1 exp-013-service-pipe

1.0.1 1. Title:

Service Pipe Architecture Evaluation: Composing Services with Unix-Style Pipes with messages formated in JSONL

1.0.2 2. Experiment Number & Status:

- **Number:** `exp-013`
- **Status:** `Proposed`
- **Author(s):** Divan Visagie
- **Date:** 03/06/2025

1.0.3 3. Abstract / Summary:

This experiment proposes to evaluate the effectiveness of composing independent services using Unix-style pipes. It focuses on a lightweight, stream-based protocol leveraging **JSON Lines (JSONL)** with structured message tagging (`"type"` and `"correlationId"`) to enable highly composable, modular, and reconfigurable architectures. Key areas of exploration include efficient message processing, scaling patterns (batch parallelism with `gnu parallel`, multi-client distribution with a `load-balancer` utility), and integrating arbitrary commands via `jsonl-wrap`. The primary demonstration system for this experiment will be a Telegram bot backend pipeline, showcasing end-to-end user request processing.

1.0.4 4. Motivation:

As systems grow in complexity, the need for highly modular, reconfigurable, and language-agnostic components becomes critical. This experiment is motivated by the desire to explore a simple yet powerful paradigm—Unix pipes—as a robust foundation for inter-service communication. By standardizing on JSONL streaming and specific message protocol rules, we aim to reduce coupling, simplify deployment, and enhance the ability to dynamically chain and adapt services, mirroring the composability found in traditional shell scripting but for complex application logic.

1.0.5 5. Goal / Hypothesis:

We hypothesize that a stream-based, JSONL-centric pipe protocol, combined with clear message typing and correlation, will allow independent services to be seamlessly composed and reconfigured through simple command-line operations. This approach is expected to yield highly composable, modular, efficient, and language-agnostic system architectures capable of handling both batch and continuous data streams, as well as enabling various scaling patterns.

1.0.6 6. Methodology:

This experiment involves developing and testing a pipeline where services are chained using Unix pipes, with communication strictly adhering to a defined JSONL streaming protocol.

All findings, unexpected outcomes, performance observations, and debugging insights will be documented in a `logs/` directory with entries dated in ISO format (e.g., `logs/2025-05-31-pipeline-performance.md`, `logs/2025-06-02-load-balancer-behavior.md`). This documentation will capture both successful patterns and failure modes to inform future iterations and architectural decisions.

1.0.6.1 Primary Demonstration System:

The core of this experiment will be proven by implementing and observing the following Telegram bot backend pipeline:

```
telegram-input \
| auth-service \
| capability-dispatcher --capabilities 'canned-responder;llm-
        responder' \
| load-balancer --workers 4 \
| parallel --jobs 4 --pipe --line-buffer 'llm-proxy | response-
        formatter' \
| telegram-output
```

Let’s break down how each service in this specific pipeline adheres to the proposed pipe protocol and contributes to the overall goals:

- **`telegram-input`:** This service acts as the initial ingress point. It will receive incoming messages from the Telegram API, convert them into JSONL format, assign a unique `correlationId` to each request, and set an initial `type` (e.g., `"type": "telegram_message"`). It streams these JSONL messages to `stdout`.
- **`auth-service`:** This service consumes JSONL messages (e.g., `type: "telegram_message"`). It performs authentication/authorization checks. If successful, it passes the message through, potentially enriching it with user details or changing its `type` (e.g., `"type": "authenticated_request"`). If authentication fails, it might emit an error message and/or drop the original message, demonstrating filtering.

- **`capability-dispatcher --capabilities 'canned-responder;llm-responder'`:** This is a key component demonstrating the "Centralized Capability Dispatcher" pattern. It receives `authenticated_request` messages, inspects their content (e.g., user query), and based on predefined logic or `--capabilities` arguments, determines the appropriate next step. It then transforms the message's `type` (e.g., to `"type": "canned_response_request"` or `"type": "llm_request"`) and streams it onward. Messages not matching a capability are passed through unchanged.
- **`load-balancer --workers 4`:** This service assigns worker IDs to incoming messages to ensure proper distribution across parallel workers. It receives messages (e.g., `type: "llm_request"`) and adds a `workerId` field (0-3) using round-robin or hash-based assignment. This ensures that parallel workers can identify which messages are intended for them to process, enabling more sophisticated load distribution than simple line-based splitting.
- **`parallel --jobs 4 --pipe --line-buffer 'llm-proxy | response-formatter'`:** This block showcases "Batch Parallelism with `gnu parallel`".
    - The `capability-dispatcher` will likely output messages of `type: "llm_request"` when an LLM interaction is needed.
    - The `load-balancer` assigns a `workerId` (0-3) to each `llm_request` message, enabling intelligent work distribution.
    - `gnu parallel` will read these worker-assigned `llm_request` messages line by line (`--pipe --line-buffer`). For each incoming `llm_request`, it will launch a new instance (up to 4 concurrent jobs) of the sub-pipeline:
        - **`llm-proxy`:** This service receives `llm_request` JSONL messages, makes the actual API call to the LLM (e.g., Gemini, Ollama), and wraps the LLM's raw output back into a JSONL message (e.g., `type: "llm_raw_response"`), preserving the `correlationId`. This service might involve significant startup costs, making persistent workers (though not directly used here with `parallel`'s per-job process model) a consideration for future iteration.
        - **`response-formatter`:** This service takes `llm_raw_response` messages, processes/formats the LLM's output into a user-friendly format, and sets the final `type` (e.g., `"type": "final_telegram_response"`).
    - `gnu parallel` collects outputs from all concurrent `llm-proxy | response-formatter` sub-pipelines before streaming to `telegram-output`.
- **`telegram-output`:** This final service receives JSONL messages (e.g., `type: "final_telegram_response"`). It converts the structured JSONL data back into a Telegram message format and sends it to the original user via the Telegram API, completing the loop.

1.0.6.2 General Pipe Protocol Adherence:

Beyond this specific pipeline, the experiment will generally ensure:

- **JSON Lines (JSONL) Streaming:** All services emit individual JSON objects, each on a single line, leveraging the newline character (`\n`) as the message boundary.
- **Message Protocol Rules:** Every JSON object transmitted includes:

    - **Rule #1: The Essential `"type"` Field (REQUIRED!):** Used by services to decide whether to process, ignore, or just pass the message through.

    - **Rule #2: The Recommended `"correlationId"` (For Tracing):** Used for end-to-end tracing, debugging, and output ordering with parallelism.

    - **Rule #3: The Optional `"workerId"` Field (For Load Balancing):** Used by load-balancer services to assign messages to specific parallel workers, enabling intelligent work distribution and preventing worker starvation.

- **Processing and Forwarding:** Services inspect the `"type"` field to determine relevance. Additionally, services designed for parallel execution also inspect the `"workerId"` field - if present, the service only processes messages that match its assigned worker ID and ignores all others. Relevant messages are processed and potentially transformed; irrelevant messages are passed through unchanged.

1.0.6.3 Additional Utilities and Patterns Demonstrated:

- The `jsonl-wrap` program will be developed and demonstrated to show how arbitrary shell commands can be integrated into the JSONL pipeline.
- The `load-balancer` utility will be demonstrated in the primary pipeline to show how worker ID assignment enables intelligent message distribution across parallel workers, ensuring optimal load balancing and preventing worker starvation.
- Both "Sequential Filter Chain" and "Centralized Capability Dispatcher" patterns will be conceptually validated through the design of services like `capability-dispatcher`.

1.0.7 7. Success Criteria:

The experiment will be deemed successful if: * The `telegram-input` to `telegram-output` pipeline functions seamlessly, with messages transforming correctly between each service.

- The `type` and `correlationId` fields effectively enable intelligent filtering, routing (via `capability-dispatcher`), and end-to-end tracing across the chained services.

- The `parallel` block successfully demonstrates increased throughput for LLM processing tasks.

- The `jsonl-wrap` utility successfully integrates arbitrary shell commands into the JSONL pipeline.

- The pipeline exhibits high reconfigurability, allowing changes to service order or inclusion via simple command-line modifications without code redeployment.

- Long-running systems and persistent servers can be effectively composed using Unix pipes, demonstrating that the architecture supports both batch processing and continuous service orchestration without requiring service restarts or complex deployment procedures.

1.0.8 8. Potential Challenges / Considerations:

- Ensuring strict JSONL adherence across all services and managing potential parsing errors, especially for malformed input.
- Managing backpressure effectively in complex pipelines, particularly when consuming services are slower than producers.
- Overhead associated with spawning new processes for very short-lived tasks when using `gnu parallel` frequently.
- Complexity of implementing a robust `load-balancer` utility that handles worker failures, graceful shutdowns, and potentially maintains warm connections for external APIs (relevant for future iterations of this base system).
- Debugging complex pipe chains without dedicated tooling that understands `correlationId` for interleaved outputs.

1.0.9 9. Expected Outcome / Learnings:

We expect to validate the significant advantages of this Unix-pipe, JSONL-based approach: * **Composability:** Ability to build powerful pipelines by chaining single-responsibility services.

- **Modularity:** Each service is isolated, allowing independent updates or replacements.

- **Reconfigurability:** Adjust pipeline structure with simple shell commands, eliminating the need for redeployment or re-architecting.

- **Efficiency:** JSONL streaming is low-latency and low-memory, supporting infinite or long-lived data streams.

- **Language Agnosticism:** Any service adhering to the protocol can participate, regardless of implementation language.

- **Leveraging Existing Knowledge:** Complex systems can be configured and orchestrated using familiar shell scripting tools and techniques, reducing the learning curve and allowing teams to apply existing Unix/Linux administration skills.

- **Insights into Scaling:** Learnings on effective strategies for both batch and continuous parallelism using standard Unix tools and custom utilities.

1.0.10 10. Conclusion / Next Steps (Post-Experiment):

The primary outcome of this experiment will be a set of learnings and insights to inform the development of the utility commands and supporting tools necessary to make this architectural pattern feasible for production use. Rather than integrating into an existing system, the focus will be on identifying and building the essential utilities and protocols required for robust, composable, and maintainable pipe-based service architectures. Next steps may include:

- Finalizing core utilities: Complete development and testing of essential tools like `jsonl-wrap` (for integrating arbitrary shell commands into JSONL pipelines) and `load-balancer` (for intelligent worker assignment and distribution).

- Developing a standardized JSONL protocol for service commands: Create a comprehensive specification defining command structures, response formats, error handling, and control messages that enable services to communicate operational commands (start, stop, configure, health-check) through the same JSONL streaming interface.
- Designing standardized JSONL message schemas for common `type` values to ensure interoperability.
- Developing additional helper utilities for monitoring, logging, and debugging JSONL-based pipelines.
- Creating robust error handling and recovery mechanisms for pipe-based architectures.
- Investigating approaches for dynamic service discovery and flexible pipeline orchestration beyond static shell command composition.

1.0.11 11. References / Related Work:

- Unix Philosophy and Inter-Process Communication (IPC) via pipes
- JSON Lines (JSONL) Specification
- GNU Parallel documentation
- Microservices Architecture patterns
- Event-Driven Architecture concepts
- Unix Pipes and Filters pattern