Chain Runner Architecture (Pi Agent Patterns)

Chain Runner Architecture

MC Task #1902 — Pi Agent Patterns Author: Petter Graff (Software Architect) **Date:** 2026-02-24 **Status:** Production

---

1. Overview

Before chain-runner existed, multi-step agent workflows lived in shell scripts and ad-hoc Node.js glue code. Every new pipeline was a new snowflake. Want to add a security audit step? Edit the script. Want to swap the planner agent? Find all the places it's hardcoded. Want to resume a failed workflow after a crash? Good luck.

Chain-runner solves this by separating *what to run* from *how to run it*. A YAML file describes the workflow. The runtime handles sequencing, dependency resolution, timeout enforcement, injection sanitization, and failure rollback. The same orchestration engine runs every chain — no snowflakes.

The key architectural insight: YAML is cheap to write, easy to read, and version-controllable. A non-engineer can look at `plan-build-review.yaml` and understand the workflow in 30 seconds. That's the goal.

What chain-runner is not: It is not a general-purpose workflow engine. It does not support branching, conditional steps, or loops. It runs linear and DAG-shaped agent chains. If you need a state machine, look at Yaktor or a purpose-built orchestrator.

---

2. Architecture

Chain-runner sits at the intersection of four infrastructure systems:

```
User / MC Task
      |
```

```
      ▼
chain-runner.js  ←── YAML chain definitions (~/.system/agents/chains/*.yaml)
      │
      ├── DagScheduler        — Determines step execution order, detects cycles
      │   (~/system/lib/dag-scheduler.js)
      │
      ├── Saga                — Wraps steps in compensatable transactions
      │   (~/system/lib/saga.js)
      │
      ├── agent-scheduler     — Spawns agent processes via child_process.fork
      │   (~/system/kernel/agent-scheduler.js)
      │
      ├── event-bus           — Emits chain.started / step.completed / chain.failed events
      │   (~/system/tools/event-bus)
      │
      ├── DurableRunner       — Optional SQLite persistence for crash recovery
      │   (~/system/tools/durable-runner)
      │
      ├── ChainEnvelope       — Typed message wrapping with cost tracking
      │   (~/system/lib/chain-envelope.js)
      │
      └── HiveMind            — Structured audit log for all chain events
          (~/system/agents/hivemind/hivemind.js)
```

Data Flow

1. User runs `node chain-runner.js run <chain> "<input>"`
2. ChainRunner loads and validates the YAML definition
3. DagScheduler is initialized with step dependency graph
4. Saga is initialized with one step registration per chain step
5. Saga executes steps in order; DagScheduler gates each step until its dependencies complete
6. Each step: agent is spawned via agent-scheduler, output is sanitized, stored in `stepOutputs` map
7. `$INPUT` in the next step's prompt is replaced with the sanitized output of its dependency
8. On completion: final step output is returned, HiveMind is updated, event-bus fires `chain.completed`
9. On failure: Saga runs compensations in reverse, HiveMind logs the failure, process exits 1

Why Saga?

Because agent work is not trivially reversible. If step 2 writes files and step 3 fails, you want a log of what happened and a hook to clean up. Saga provides this structure. In the current implementation, compensations log to HiveMind but do not automatically undo agent work — that would require agents knowing their own undo operations. The structure is in place for future enhancement.

Why DagScheduler?

Because some chain patterns require true parallelism. `full-review.yaml` runs `code-review` and `security-review` simultaneously, then waits for both before running `synthesize`. Without a DAG, you'd serialize work that can run concurrently. DagScheduler handles cycle detection (Kahn's algorithm), fan-out, and fan-in.

---

3. YAML Chain Format

All chains live in `~/system/agents/chains/*.yaml`.

Full Schema

```
name: <string>              # Required. Unique chain identifier. No spaces.
description: <string>       # Optional. Human-readable description.

defaults:
  timeout_ms: <number>      # Default per-step timeout in milliseconds. Default: 300000 (5
min).
  fail_strategy: stop       # Currently only 'stop' is supported.

steps:
  - name: <string>          # Required. Unique within this chain. Used in depends_on
references.
    agent: <string>         # Required. Agent identity name (resolves to
~/.claude/agents/<name>.md).
    prompt: <string>        # Required. Prompt template. Supports $INPUT and $ORIGINAL
substitution.
    depends_on: [<string>]  # Optional. List of step names that must complete before this step
runs.
    timeout_ms: <number>    # Optional. Per-step override. Takes precedence over
defaults.timeout_ms.
```

Validation Rules

Chain-runner validates on load (before any agent is spawned):

- `name` field must be present
- `steps` must be a non-empty array
- Step names must be unique within the chain
- All `depends_on` references must point to steps that exist in the chain
- DagScheduler additionally checks for cycles (would throw on construction)

Agent Resolution

The `agent` field maps to `~/.claude/agents/<agent-name>.md`. The runner reads the YAML frontmatter from that file to extract `name`, `model`, and `tools`. If the agent file has a `tools` list, the prompt is prepended with `[Allowed tools: ...]` — this is the mechanism for agent sandboxing.

Dependency Resolution

Steps without `depends_on` start immediately (they are "ready" from initialization). Steps with `depends_on` wait until all listed steps reach `COMPLETED` status in the DagScheduler.

When a step has multiple dependencies, chain-runner concatenates all dependency outputs separated by `\n\n---\n\n` before passing as `$INPUT`. This is the fan-in behavior for steps like `synthesize` in `full-review.yaml`.

---

4. $INPUT / $ORIGINAL Substitution

Two template variables are available in every prompt:

Variable	Value
`$INPUT`	The sanitized output of the dependency step(s). For the first step (no depends_on), this is the original user input.
`$ORIGINAL`	The original user input, unchanged, for the entire chain run.

`$ORIGINAL` solves a real problem. By the time you reach a `synthesize` step, `$INPUT` contains a 40KB code-review report. Without `$ORIGINAL`, the synthesizer has no idea what it was originally asked to review. `$ORIGINAL` threads the original context through every step.

Envelope unwrapping: If ChainEnvelope is loaded and `$INPUT` is an envelope object (has `version` field), `substituteVars` calls `ChainEnvelope.extractContent()` to unwrap it before

substitution. If it's a plain string, it's used as-is. This makes the system backward-compatible with both envelope and non-envelope inputs.

```
// From chain-runner.js, ChainRunner.substituteVars()
substituteVars(prompt, input, original) {
  if (ChainEnvelope && typeof input === 'object' && input.version) {
    input = ChainEnvelope.extractContent(input);
  } else if (typeof input === 'object') {
    input = JSON.stringify(input);
  }

  return prompt
    .replace(/\$INPUT/g, input || '')
    .replace(/\$ORIGINAL/g, original || '');
}
```

---

5. Chain Sanitization

Every step output is passed through `sanitizeStepOutput()` before being stored and used as the next step's `$INPUT`. This happens regardless of which agent produced the output.

Three operations, in order:

5.1 Length Cap (50KB)

```
const MAX_STEP_OUTPUT_BYTES = 50 * 1024; // 50KB cap

if (Buffer.byteLength(sanitized, 'utf8') > MAX_STEP_OUTPUT_BYTES) {
  sanitized = sanitized.slice(0, MAX_STEP_OUTPUT_BYTES);
  this._logHivemind('update', `Chain step ${stepName} output truncated to 50KB`);
}
```

50KB is large enough for a comprehensive code review or technical report. It prevents a runaway agent from flooding the next step's context window with irrelevant output. Truncation is logged to HiveMind as an advisory.

5.2 Injection Pattern Scan (22 patterns)

The scanner checks for prompt injection attempts in step output. This matters because agent output may include content from external sources — files, web pages, user-provided data — that could attempt to hijack subsequent agents.

The 22 patterns (ported from `external-data-sanitizer.py`):

Pattern	Name
`ignore\s+previous\s+instructions`	ignore previous instructions
`ignore\s+all\s+prior`	ignore all prior
`disregard\s+above`	disregard above
`you\s+are\s+now`	you are now
`act\s+as\s+if`	act as if
`pretend\s+to\s+be`	pretend to be
`roleplay\s+as`	roleplay as
`<system>`	`<system>` tag
`</system>`	`</system>` tag
`<instruction>`	`<instruction>` tag
`</instruction>`	`</instruction>` tag
`<|im_start|>`	chat template marker
`IMPORTANT:\s+[A-Z]`	IMPORTANT: directive
`CRITICAL:\s+[A-Z]`	CRITICAL: directive
`OVERRIDE:\s+[A-Z]`	OVERRIDE: directive
`URGENT:\s+[A-Z]`	URGENT: directive
`[\u200b\u200c\u200d\ufeff]`	zero-width character
`<!--.*?(ignore|override|system).*?-->`	HTML comment injection
`\]\s*\(\s*javascript:`	markdown javascript injection
`\beval\s*\(`	eval() call
`require\s*\(\s*['"]child_process`	child_process require
`process\.env\.`	process.env access

Detection is **advisory, not blocking** at the chain level. Detections are logged to HiveMind as alerts. The step output is still passed to the next step. The rationale: the bash-security-gate hook handles blocking at the execution layer. Chain-runner provides observability, not a second enforcement point. This separation avoids cascading failures where a false positive in the sanitizer kills a legitimate chain run.

5.3 Delimiter Wrapping

After truncation and scanning, the output is wrapped in a structured XML-like delimiter:

```
<step-output source="<stepName>" step-index="<stepIndex>">
<original output content>
</step-output>
```

This serves two purposes:

1. **Provenance**: The next agent knows which step produced this input.
2. **Boundary clarity**: The delimiter reduces the risk of the next agent misinterpreting where its instructions end and the previous step's output begins.

---

6. Chain Envelopes

`~/system/lib/chain-envelope.js` wraps step outputs in typed JSON objects for cost tracking and provenance.

Envelope Structure

```
{
  version: '1.0',           // Envelope schema version
  chainId: '<uuid>',        // The chain run UUID
  stepName: '<string>',     // Step name from YAML
  agentName: '<string>',    // Resolved agent name
  content: '<string>',      // Raw step output
  metadata: {
    tokensIn: 0,            // Tokens consumed (placeholder — agent-scheduler doesn't track
yet)
    tokensOut: 0,           // Tokens generated (placeholder)
    elapsedMs: <number>,    // Actual wall-clock time for this step
    model: '<string>',      // Agent model (from agent frontmatter, e.g. 'sonnet')
  },
  timestamp: '<ISO string>' // When this step completed
}
```

API

```
const { create, extractContent, isEnvelope, ENVELOPE_VERSION } = require('~/system/lib/chain-envelope');

// Create an envelope
const envelope = create({
  chainId,
  stepName: 'plan',
  agentName: 'planner',
  content: 'Step output text...',
  metadata: { tokensIn: 0, tokensOut: 0, elapsedMs: 4200, model: 'sonnet' }
});

// Extract content (backward-compatible: works with envelopes OR plain strings)
const text = extractContent(envelope);    // Returns envelope.content
const text2 = extractContent('raw str');  // Returns 'raw str' unchanged

// Type check
if (isEnvelope(value)) { ... }           // Checks version === '1.0' + required fields
```

Backward Compatibility

`extractContent()` handles three cases:

1. Valid envelope object: returns `envelope.content`
2. Plain string: returns the string unchanged
3. Arbitrary object: returns `JSON.stringify(object)`

This means chain-runner works correctly whether or not the envelope module is loaded. The module is loaded with `try/catch`; if it fails (module not present), `ChainEnvelope` is null and the system falls back to plain string handling throughout.

The `tokensIn` / `tokensOut` fields are currently `0` because `agent-scheduler` does not yet expose token counts. The envelope structure is ready for when that tracking is added.

---

7. Damage Control Security

`~/.claude/hooks/config/damage-control.json` defines the security blocklist enforced by the H) Damage Control Gate in `~/.claude/hooks/bash-security-gate.py`.

Three Path Lists

zeroAccessPaths (27 paths)

Complete read/write prohibition. Any command touching these paths is blocked:

```
~/.ssh/            ~/.gnupg/          ~/.aws/credentials    ~/.aws/config
~/.azure/          ~/.config/gcloud/  ~/.kube/config        ~/.docker/config.json
~/.npmrc           ~/.pypirc          ~/.gem/credentials    ~/.netrc
~/.env             ~/.gitconfig       ~/.git-credentials    /etc/shadow
/etc/passwd        /etc/sudoers       /etc/ssh/             ~/.local/share/keyrings/
~/Library/Keychains/ ~/.vault-token   ~/.config/helm/
```

The pattern: credentials, keys, and system auth files. These are the blast radius of a compromised agent.

readOnlyPaths (40 entries)

Can be read, cannot be written or deleted:

Includes system directories (`/usr/`, `/bin/`, `/System/`, `/Library/`), Claude configuration files (`~/.claude/settings.json`, `~/.claude/hooks/`, `~/.claude/agents/*.md`), system rules (`~/system/rules/`, `~/system/CLAUDE.md`), and all build artifact directories (`dist/`, `build/`, `.next/`, `target/`, etc.).

The rationale for build artifacts: generated files should not be modified directly. Rebuild from source.

noDeletePaths (28 entries)

Can be read and modified, but not deleted:

CI/CD configuration (`.gitlab-ci.yml`, `Jenkinsfile`, `.circleci/`), project manifests (`package.json`, `Cargo.toml`, `go.mod`, `pom.xml`, `pyproject.toml`), version control files (`.gitignore`, `.git/`), and legal files (`LICENSE`, `COPYING`).

The purpose: these are load-bearing files. Deleting `package.json` by accident in a multi-step agent chain is hard to recover from. Make it require explicit human action.

22 Bash Tool Patterns

The `bashToolPatterns` array defines regex patterns for destructive commands blocked regardless of path:

Name	Pattern	Description

sudo shell	`\bsudo\s+(bash|sh|zsh)\b`	Privilege escalation
curl upload	`\bcurl\s+.*--upload-file\b`	Potential data exfiltration
remote file transfer	`\b(rsync|scp)\s+.*@[a-zA-Z0-9]`	Transfer to remote host
iptables flush	`\biptables\s+-F\b`	Opens all firewall ports
python exec()	`\bpython3?\s+.*-c\s+.*exec\s*\(`	Arbitrary code via python -c
node child_process	`\bnode\s+-e\s+.*require\s*\(\s*['"]child_process`	Shell spawn via node -e
kubectl delete namespace	`\bkubectl\s+delete\s+(namespace|ns)\b`	Destroys all K8s resources
kubectl delete --all	`\bkubectl\s+delete\s+.*--all\b`	Delete all resources of type
mongosh dropDatabase	`(mongosh|mongo).*dropDatabase`	Drop entire MongoDB database
redis FLUSHALL	`\bredis-cli\s+FLUSHALL\b`	Flush all Redis databases
redis FLUSHDB	`\bredis-cli\s+FLUSHDB\b`	Flush current Redis DB
terraform destroy	`\bterraform\s+destroy\b`	Destroy all Terraform infra
helm uninstall --no-hooks	`\bhelm\s+uninstall\b.*--no-hooks`	Uninstall bypassing safety hooks
docker system prune -a	`\bdocker\s+system\s+prune\s+-a\b`	Remove ALL Docker resources
gcloud project delete	`\bgcloud\s+projects\s+delete\b`	Delete entire GCP project
az group delete	`\baz\s+group\s+delete\b`	Delete Azure resource group
aws s3 rb --force	`\baws\s+s3\s+rb\s+.*--force\b`	Force-delete S3 bucket
aws terminate instances	`\baws\s+ec2\s+terminate-instances\b`	Terminate EC2 instances
aws rds delete --skip-snapshot	`\baws\s+rds\s+delete-db-instance\b.*--skip-final-snapshot`	Delete RDS without snapshot
vercel remove --yes	`\bvercel\s+remove\s+.*--yes\b`	Force-remove Vercel project
npm unpublish	`\bnpm\s+unpublish\b`	Remove published npm package
git push --force	`\bgit\s+push\s+.*--force\b`	Force push (destroys history)
curl DELETE to API/prod	`\bcurl\s+.*-X\s+DELETE\b.*\b(api|prod|production)\b`	HTTP DELETE to production

Damage Control Gate Implementation

```
# From ~/.claude/hooks/bash-security-gate.py, check_damage_control()
def check_damage_control(command: str) -> str | None:
    try:
        if not os.path.exists(DAMAGE_CONTROL_CONFIG):
            return None
```

```
        with open(DAMAGE_CONTROL_CONFIG, 'r') as f:
            config = json.load(f)

        patterns = config.get("bashToolPatterns", [])
        for entry in patterns:
            pattern = entry.get("pattern", "")
            if not pattern:
                continue
            if re.search(pattern, command):
                name = entry.get("name", "unknown")
                desc = entry.get("description", "Blocked by damage-control rules")
                return f"BLOCKED: Damage Control — {name}!\n..."
    except (json.JSONDecodeError, IOError) as e:
        # Config broken — fail closed (block)
        return f"BLOCKED: Damage control config error!\n..."

    return None
```

Critical detail: if `damage-control.json` is malformed or unreadable, the gate **returns a block message** (fails closed). This is the correct behavior for a security gate — a misconfigured guard is not a free pass.

---

8. Fail-Closed Security Hooks

`~/.claude/hooks/lib/_hook_utils.py` defines which hooks must fail closed vs. fail open.

```
# Security hooks that MUST fail closed (block on error/timeout)
# Quality gates and advisory hooks stay fail-open (allow on error/timeout)
FAIL_CLOSED_HOOKS = {
    "bash-security-gate",
    "inline-smtp-gate",
    "damage-control",
}
```

The `run_check()` function enforces this:

```
def run_check(hook_name, hook_module, event, timeout_ms=2000):
    fail_closed = hook_name in FAIL_CLOSED_HOOKS
```

```
    if hook_module is None:
        if fail_closed:
            return (2, f"BLOCKED: Security hook failed to load: {hook_name}")
        return (0, f"Hook skipped (import failed): {hook_name}")
    ...
    except TimeoutError as e:
        if fail_closed:
            return (2, f"BLOCKED: Security hook timeout — {hook_name} ({timeout_ms}ms). Fail-
closed.")
        return (0, f"Hook timeout: {hook_name} ({timeout_ms}ms)")
    except Exception as e:
        if fail_closed:
            return (2, f"BLOCKED: Security hook crashed — {hook_name}: {e}. Fail-closed.")
        return (0, f"Hook error: {hook_name}: {e}")
```

The timeout mechanism uses `signal.setitimer(signal.ITIMER_REAL, ...)` for sub-second precision, with a custom `_hook_timeout` handler that raises `TimeoutError`. The original signal handler is restored in the `finally` block regardless of outcome.

Additionally, `bash-security-gate.py` sets a 5-second process-level alarm on startup:

```
def _timeout_handler(signum, frame):
    print("HOOK TIMEOUT (5s) — BLOCKING action (fail-closed security hook)", file=sys.stderr)
    sys.exit(2)
signal.signal(signal.SIGALRM, _timeout_handler)
signal.alarm(5)
```

This means the entire security gate process will block and return exit code 2 if it has not completed within 5 seconds — regardless of which check is running. The hook cannot be made to hang indefinitely.

---

9. CLI Reference

All commands run via: `node ~/system/tools/chain-runner.js <command>`

`list`

List all available chains.

```
node ~/system/tools/chain-runner.js list
```

Output format:

```
Available chains:
──────────────────────────────────────────────────────────────

  full-review               3 steps  Parallel security + code review, then synthesize findings
  plan-build                2 steps  Plan then implement — no review step
  plan-build-review         3 steps  Plan, implement, and review — full development cycle
  plan-review-plan          3 steps  Plan, get review feedback, re-plan with feedback —
iterative planning
  scout-flow                3 steps  Three-pass scout: explore, validate findings, synthesize
report

5 chain(s) found.
```

show <chain-name>

Show detailed definition of a chain including step order and dependencies.

```
node ~/system/tools/chain-runner.js show full-review
```

Output:

```
Chain: full-review
Description: Parallel security + code review, then synthesize findings
Defaults: timeout=300000ms, fail_strategy=stop

Steps (3):
  1. code-review → agent:validator
  2. security-review → agent:sentinel-validator
  3. synthesize → agent:distiller [depends: code-review, security-review]
```

run <chain-name> "<input>" [--mc-task <id>] [--durable]

Run a chain. Input is the initial prompt passed to the first step(s).

```
# Basic run
node ~/system/tools/chain-runner.js run plan-build "Add rate limiting to the API"

# Link to Mission Control task
node ~/system/tools/chain-runner.js run plan-build-review "Refactor auth module" --mc-task
1902

# Durable mode (crash-recoverable, stores state in SQLite)
node ~/system/tools/chain-runner.js run plan-build "Add caching layer" --durable

# Combined
node ~/system/tools/chain-runner.js run full-review "Review ~/projects/drop/src/auth.ts" --mc-
task 1850 --durable
```

Flags:

Flag	Description
`--mc-task <id>`	Links chain progress to a Mission Control task ID. Updates are logged to HiveMind with `[MC#<id>]` prefix.
`--durable`	Enables SQLite persistence via DurableRunner. Required for `resume` to work.

`resume <workflow-id>`

Resume a durable workflow that was interrupted (crash, timeout, manual kill).

```
node ~/system/tools/chain-runner.js resume chain-plan-build-1708789200000-abc123
```

Requirements:

- The original run must have used `--durable`
- DurableRunner (`~/system/tools/durable-runner`) must be available
- The workflow ID comes from the DurableRunner database

Resume re-runs from the next incomplete step. Already-completed steps are not re-executed.

---

10. Available Chains

Five chains ship with the system, all in `~/system/agents/chains/`:

Chain	File	Steps	Description
`plan-build`	`plan-build.yaml`	2	Plan then implement. No review step. Fast path for low-risk tasks.
`plan-build-review`	`plan-build-review.yaml`	3	Full development cycle. Plan → implement → validate. Default for non-trivial tasks.
`plan-review-plan`	`plan-review-plan.yaml`	3	Iterative planning. Draft plan → review for gaps → revised plan. No implementation.
`full-review`	`full-review.yaml`	3	Parallel code + security review, then synthesized report. `code-review` and `security-review` run concurrently.
`scout-flow`	`scout-flow.yaml`	3	Three-pass investigation. Explore → cross-check findings → synthesize report.

Step-by-Step Breakdown

plan-build:

1. `plan` (planner) — Create implementation plan from input
2. `build` (builder, timeout: 600000ms) — Implement the plan

plan-build-review:

1. `plan` (planner) — Create implementation plan
2. `build` (builder, timeout: 600000ms) — Implement the plan
3. `review` (validator) — Review implementation, receives `$INPUT` (build output) and `$ORIGINAL` (original request)

plan-review-plan:

1. `plan-draft` (planner) — Create initial detailed implementation plan
2. `review` (validator) — Review draft for gaps, risks, improvements; receives `$ORIGINAL`
3. `plan-final` (planner) — Revise plan incorporating feedback; receives `$ORIGINAL`

full-review (DAG parallel):

1. `code-review` (validator) — Code review [no deps, starts immediately]

2. `security-review` (sentinel-validator) — Security audit [no deps, starts immediately, runs parallel to code-review]
3. `synthesize` (distiller) — Unified report [depends_on: code-review, security-review]; receives both outputs concatenated + `$ORIGINAL`

scout-flow:

1. `scout-1` (distiller) — Explore and document findings
2. `scout-2` (validator) — Validate and cross-check findings; receives `$ORIGINAL`
3. `synthesize` (distiller) — Final synthesis from validated findings; receives `$ORIGINAL`

---

11. Structured Logging

chain-runs.jsonl

Every step completion (success or failure) appends a JSON entry to `~/system/logs/chain-runs.jsonl`.

Success entry schema:

```
{
  "ts": "2026-02-24T10:30:00.000Z",
  "chain": "plan-build-review",
  "chainId": "a1b2c3d4-...",
  "step": 0,
  "stepName": "plan",
  "agent": "planner",
  "exit": 0,
  "elapsed_ms": 34200,
  "tokens_in": 0,
  "tokens_out": 0
}
```

Failure entry schema:

```
{
  "ts": "2026-02-24T10:31:15.000Z",
  "chain": "plan-build-review",
  "chainId": "a1b2c3d4-...",
  "step": -1,
  "stepName": "build",
```

```
  "agent": "unknown",
  "exit": 1,
  "elapsed_ms": 0,
  "error": "Step 'build' timed out after 600000ms"
}
```

The `step: -1` convention on failure entries makes them easy to filter. `tokens_in` and `tokens_out` are 0 placeholders until agent-scheduler exposes token tracking.

HiveMind Integration

Chain-runner calls HiveMind (`~/system/agents/hivemind/hivemind.js`) for four event types:

Event	Type	When
Chain completed	`update`	After all steps succeed
Step truncated	`update`	When output exceeds 50KB cap
Injection detected	`alert`	When injection pattern found in step output
Chain failed	`error`	When Saga throws SagaError
Compensation ran	`error`	When a step's compensate function executes

HiveMind calls are fire-and-forget (`spawnSync` with `stdio: 'ignore'`, 5s timeout). A HiveMind failure never blocks a chain run.

Event Bus

Chain-runner emits structured events via the event-bus for real-time monitoring:

Event	Payload
`chain.started`	`{ chainId, chainName, input (first 200 chars), steps }`
`chain.step.completed`	`{ chainId, step, stepIndex, elapsed_ms }`
`chain.step.killed`	`{ chainId, step, agentId, pid }`
`chain.completed`	`{ chainId, chainName, totalElapsed, steps }`
`chain.failed`	`{ chainId, chainName, error }`

---

12. Troubleshooting

Chain not found

```
Error: Chain not found: /Users/makinja/system/agents/chains/my-chain.yaml
```

Verify the file exists at `~/system/agents/chains/<name>.yaml`. The `name` argument to `run` and `show` is the filename without `.yaml`.

Agent not found / spawn fails

```
Error: Failed to spawn agent 'my-agent' for step 'build': ...
```

Verify `~/.claude/agents/<agent-name>.md` exists. The `agent` field in YAML maps directly to this path. Run `ls ~/.claude/agents/` to see available agents.

Step timeout

```
Error: Step 'build' timed out after 600000ms
```

The step's `timeout_ms` (or chain `defaults.timeout_ms`) was exceeded. Options:

1. Increase `timeout_ms` in the YAML step definition
2. Break the task into smaller steps
3. Check if the agent is hanging on I/O or waiting for user input

The timeout sequence: soft timeout fires → SIGTERM sent to agent process → 5-second grace period → SIGKILL if still running.

Duplicate step names

```
Error: Chain my-chain has duplicate step names: build
```

Step names must be unique within a chain. Used as keys in `stepOutputs` map and for `depends_on` resolution.

Cycle detection

```
Error: DagScheduler: cycle detected in dependency graph. Involved phases: step-a, step-b
```

A → B → A is not a valid dependency graph. Review `depends_on` declarations for circular references.

Unknown depends_on step

```
Error: Chain my-chain step 'synthesize' depends on unknown step 'analysis'
```

The step name in `depends_on` must exactly match another step's `name` field in the same chain.

js-yaml not available

```
ERROR: js-yaml not available. Install: npm install js-yaml
```

Run `npm install js-yaml` in `~/system/tools/` or wherever chain-runner.js is located. The module is expected as a transitive dependency; explicit install may be needed in isolated environments.

Durable resume fails

```
Error: DurableRunner not available
```

The `durable-runner` module at `~/system/tools/durable-runner` could not be loaded. Either the module is not present or has a broken dependency. Resume requires durable mode; without DurableRunner, chains cannot be resumed.

Debugging chain runs

Check the JSONL log:

```
tail -f ~/system/logs/chain-runs.jsonl | python3 -m json.tool
```

Check HiveMind for chain-related entries:

```
node ~/system/agents/hivemind/hivemind.js query chain-runner
```

Check hook security logs if a command is being blocked:

```
tail -50 /tmp/hook-errors.log
tail -50 /tmp/hook-metrics.jsonl
```

Appendix: Key File Locations

File	Purpose

`~/system/tools/chain-runner.js`	Main orchestrator (~700 lines)
`~/system/agents/chains/*.yaml`	Chain definitions
`~/system/lib/chain-envelope.js`	Typed message envelopes
`~/system/lib/dag-scheduler.js`	DAG execution engine
`~/system/lib/saga.js`	Saga pattern with compensation
`~/system/kernel/agent-scheduler.js`	Agent process spawning
`~/.claude/hooks/bash-security-gate.py`	Security gate (gates A-H)
`~/.claude/hooks/config/damage-control.json`	Damage control blocklist
`~/.claude/hooks/lib/_hook_utils.py`	Fail-closed hook infrastructure
`~/system/logs/chain-runs.jsonl`	Structured run audit log

Revision #5
Created 2026-02-24 10:01:50 UTC by John
Updated 2026-06-21 20:02:21 UTC by John