs	RL context + log_action(), get_reward()
Day 2	MCP ↔ Agent Loop + Action Space Defined	Agent/LLM routing, fallback handling, reward schema	Define ActionSpace, add reward_fn() stubs
Day 3	PDF/Image/Audio Input + Replay Memory	Multimodal routing complete, reward logging live	Add replay_buffer, log_run(input, result)
Day 4	Retry/Fallback + RL Hooks in MCP	Full input-agent-tool-adapter flow, retry logic	RL selector stub runs if confidence > threshold
Day 5	End-to-End Runs + Simulated Feedback	10 sample tasks logged, reward trends observed	Store reward + decisions in learning_log.json
Day 6	Cleanup + Alpha Review + RL Toggle	Final FastAPI/CLI runs, docs, Loom demo	RL toggle + notes for future model training

Component-Wise Integration

Nisarg (Agent Router)

- Add rl_agent_selector.select_agent(task) fallback
- Track selected_agent, fallback_used, and log to rl_context.json
- Simulate exploration by randomizing fallback agent 10% of the time

Vijay (LLM Selector)

- Add rl_model_selector.select_model(input) suggestion hook
- Track model choice, estimated cost, simulated quality score
- Implement lightweight bandit logic (e.g., random → best performing model)

Nipun (Learning Adapter)

- Add get_reward_from_output(output) scoring
- Track tag count, clarity score, relevance → reward score
- Feed these back into RL context and learning_log.json

RL Core Files (Shared Among Devs)

File	Purpose
rl_context.py	Central logging and action/reward tracking
agent_selector.py	Random → RL-based agent suggestions
model_selector.py	Bandit/softmax-based model selector
reward_functions.py	Reward logic from adapter and logs
replay_buffer.py	Stores past inputs, outputs, decisions
learning_log.json	JSON file for analysis and future training

Deliverables Recap by Day 6

- BHIV Core: CLI + FastAPI, agent registry, LLM router, adapter, toolchains, logging
- RL Layer: Context logger, stub selectors, reward estimators, memory log, toggle flag
- MongoDB: Logs agent chains, token use, retry paths, reward scores
- Docs: Final schema, adapter fields, reward function docs, README for RL toggle
- Loom: Show run with/without RL advice on same task, compare output

Day-by-Day Breakdown

Objective: Complete and integrate all components of the Agentic LLM system – including MCPs, multimodal agents, toolchains, adapters, memory, token logging, and full input/output routing – for Alpha readiness.

Collective Sprint Outcome by End of Day 6:

- One-click task: audio/text/image/PDF input
- Dynamically routed via MCP → correct agent/tool/LLM
- Result passed to Nipun Adapter → Learning Object JSON
- MongoDB logs: agent chain, token cost, memory
- Retry/fallback logic and lightweight memory
- CLI and FastAPI triggers tested and working
- Integration with all 3: LLMs, Vision (img), Speech (audio), PDF, Text

Day 1 – Final Pipeline Mapping and Registry Extensions

Goal: Set up the complete registry, base agent, and interface for all inputs and agents.

Nisarg:

- Refactor agent_registry.py to support:
  - ImageAgent
  - AudioAgent
  - TextAgent
  - ArchiveAgent
- Enable model preference tags and fallback logic
- Confirm all agents implement a consistent .run(payload) interface

Vijay:

- Extend llm_router.py:
  - Add working stubs for OpenAI Vision, Whisper
  - Add run_with_model() for: gpt, gemini, claude, grok
- Begin building StreamTransformerAgent for real-time input merging

Nipun:

- Finalize schema for Learning Object
- Document all required fields
- Scaffold final version of nipun_adapter.py

Integration Check:

- All agents return response in a unified format: { agent_name, output, metadata }
- Ensure MCP can dynamically pick agents from registry

Q&A:

- Q: How are agent inputs unified?
  A: The MCP accepts a payload: {input_type, content, model, tags} and dispatches via the registry.

Side Note: Assume all input comes from learners. Focus on reliability over speed. Handle missing fields gracefully.

Day 2 – MCP ↔ Agent Bridge and Tooling Chain

Goal: MCP runs full input-agent-tool-adapter loop.

Nisarg:

- Extend mcp_bridge.py to:
  - Accept payloads with file-type hints
  - Dispatch to relevant agent via registry
  - Log task UID, timestamp, agent used, fallback (if any)

Vijay:

- Finalize token tracking stub:
    - Track input word count, output chars
    - Estimate cost based on model
- Add simple calculator and RAG tool as callable functions

Nipun:

- Begin writing adapter tests:
    - Feed dummy outputs from all agents
    - Map to learning object
    - Add error handling for missing metadata

Integration Check:

- MCP → Agent → Tool/LLM → Adapter → Output log
- Mongo logging active with { input, agent_chain, final_output, timestamp }

Q&A:

- Q: What happens if the agent fails?
  A: The MCP should retry with a fallback model or return "status": "error" with a log entry.

Side Note: Keep each layer stateless except for memory logs. Retry logic should not loop infinitely.

Day 3 – Multimodal Chain Handling and Memory Layer

Goal: Enable audio, image, and PDF inputs across the entire chain.

Nisarg:

- Create cli_runner.py to accept inputs via CLI
- Test chain: PDF → ArchiveAgent → GPT → Adapter
- Add fallback if ArchiveAgent fails

Vijay:

- Integrate Whisper (stub or real) to convert audio to text
- Integrate Vision model (OpenAI Vision API or stub)
- Add stream_handler.py to simulate multimodal merging

Nipun:

- Extend adapter schema:
  - Include content_type, confidence, difficulty, subject_tag
- Begin adapter endpoint in FastAPI for standalone testing

Integration Check:

- All 4 input types (Text, PDF, Image, Audio) should return a JSON via the MCP

Q&A:

- Q: How do we unify all inputs?
  A: The input_type field determines the preprocessing step before routing to the agent.

Side Note: Prioritize education-friendly outputs. Every tool must return something usable to a learner.

Day 4 – Retry + Fallback Logic, Token Tracker, and Logging

Goal: Add reliability, observability, and cost-awareness to the system.

Nisarg:

- Implement agent_memory_handler.py:
  - Cache recent inputs and outputs per agent
  - Store as short-term memory logs

Vijay:

- Extend logging to include:

- Tokens used
  - Model used
  - Retry attempts
- Add fallback transformer logic (e.g., if GPT fails → use Gemini)

Nipun:

- Finalize FastAPI endpoint:
  - POST input → return adapter-mapped JSON
  - Validate request schema and response fields

Integration Check:

- Logs include: UID, retries, fallback path, output, token cost, agent chain
- Adapter handles edge-case failures gracefully

Q&A:

- Q: How is token cost calculated?
  A: Input word count + output char count × model rate (mocked for now)

Side Note: Use mocks for now. Real token tracking can be integrated once the core is stable.

Day 5 – End-to-End Tests, Pipeline Demos, and CLI Wrapping

Goal: Full pipeline test and dry-run of Alpha test cases.

Nisarg:

- Finalize CLI: python cli_runner.py --input image.jpg --model gpt
- Ensure result logs as JSON in /logs/ with full trace

Vijay:

- Refactor LLM router and agents for clean output structure
- Ensure all models (gpt, gemini, claude) are callable via a common router

Nipun:

- Sync with agent outputs and finalize adapter mappings
- Push adapter_usage.md, nlo_schema.md, and FastAPI test guide

Integration Check:

- CLI and FastAPI both work end-to-end for:
  - Text → GPT
  - Image → Vision → GPT
  - Audio → Whisper → GPT
  - PDF → Archive → GPT

Q&A:

- Q: Can we trigger this from a simple UI or chatbot later?
  A: Yes, CLI + API is the foundation. Frontend can wrap this post-Alpha.

Side Note : This sprint is to stabilize the mind of the system. UX will follow only after agent sanity is confirmed.

Day 6 – Buffer, Code Cleanup, Docs, and Alpha Review

Goal: Final cleanup, Loom demo, and Alpha readiness check.

Everyone:

- Clean up folders, remove unused files, push final docs
- Record Loom of:
  - Input → agent route → output (JSON log)
  - CLI and FastAPI both
- Alpha checklist:
  - 4 input types work?
  - Retry/fallback working?

- Logs captured?
- Output learning object consistent?

ADDING RL COMPONENTS

Day 1 — Add RL Infrastructure

All Devs

- Add reinforcement/rl_context.py — centralized context for logging actions, results, and rewards
- Add utils/task_reward.py with get_reward_from_log(output_json) stub

Nisarg (Agent Router)

- Log agent selection per task to RL context
- Add routing fallback tracking

Vijay (LLM Router)

- Log LLM selection and mock "response quality" score
- Add dummy estimate_cost() and estimate_usefulness() for RL

Nipun (Learning Adapter)

- Add reward feedback into NLOs (agent_score, clarity_score, etc.)

Side Note:

Let's not train RL from Day 1 — just start recording everything.

Day 2 — Define Action Space + Rewards

All Devs

- Define ActionSpace: agent choice, LLM choice, retry, cost path, token budget

Nisarg

- Add random fallback agent routing (for exploration)

Vijay

- Implement bandit-style model selector (random → reward-based)

Nipun

- Simulate scoring of learning output quality (reward = tag_count + summary_clarity)

Q&A

- Q: How is reward collected?
  A: Via task_log.json → reward_function()

Day 3 — Log → Learn Stub

New file: reinforcement/agent_selector.p

- Add stub: select_agent(task_input, history)
- Start with random choice, track result
- Same for select_model()

Add reinforcement/replay_buffer.py

- Store past runs with: input, selected agent/LLM, reward

Side Note:

Use RL as advice layer — agent still runs if RL fails or is unsure

Day 4 — RL-Driven Selection Test

Nisarg

- Modify MCP routing to try rl_selector.select_agent() if available

Vijay

- Same for LLM router — let RL suggest which model to pick

Nipun

- Log all adapter evaluations into reinforcement/learning_log.json

Day 5 — Self-Improving Feedback Loop

- Simulate 10 sample tasks from CLI (across Archive, PDF, Search)
- Run pipeline with and without RL advice
- Compare output quality

- Save best runs as “mentor logs” for fine-tuning RL later

Day 6 — Integrate + Modularize

- Push RL into bhiv_core/reinforcement module
- Add settings flag: use_rl: true/false
- Document all:
  - agent_selector.py
  - reward_functions.py
  - learning_log.json
- Add notes for future upgrade to full policy-based learning

Deliverables by Day 6:

Nisarg – Agent Orchestration & RL Integration

1. Agent Registry updated for text, PDF, image, and audio agents with dynamic loading
2. MCP Bridge handles multimodal payloads and routes correctly
3. Retry/fallback logic implemented with short-term agent memory
4. Agent-level RL hooks: actions logged to agent_log.json with task metadata

Vijay – LLM Routing, Token Logging & RL Hooks

1. LLM Router built with support for GPT, Gemini, Claude, Grok
2. Token/cost estimator logs tokens used, estimated costs per call
3. RL model resolver: model selections logged to model_log.json
4. Tool integration (e.g. calculator or RAG stub) and LLM chain test

Nipun – Learning Adapter & Reward Layer

1. Learning Adapter returns structured Learning Objects (NLO schema)

2. Schema validation ensures outputs include necessary educational metadata
3. FastAPI preview endpoint for returning NLO JSON
4. Reward function integrates into RL pipeline, computing reward per output

Collective Deliverables by End of Day 6

- CLI runner and FastAPI endpoint triggers complete agent+LLM+adapter pipeline
- Agent Registry supports multimodal agents: text, PDF, image, audio
- LLM Router logs token usage and model decisions
- Learning Adapter returns structured Learning Object JSONs
- MongoDB logging captures full activity, costs, and retry metadata
- RL components operational:
  - agent_selector.py with random + logging logic
  - model_selector.py with bandit-style selection and logging
  - Reward integration via reward_functions.py
- Retry/fallback mechanism in MCP for failed or low-confidence tasks
- Logs files: agent_log.json, model_log.json, and learning_log.json
- Learning Dashboard CLI displays top agents/models and reward heatmaps
- Final documentation including README_RL.md and sample payload specs
- Loom walkthrough demonstrating input → agent/LLM → adapter → logs flow

These deliverables will provide a fully operational BHIV Core, complete with multimodal inputs, adaptive intelligence via RL, and structured educational outputs — ready for Alpha testing and future integration.

Final Integration Strategy

- RL doesn't block or replace current architecture
- It layers intelligence, memory, and tuning over time
- Makes BHIV Core adaptive, not static

Final Note

By starting with logging, not training, we make the RL layer:

- Lightweight
- Optional
- Evolvable

Then later — once data is rich — we can switch from logging intern to learning strategist.

Best of Luck!