OPENCLAW

Machine Learning Operations Guide

How to build autonomous AI agents that learn, compound, and self-optimize — without retraining a single model

---

ScaleUP Media · THE SPRINT Program · @mattganzak

Version 2.0 · For internal use and SPRINT members

Table of Contents

PART I

FOUNDATIONS

Chapter 1: The Machine Learning Mindset for OpenClaw

Most people using OpenClaw treat it like a sophisticated task executor. They build a workflow, run it, check the output, maybe tweak a prompt when something breaks, and repeat. That is automation — not intelligence.

Machine learning in OpenClaw is a fundamentally different operating philosophy. Instead of running tasks and hoping they work well, you instrument every action, measure every outcome, and systematically use that data to make the next run better than the last. The model itself never changes — what changes is the intelligence you build around it.

> **The Core Insight**
>
> Claude, Haiku, and Sonnet are static models. They cannot learn from your specific use case on their own.
>
> But the SYSTEM you build around them — the prompts, routing rules, examples, and context — is infinitely
>
> trainable. OpenClaw ML Ops is the discipline of training that system, not the model.

1.1 Why Standard Automation Hits a Ceiling

Standard automation plateaus. You launch a cold outreach agent. It works at maybe 4% reply rate. You tweak the prompt a few times, get it to 6%, then it stagnates. You accept 6% as 'good enough' because you have no systematic way to push further. This is the automation ceiling — and virtually every OpenClaw deployment hits it within 30-60 days.

The ceiling exists because standard automation has no memory. Each run is isolated. The agent does not know that the last 200 runs happened. It cannot know that messages opening with a question got 3x more replies than messages opening with a compliment. Without memory, there is no learning. Without learning, there is no compounding.

1.2 The ML Ops Difference

An ML Ops deployment treats every agent run as a training event. The run happens, the outcome is measured, the measurement is stored, patterns are extracted, and the agent configuration is updated. After 200 runs, the agent has effectively been trained on 200

real-world examples from your specific market, niche, and use case. No competitor starting today can replicate that without running the same 200 cycles.

Dimension	Standard Automation	ML Ops Deployment
Memory	None — each run is isolated	Persistent — every run feeds forward
Improvement	Manual and infrequent	Autonomous and continuous
Performance	Plateaus within weeks	Compounds indefinitely
Cost	Fixed or creeping upward	Decreases as routing sharpens
Failure handling	Silent failures accumulate	Failures are flagged and learned from
Competitive moat	None — replicable immediately	Deep — requires months of data to replicate
Operator time	High — constant manual tuning	Low — system self-maintains

1.3 What 'Machine Learning' Actually Means Here

To be precise: we are not doing gradient descent. We are not fine-tuning model weights. What we are doing is a form of in-context learning combined with automated prompt engineering — which, in practice, produces the same business outcomes as traditional ML at a fraction of the cost and complexity.

The three mechanisms that drive learning in an OpenClaw ML Ops system:

- **Prompt Evolution:** System prompts are rewritten over time based on observed performance data. The prompt for your outreach agent in month 3 is not the same as the one you wrote on day 1 — it has been refined through dozens of optimization cycles using real outcome data.
- **Example Injection:** High-performing outputs are stored and injected into future prompts as few-shot examples. Low-performing outputs are stored as negative examples. The model's behavior shifts because its context shifts.
- **Routing Intelligence:** The system learns which tasks genuinely require Sonnet versus Haiku, which contexts need more tokens, and which configurations produce the best cost-to-quality ratio. This routing intelligence sharpens with every run.

1.4 The Mindset Shift Required

Building an ML Ops system in OpenClaw requires a different way of thinking about your agents. Before you write a single workflow, you need to answer four questions:

1. **What is the single measurable outcome that defines success for this agent?** Not 'good quality output' — a specific, numeric metric tied to business value.
2. **How will I know if that outcome occurred?** Identify the exact data source and the exact signal that tells you whether the agent succeeded or failed.
3. **How long after the agent runs will I know the outcome?** Outreach replies come within 24-72 hours. Content engagement accumulates over 7 days. Support resolution is immediate. Your feedback loop cycle time depends on this lag.
4. **What data from the agent run itself might predict that outcome?** This is the feature set your pattern analyzer will use. Think about it before you build the logger.

Action Before You Proceed

Before moving to Chapter 2, write down your answers to the four questions above for your highest-volume OpenClaw agent. These answers will drive every architectural decision in the chapters that follow. Do not skip this step — it is the most important work in this entire guide.

Chapter 2: How Feedback Loops Work — The Theory

A feedback loop is a closed system where outputs influence future inputs. In engineering, this is called a control system. In biology, it is called homeostasis. In machine learning, it is called reinforcement learning. In OpenClaw, it is called a performance loop — and it is the core architecture of everything in this guide.

2.1 The Basic Loop Structure

Every feedback loop, regardless of complexity, follows the same four-phase cycle:

The Four Phases of Every Feedback Loop

PHASE 1 — ACT: The agent performs a task and produces an output
PHASE 2 — MEASURE: The outcome of that output is observed and scored
PHASE 3 — ANALYZE: Patterns are extracted from accumulated measurements
PHASE 4 — ADJUST: The agent configuration is updated based on patterns

Then the cycle repeats — each iteration informed by all previous iterations.

The power of the loop comes from two properties: accumulation and compounding. Each cycle adds to the store of knowledge, and each improvement in agent performance makes the next cycle's data higher quality, which drives better improvements, which drives higher quality data — a compounding spiral upward.

2.2 Positive vs. Negative Feedback

Not all feedback loops improve performance. There are two types:

- **Positive feedback loops** amplify a signal — small improvements lead to bigger improvements, which lead to even bigger improvements. This is what you want from a performance loop: early wins compound into large gains.
- **Negative feedback loops** dampen a signal — deviations from a target are corrected. This is what you want from your anomaly detection and cost control systems: prevent runaway costs or quality degradation from spiraling out of control.

A well-designed OpenClaw ML Ops system uses both. The performance loop is a positive feedback loop (amplify what works). The guardrail system is a negative feedback loop (correct deviations from cost and quality targets).

2.3 Feedback Lag and Loop Cadence

One of the most important and most underappreciated aspects of feedback loop design is lag — the time between when the agent acts and when you can observe the outcome. Lag determines your minimum loop cycle time, and getting it wrong is the most common cause of loop failure.

Agent Type	Outcome Lag	Minimum Loop Cadence	Cadence Rationale
Cold outreach	24-72 hours	Weekly	Need 50+ runs to get statistical signal
Content posting	48-168 hours	Bi-weekly	Engagement accumulates over 7 days
Lead qualification	7-30 days	Monthly	Close rate lag requires longer window
Support resolution	0-4 hours	3x per week	Fast feedback enables faster cycles
Data processing	Immediate	Daily	Can run tight feedback cycles
Ad copy generation	24-48 hours	Weekly	CTR/CVR data needs time to accumulate

The practical rule: your loop cadence should be long enough to collect at least 30-50 outcomes before each analysis cycle. Optimizing on fewer data points than this produces noisy, unreliable signals that will cause the optimizer to make changes that hurt rather than help.

2.4 The Signal-to-Noise Problem

Raw outcome data is noisy. On any given day, your outreach agent might perform worse because of a public holiday, a news event, or random variation — not because your prompt is getting worse. If your pattern analyzer sees this noise and treats it as signal, it will make harmful changes.

Three techniques to manage signal-to-noise in your loop:

5. **Minimum sample thresholds:** Never run the optimizer on fewer than 50 scored outcomes per agent. Set this as a hard rule in the Loop Orchestrator.
6. **Rolling averages:** Compare the last 14-day average against the prior 14-day average. Point-in-time comparisons are too noisy. Rolling windows smooth out daily variation.

7. **Statistical confidence gates:** Only trigger the optimizer if the performance delta between prompt versions exceeds a minimum threshold (suggest: 10% relative change). Small fluctuations are noise, not signal.

2.5 Cold Start Problem

Every feedback loop faces a cold start — the period before you have enough data to run the analyzer meaningfully. This is not a failure state. It is a data collection phase. The right approach during cold start:

- Run the agent normally — focus on logging quality, not optimization
- Manually review outputs for the first 20-30 runs to calibrate your scoring rubric
- Do NOT run the optimizer until you have 50+ scored outcomes
- Use the cold start period to validate that your outcome collector is working correctly
- Expected cold start duration: 1-3 weeks depending on agent volume

Cold Start Milestone

You are out of cold start when you have: (1) 50+ scored outcomes in your memory store, (2) at least 2 different prompt versions with measurable performance data, and (3) outcome collection working reliably for your specific downstream tool. Do not rush this phase. Data quality now determines loop quality for months.

Chapter 3: The Five-Layer Architecture

The OpenClaw ML Ops system is organized into five distinct layers. Each layer has a specific responsibility, and they communicate through structured data contracts — defined schemas that ensure each layer can evolve independently without breaking the others.

The Five Layers

LAYER 1 — EXECUTION: Your operational agents doing real work (outreach, content, support, etc.)
LAYER 2 — OBSERVATION: Action Logger + Outcome Collector capturing every signal
LAYER 3 — MEMORY: Persistent store of all runs, scores, and analysis outputs
LAYER 4 — INTELLIGENCE: Pattern Analyzer + Prompt Optimizer extracting and applying insights
LAYER 5 — CONTROL: Loop Orchestrator managing scheduling, health, and safety rails

3.1 Layer Responsibilities

Understanding what each layer is and is not responsible for prevents the most common architectural mistakes:

Layer	Responsible For	NOT Responsible For
Execution	Running tasks, producing outputs, calling tools	Logging, scoring, or self-improving
Observation	Capturing complete run context, collecting outcomes	Analyzing patterns or making changes
Memory	Storing, indexing, and serving data reliably	Interpreting or acting on data
Intelligence	Analyzing patterns, generating improved configurations	Deploying changes or scheduling itself
Control	Scheduling, health monitoring, anomaly detection, escalation	Analysis, optimization, or execution

3.2 Data Contracts Between Layers

The layers communicate through two primary data schemas. Every component in the system must produce and consume these schemas consistently. Do not deviate from them without updating all dependent components.

Schema A — Task Run Record (Layer 2 → Layer 3)

Every completed agent run produces one Task Run Record written to the memory store:

```
{
  // Identity
  task_id:              UUID v4 (generated at log time),
  run_timestamp:        ISO 8601 UTC,
  agent_name:           string (e.g. 'outreach-agent-v2'),
  prompt_version:       string (e.g. 'v2.4'),

  // Execution context
  model_used:           string (e.g. 'claude-haiku-3-20240307'),
  input_tokens:         integer,
  output_tokens:        integer,
  cost_usd:             float (6 decimal places),
  execution_time_ms:    integer,
  task_type:            enum ['outreach','content','support','research',
                              'data_processing','qualification','other'],

  // Content
  input_preview:        string (first 200 chars of input),
  output_preview:       string (first 300 chars of output),
  output_word_count:    integer,

  // Metadata (task-specific key-value pairs)
  metadata:             object (campaign_id, lead_id, content_type, etc.),

  // Outcome (filled by Outcome Collector, null at log time)
  outcome_score:        integer 0-100 | null,
  outcome_data:         object | null,
  outcome_collected_at: ISO 8601 UTC | null,
  outcome_source:       string | null
}
```

Schema B — Analysis Report (Layer 4 → Layer 3 + Layer 5)

Every Pattern Analyzer run produces one Analysis Report:

```
{
  // Identity
  analysis_id:          UUID v4,
  analysis_timestamp:   ISO 8601 UTC,
  agent_name:           string,
  records_analyzed:     integer,
  date_range_start:     ISO 8601 date,
  date_range_end:       ISO 8601 date,

  // Performance summary
  avg_outcome_score:    float,
```

```
  score_vs_prior_period: float (delta, positive = improvement),
  avg_cost_per_run:      float,
  cost_vs_prior_period: float (delta, negative = improvement),
  total_runs:            integer,
  scored_runs:           integer,
  scoring_coverage_pct: float,

  // Intelligence outputs
  top_wins:              [{ pattern, evidence, recommendation, priority,
expected_impact }],
  top_failures:          [{ pattern, evidence, recommendation, priority,
expected_impact }],
  cost_opportunities:    [{ task_type, current_model, suggested_model,
est_savings_pct }],
  prompt_rankings:       { version: avg_score } object,
  best_prompt_version:  string,
  worst_prompt_version: string,

  // Meta
  overall_health_score: integer 0-100,
  optimization_warranted: boolean,
  executive_summary:     string (3-4 sentences),
  confidence_level:      enum ['low','medium','high']
}
```

3.3 Technology Stack Options

You have choices at each layer. The table below maps the options from simplest to most robust. Start simple — upgrade the stack as your volume and requirements grow.

Layer	Starter Stack	Growth Stack	Production Stack
Memory Store	Google Sheets	Airtable or Notion DB	Supabase (PostgreSQL)
Action Logger	OpenClaw webhook → Sheets	OpenClaw webhook → Supabase	OpenClaw webhook → Supabase + S3
Outcome Collector	Manual entry + polling agent	Automated polling agent	Real-time webhooks from all sources
Pattern Analyzer	Scheduled agent, JSON output	Agent + Python analytics	Agent + dbt + analytics DB
Prompt Optimizer	Agent output → manual review	Agent output → Slack approval	Agent output → auto-deploy on confidence:high
Orchestrator	Cron triggers in OpenClaw	OpenClaw + n8n/Make	Dedicated orchestration layer

Recommendation: Start with Google Sheets for memory and manual outcome scoring for the first 2 weeks. The discipline of manually reviewing outcomes will calibrate your scoring rubric better than any automated system. Upgrade to Supabase when you hit 500+ records or need to query the data programmatically.

PART II

DATA INFRASTRUCTURE

Chapter 4: Building Your Memory Store

The memory store is the nervous system of your ML Ops deployment. Everything else depends on it. A poorly designed memory store creates data quality problems that corrupt your pattern analysis and cause the optimizer to make harmful changes. Build it right the first time.

4.1 Supabase Setup (Recommended)

Supabase is the recommended memory store because it gives you a real PostgreSQL database with a REST API that OpenClaw can write to directly via webhook, plus a dashboard for manual inspection, plus the ability to run SQL queries for ad hoc analysis. The setup takes approximately 20 minutes.

Step 1 — Create the Supabase Project

8. Go to supabase.com and create a free project
9. Name it 'openclaw-ml' or similar — keep it separate from any application databases
10. Save your project URL and anon key — you will need these for OpenClaw webhook configuration
11. Enable Row Level Security (RLS) — you will configure it to allow OpenClaw service role writes

Step 2 — Create the agent_task_log Table

Run this SQL in the Supabase SQL editor:

```
CREATE TABLE agent_task_log (
  id                  UUID DEFAULT gen_random_uuid() PRIMARY KEY,
  task_id             UUID NOT NULL UNIQUE,
  run_timestamp       TIMESTAMPTZ NOT NULL DEFAULT NOW(),
  agent_name          TEXT NOT NULL,
  prompt_version      TEXT NOT NULL,
  model_used          TEXT NOT NULL,
  input_tokens        INTEGER NOT NULL DEFAULT 0,
  output_tokens       INTEGER NOT NULL DEFAULT 0,
  cost_usd            NUMERIC(12, 6) NOT NULL DEFAULT 0,
  execution_time_ms   INTEGER,
  task_type           TEXT NOT NULL DEFAULT 'other',
  input_preview       TEXT,
  output_preview      TEXT,
```

```
  output_word_count    INTEGER,
  metadata             JSONB DEFAULT '{}',
  outcome_score        INTEGER CHECK (outcome_score BETWEEN 0 AND 100),
  outcome_data         JSONB,
  outcome_collected_at TIMESTAMPTZ,
  outcome_source       TEXT,
  created_at           TIMESTAMPTZ DEFAULT NOW()
);

-- Indexes for common query patterns
CREATE INDEX idx_task_log_agent     ON agent_task_log(agent_name);
CREATE INDEX idx_task_log_timestamp ON agent_task_log(run_timestamp DESC);
CREATE INDEX idx_task_log_version   ON agent_task_log(prompt_version);
CREATE INDEX idx_task_log_score     ON agent_task_log(outcome_score);
CREATE INDEX idx_task_log_unscored  ON agent_task_log(outcome_score)
  WHERE outcome_score IS NULL;
```

Step 3 — Create the pattern_analysis_log Table

```
CREATE TABLE pattern_analysis_log (
  id                     UUID DEFAULT gen_random_uuid() PRIMARY KEY,
  analysis_id            UUID NOT NULL UNIQUE,
  analysis_timestamp     TIMESTAMPTZ NOT NULL DEFAULT NOW(),
  agent_name             TEXT NOT NULL,
  records_analyzed       INTEGER NOT NULL,
  date_range_start       DATE NOT NULL,
  date_range_end         DATE NOT NULL,
  avg_outcome_score      NUMERIC(5,2),
  score_delta            NUMERIC(5,2),
  avg_cost_per_run       NUMERIC(10,6),
  cost_delta             NUMERIC(10,6),
  overall_health_score   INTEGER,
  optimization_warranted BOOLEAN DEFAULT FALSE,
  confidence_level       TEXT,
  executive_summary      TEXT,
  full_report            JSONB,
  created_at             TIMESTAMPTZ DEFAULT NOW()
);
```

Step 4 — Create the prompt_version_registry Table

This table tracks every version of every agent's system prompt — the critical audit trail for your feedback loop:

```
CREATE TABLE prompt_version_registry (
  id               UUID DEFAULT gen_random_uuid() PRIMARY KEY,
  agent_name       TEXT NOT NULL,
  version          TEXT NOT NULL,
  system_prompt    TEXT NOT NULL,
  created_at       TIMESTAMPTZ DEFAULT NOW(),
  created_by       TEXT DEFAULT 'optimizer',
```

```
  change_log       TEXT,
  is_active        BOOLEAN DEFAULT TRUE,
  avg_score        NUMERIC(5,2),
  run_count        INTEGER DEFAULT 0,
  UNIQUE(agent_name, version)
);
```

4.2 Google Sheets Setup (Starter)

If you are starting with Google Sheets, create a spreadsheet with the following sheets and column headers:

Sheet Name	Column Headers (in order)
task_log	task_id | run_timestamp | agent_name | prompt_version | model_used | input_tokens | output_tokens | cost_usd | task_type | output_preview | metadata | outcome_score | outcome_data | outcome_collected_at
analysis_log	analysis_id | analysis_timestamp | agent_name | records_analyzed | avg_score | score_delta | health_score | optimization_warranted | executive_summary
prompt_registry	agent_name | version | created_at | change_log | avg_score | run_count | is_active
daily_health	date | agent_name | tasks_logged | tasks_scored | avg_score_7d | total_cost | anomaly_flags | status

4.3 Essential Queries for Monitoring

These SQL queries (for Supabase) give you the dashboard views you need to monitor your feedback loop health. Run them periodically or build them into a simple monitoring dashboard:

Query: Performance by Prompt Version

```
SELECT
  agent_name,
  prompt_version,
  COUNT(*) as run_count,
  ROUND(AVG(outcome_score), 2) as avg_score,
  ROUND(AVG(cost_usd), 6) as avg_cost,
  ROUND(AVG(outcome_score) / NULLIF(AVG(cost_usd), 0), 2) as score_per_dollar
FROM agent_task_log
WHERE outcome_score IS NOT NULL
  AND run_timestamp > NOW() - INTERVAL '30 days'
GROUP BY agent_name, prompt_version
ORDER BY agent_name, avg_score DESC;
```

Query: Unscored Records Needing Attention

```
SELECT agent_name, COUNT(*) as unscored_count,
  MIN(run_timestamp) as oldest_unscored
FROM agent_task_log
WHERE outcome_score IS NULL
  AND run_timestamp < NOW() - INTERVAL '48 hours'
GROUP BY agent_name
ORDER BY unscored_count DESC;
```

Query: 7-Day Rolling Performance

```
SELECT
  DATE_TRUNC('day', run_timestamp) as day,
  agent_name,
  COUNT(*) as runs,
  ROUND(AVG(outcome_score), 2) as avg_score,
  ROUND(SUM(cost_usd), 4) as total_cost
FROM agent_task_log
WHERE run_timestamp > NOW() - INTERVAL '14 days'
  AND outcome_score IS NOT NULL
GROUP BY 1, 2
ORDER BY 1 DESC, 2;
```

Chapter 5: The Action Logger — Capturing Every Signal

The Action Logger is the entry point for all data in your ML Ops system. Its job is deceptively simple: after every agent task, capture everything about that run and write it to the memory store. In practice, building a reliable, complete logger requires more care than people expect.

5.1 What to Capture and Why

Every field in the Task Run Record schema serves a specific analytical purpose. Understanding why each field matters helps you log the right data from day one:

Field	Why It Matters for ML Ops
prompt_version	Without this, you cannot attribute performance changes to specific prompt updates. Most important field.
model_used	Enables cost-quality analysis. Required to identify model routing improvements.
input_tokens + output_tokens	The raw cost drivers. Tracking over time reveals token bloat and optimization opportunities.
cost_usd	Normalize outcome scores against cost. An agent with 70 score at $0.001/run is better than 75 score at $0.005/run.
execution_time_ms	Latency matters for user-facing agents. Flag if execution time increases after prompt updates.
task_type	Enables per-task-type performance segmentation. Some task types improve faster than others.
output_preview	Lets the pattern analyzer evaluate output characteristics (length, structure, tone) as predictors.
metadata	Capture campaign_id, lead_industry, content_topic, etc. These become the features that explain performance variance.

5.2 Metadata Design — The Most Underused Lever

The metadata field is where most teams leave the most value on the table. Generic logs tell you that your outreach agent scored 45 on average. Rich metadata tells you that it scored 72 when the lead was in the SaaS industry and 28 when the lead was in manufacturing — a distinction that lets you build industry-specific prompt variants.

Design your metadata fields before you launch any agent. For each agent type, capture at minimum:

Agent Type	Essential Metadata Fields
Outreach agent	lead_industry, lead_company_size, lead_title, campaign_name, personalization_source
Content agent	content_type, platform, topic_category, target_audience, brand_voice_variant
Support agent	issue_category, customer_tier, channel, previous_contacts, escalation_history
Lead qualification	lead_source, industry, budget_range, timeline, pain_point_category
Data processing	data_source, record_count, transformation_type, schema_version

5.3 The Logger Prompt — OpenClaw Configuration

The Action Logger operates as a post-task webhook action in OpenClaw, not a conversational agent. Configure it as follows:

Action Logger — System Prompt

You are a precision data logging agent for an ML Ops system.
Your sole function is to capture agent task run data and write it to the memory store.
You do not evaluate, summarize, or comment on the task. You only log.

When triggered, you will receive these variables from the completed task:
{{agent_name}}, {{prompt_version}}, {{model_used}}, {{input_tokens}},
{{output_tokens}}, {{cost_usd}}, {{execution_time_ms}}, {{task_type}},
{{full_input}}, {{full_output}}, {{metadata_json}}

Construct a Task Run Record JSON object following this exact schema:
- task_id: generate a UUID v4
- run_timestamp: current UTC time in ISO 8601 format
- agent_name: {{agent_name}}
- prompt_version: {{prompt_version}}
- model_used: {{model_used}}
- input_tokens: {{input_tokens}} as integer
- output_tokens: {{output_tokens}} as integer
- cost_usd: {{cost_usd}} as float with 6 decimal places
- execution_time_ms: {{execution_time_ms}} as integer
- task_type: {{task_type}}
- input_preview: first 200 characters of {{full_input}}
- output_preview: first 300 characters of {{full_output}}

- output_word_count: word count of {{full_output}}
- metadata: parse {{metadata_json}} as object
- outcome_score: null
- outcome_data: null
- outcome_collected_at: null
- outcome_source: null

Use the supabase_insert tool to write this record to the agent_task_log table.
If the insert fails, retry once. If it fails again, use the error_log tool.
Output only: {status: 'logged', task_id: '[UUID]'} on success.
Output only: {status: 'error', message: '[error]'} on failure.
No other output under any circumstances.

5.4 Common Logger Failures and Fixes

Failure Mode	Symptom	Fix
Missing prompt_version	All records show same version; optimizer cannot compare	Hardcode version string in agent config; update it every time you modify the prompt
Incomplete metadata	Pattern analyzer cannot segment by context	Audit metadata fields before launch; use a metadata template per agent type
Token count mismatch	Reported cost doesn't match actual billing	Use OpenClaw's native token tracking variables — do not calculate manually
Output truncation too short	Analyzer cannot evaluate output quality	Set output_preview to 500 chars minimum for text-heavy agents
Logger fires on errors	Error runs pollute the dataset	Add a success_flag field; filter to success_flag=true in all analysis queries
No deduplication	Retried tasks create duplicate records	Use task_id as UNIQUE constraint; retry logic should use same task_id

Chapter 6: The Outcome Collector — Closing the Loop

The Outcome Collector is the component that transforms raw task logs into training data. Without it, you have a record of what the agent did — but no idea whether it worked. The Outcome Collector goes out to every downstream system where results manifest (your CRM, social platforms, email tool, helpdesk) and writes outcome scores back to the corresponding log records.

6.1 The Universal Scoring Rubric

Regardless of agent type, all outcomes are scored on a 0-100 scale. This normalization is critical — it lets the Pattern Analyzer compare performance across agent types, and it lets you track overall system health with a single number.

Score Range	Meaning	Typical Examples
0-10	Complete failure / negative outcome	Spam complaint, error, bounce, unsubscribe, support escalation
11-25	Significant underperformance	No response, low quality output, task incomplete, very low engagement
26-40	Below expectation	Weak response, partial completion, below-average engagement
41-60	Met basic expectation	Task completed, neutral outcome, average engagement, no conversion
61-75	Above expectation	Positive response, good quality, above-average engagement, soft conversion
76-90	Strong performance	Meeting booked, content went viral, ticket resolved first contact
91-100	Exceptional outcome	Closed deal from outreach, viral content, perfect CSAT, zero-touch resolution

6.2 Scoring Formulas by Agent Type

Beyond the general rubric, each agent type needs a specific scoring formula that maps raw outcome data to the 0-100 scale. Define these formulas before you launch the Outcome Collector. Changing scoring formulas mid-loop corrupts your historical comparison data.

Outreach Agent Scoring Formula

```
Base score calculation:
  No reply within 5 days:  score = 10
  Reply received:          score = 50
  Positive reply:          score = 65
  Meeting booked:          score = 85
  Deal opened in CRM:      score = 95

Modifiers (applied to base):
  Reply within 24 hours:   +10
  Unsubscribe request:     -30 (minimum score: 0)
  Spam complaint:          score = 0 (override)
  Forward to colleague:    +15
```

Content Agent Scoring Formula

```
Engagement Rate = (likes + comments + shares + saves) / reach

score = CASE
  WHEN engagement_rate < 0.01  THEN 20
  WHEN engagement_rate < 0.02  THEN 35
  WHEN engagement_rate < 0.03  THEN 50
  WHEN engagement_rate < 0.05  THEN 65
  WHEN engagement_rate < 0.08  THEN 78
  WHEN engagement_rate < 0.12  THEN 88
  ELSE 95
END

Modifiers:
  Reach > 10x account average:    +10 (capped at 100)
  Negative comment ratio > 10%:   -20
  Content removed by platform:    score = 0
```

Support Agent Scoring Formula

```
Base score:
  Ticket resolved first contact, CSAT > 4:     95
  Ticket resolved first contact, no CSAT:      75
  Ticket resolved, required 2nd contact:       55
  Ticket resolved, required escalation:        35
  Ticket unresolved after 24 hours:            15
  Customer cancelled/churned post-interaction: 0

Modifiers:
  Resolution time < 1 hour:      +5
  CSAT = 5 (max):                +5
  Follow-up complaint:           -20
```

6.3 Outcome Collector Prompt — Full System Prompt

Outcome Collector — System Prompt

```
You are an outcome tracking and scoring agent for an autonomous ML Ops system.
Your job runs once daily. You identify unscored task log records, retrieve outcome
data from downstream systems, apply scoring formulas, and update the log records.

STEP 1 — RETRIEVE UNSCORED RECORDS
Query the agent_task_log table for records where:
  outcome_score IS NULL
  AND run_timestamp < NOW() - INTERVAL '2 hours'  (give outcomes time to appear)
  AND run_timestamp > NOW() - INTERVAL '7 days'   (don't chase old records forever)
Use the supabase_query tool with this filter. Process in batches of 50.

STEP 2 — ROUTE BY AGENT TYPE
For each unscored record, route to the appropriate outcome source based on task_type:
  - 'outreach':       query CRM (HubSpot/Close) for reply/booking events by metadata.lead_id
  - 'content':        query social API for engagement metrics by metadata.post_id
  - 'support':        query helpdesk for ticket resolution by metadata.ticket_id
  - 'qualification':  query CRM for deal stage changes by metadata.deal_id
  - 'data_processing': check metadata.completion_webhook_received flag
If the outcome source does not yet have data for a record, skip it.

STEP 3 — APPLY SCORING FORMULA
Apply the scoring formula for the relevant agent type (formulas defined in your
scoring_config tool or passed as context). Produce:
  - outcome_score: integer 0-100
  - outcome_data: JSON object with the raw metrics used to calculate the score
  - outcome_source: string identifying which system provided the data
  - outcome_collected_at: current UTC timestamp

STEP 4 — UPDATE LOG RECORDS
Use supabase_update to write outcome fields back to each record by task_id.
Update in batches — do not update one record at a time.

STEP 5 — PRODUCE SUMMARY REPORT
Output a JSON summary:
{
  run_timestamp: ISO timestamp,
  records_processed: N,
  records_scored: N,
  records_skipped_no_data: N,
  records_errored: N,
  avg_score_today: float,
```

```
  errors: [{ task_id, error_message }]
}

Send this summary to the Loop Orchestrator via the orchestrator_callback tool.
```

6.4 Handling Outcome Data Sources

Source Type	Integration Method	What to Extract
HubSpot CRM	HubSpot API via OpenClaw HTTP tool	Contact activity: email opens, replies, meeting booked, deal stage
Close CRM	Close.io API	Lead activity: call logged, email replied, opportunity created
Instagram/TikTok	Platform API or third-party analytics tool	Likes, comments, shares, saves, reach, profile visits
Intercom / Zendesk	Helpdesk API	Ticket status, resolution time, CSAT score, follow-up tickets
LinkedIn Sales Nav	PhantomBuster or similar	InMail replies, connection accepts, profile views post-message
Email (Gmail/SMTP)	Gmail API or webhook on reply received	Open rate (if tracked), reply received Y/N, bounce/complaint
Internal systems	Direct webhook from your app	Any business event — purchase, upgrade, churn, NPS score

PART III

INTELLIGENCE LAYER

Chapter 7: The Pattern Analyzer — Finding What Works

The Pattern Analyzer is the intelligence engine of the entire system. It reads your accumulated run data, extracts meaningful patterns, and produces a structured analysis report that drives every optimization decision. Getting this right is the difference between a feedback loop that compounds and one that introduces noise.

7.1 What the Pattern Analyzer Is Actually Looking For

The goal is not to describe your data — it is to identify actionable interventions. Every pattern the analyzer surfaces should point to a specific change you can make to the agent configuration. Patterns without actionable recommendations are useless observations.

The analyzer searches for four categories of patterns:

- **Configuration Patterns:** Which prompt versions, model selections, or parameter settings correlate with high vs. low outcome scores? These directly drive the Prompt Optimizer.
- **Context Patterns:** Which metadata values correlate with performance? Which lead types, content topics, or customer segments does the agent handle better or worse? These drive segmentation strategies.
- **Failure Patterns:** What combination of inputs, contexts, and configurations predicts low scores? These drive negative example injection and specific failure handling instructions.
- **Efficiency Patterns:** Where are you spending tokens that aren't producing proportionally better outcomes? These drive cost optimization and model routing changes.

7.2 Minimum Data Requirements

The Pattern Analyzer should not run until these conditions are met. Running it on insufficient data produces misleading conclusions:

Condition	Minimum Threshold	Why
Total scored records per agent	50 minimum, 100+ preferred	Statistical significance — smaller samples are dominated by random variation

Prompt versions represented	At least 2 versions	Comparison requires variance — one version tells you nothing relative
Date range covered	At least 7 days	Day-of-week effects distort patterns from shorter windows
Scoring coverage	>70% of runs scored	High proportion of null scores biases averages toward scoreable runs
Metadata completeness	>80% of records have metadata	Incomplete metadata prevents context-based segmentation

7.3 The Pattern Analyzer Prompt — Full System Prompt

Pattern Analyzer — Complete System Prompt

```
You are a performance intelligence analyst for an autonomous AI agent ML Ops system.
You will be given a JSON dataset of agent task runs. Your job is to extract
actionable performance patterns that will drive concrete improvements.

ANALYSIS PROTOCOL — execute in this exact order:

STEP 1 — DATA VALIDATION
Before analyzing, verify:
  a) Total scored records >= 50. If not, output {error: 'insufficient_data',
     records_available: N} and stop.
  b) At least 2 distinct prompt_version values present. If not, note this
     in the report but continue with single-version analysis.
  c) Scoring coverage >= 70%. If lower, flag in the report.
  d) Date range spans at least 7 days. Note if shorter.

STEP 2 — BASELINE PERFORMANCE
Calculate for the analysis period:
  - avg_outcome_score (all runs), avg_outcome_score (by prompt_version)
  - avg_cost_usd per run, avg_cost_usd by model_used
  - distribution of scores: count in each decile (0-10, 11-20, ... 91-100)
  - failure rate: % of runs scoring below 30
  - top_score rate: % of runs scoring above 75

STEP 3 — PROMPT VERSION COMPARISON
For each prompt_version with >= 20 scored runs:
  - avg_score, std_dev, p25, p50, p75
  - failure rate, top_score rate
  - rank versions by avg_score
  - identify the performance delta between best and worst version
```

```
  - if delta > 10 points, flag as high priority optimization

STEP 4 — CONTEXT PATTERN MINING
For each metadata field present in >= 50% of records:
  - Group records by metadata field value
  - Calculate avg_score per group
  - Identify groups with scores > 15 points above or below overall avg
  - These are your segmentation opportunities

STEP 5 — FAILURE PATTERN ANALYSIS
Isolate records with outcome_score < 30 (failures).
Identify the top 3 conditions that appear in > 30% of failures but
< 30% of successes (score > 60). For each condition, calculate:
  - failure_prevalence: % of failures with this condition
  - success_prevalence: % of successes with this condition
  - odds_ratio: failure_prevalence / success_prevalence
Flag conditions with odds_ratio > 2.0 as strong failure predictors.

STEP 6 — COST EFFICIENCY ANALYSIS
Calculate for each model_used:
  - avg_score, avg_cost, score_per_dollar
Compare score_per_dollar across models.
Identify task_types where the more expensive model does NOT produce
meaningfully better outcomes (< 5 point score difference).
These are model downgrade opportunities.

STEP 7 — RECOMMENDATIONS
For each pattern identified in steps 3-6, produce a recommendation:
  {
    category: prompt_version | context_segment | failure_pattern | cost_efficiency,
    pattern: concise description of what you found,
    evidence: specific numbers from your analysis,
    recommendation: the exact change to make (be specific, not general),
    expected_impact: estimated score improvement or cost reduction,
    priority: 'high' | 'medium' | 'low',
    effort: 'low' | 'medium' | 'high'
  }

STEP 8 — FINAL REPORT
Output the complete Analysis Report JSON following Schema B exactly.
Set optimization_warranted = true if any high-priority recommendation exists.
Set confidence_level = 'high' if >= 100 records, 'medium' if 50-99, 'low' if < 50.
Write executive_summary in plain English. Start with the most important finding.
```

7.4 Interpreting Analysis Reports

The Pattern Analyzer output is only valuable if you know how to interpret it correctly. Common misinterpretations and how to avoid them:

Misinterpretation	Reality	Correct Interpretation
Version A scores higher, therefore Version A prompt is better	The runs in Version A may have had easier contexts (better leads, better timing)	Always check whether metadata distribution is similar across versions before concluding a prompt is better
Metadata field X correlates with high scores	X may be correlated with another variable that is the actual driver	Use correlation as a hypothesis, not a conclusion. Test by isolating the variable.
Failure rate dropped this week	Volume may have dropped — fewer hard runs, not better performance	Normalize failure rate by total run count and check if volume changed
Score_per_dollar is higher for Haiku	Quality difference may matter in your use case even if score gap looks small	Consider the business impact of the score delta, not just the dollar amount

Chapter 8: The Prompt Optimizer — Autonomous Improvement

The Prompt Optimizer is the most powerful and the most dangerous component in the system. It takes analysis findings and rewrites your agent system prompts. A well-implemented optimizer drives continuous improvement. A poorly implemented one introduces degradation. The guardrails in this chapter are not optional.

8.1 What the Optimizer Changes (and What It Never Touches)

The Optimizer Can Change	The Optimizer Must Never Change
Reasoning instructions (how the agent thinks)	Core task definition (what the agent does)
Output format requirements	Tool usage instructions
Edge case handling instructions	Safety and compliance requirements
In-context examples (few-shot)	Data schema field names or types
Tone and voice guidance	Authentication or API configuration
Error handling instructions	Personally identifiable information handling rules
Clarifying language for ambiguous situations	Hardcoded business rules approved by humans

The Golden Rule of Prompt Optimization

The optimizer is allowed to change HOW the agent thinks and communicates.
It is never allowed to change WHAT the agent is supposed to do.
Every optimizer run should be a refinement, not a reinvention.

8.2 The Approval Workflow

Every optimized prompt must go through an approval workflow before deployment. The level of human oversight required depends on the magnitude of the changes:

Change Type	Approval Required	Auto-Deploy?
Word-level wording refinement (< 5% of prompt changed)	Async review — Slack notification, 4h to reject	Yes, after 4h if no rejection
Example addition or replacement	Async review — Slack notification, 24h to reject	Yes, after 24h if no rejection

New section or instruction added	Synchronous review — human must approve before deploy	Never
Core reasoning approach changed	Synchronous review + test run before deploy	Never
System prompt length change > 20%	Synchronous review + full regression test	Never

8.3 The Prompt Optimizer — Full System Prompt

Prompt Optimizer — Complete System Prompt

You are an expert prompt engineer. You receive a performance analysis report and the current system prompt for an OpenClaw agent. Your job is to produce an improved version of that system prompt based on the analysis findings.

INPUTS:
  current_prompt: the full text of the agent's current system prompt
  analysis_report: the JSON analysis report from the Pattern Analyzer
  success_metric: the primary metric being optimized
  change_constraints: any fields or sections marked as immutable

PROTOCOL — execute in order:

STEP 1 — CONSTRAINT CHECK
Read the change_constraints list. Identify all sections of the current prompt that are immutable. Mark them mentally. You will not touch these.

STEP 2 — MAP RECOMMENDATIONS TO PROMPT SECTIONS
For each high-priority recommendation in the analysis_report:
  a) Identify which section of the current prompt is responsible for the pattern
  b) Determine what type of change would address it:
    - Clarification: add specificity to vague instructions
    - Example: add a positive or negative few-shot example
    - Constraint: add an explicit instruction to avoid a failure pattern
    - Reordering: move instructions to change processing priority
  c) Draft the change

STEP 3 — DRAFT THE REVISED PROMPT
Write the complete revised system prompt. Rules:
  - Do not increase total length by more than 20%
  - Every change must trace back to a specific analysis finding
  - Annotate each changed section with: # OPTIMIZATION [v.NEW]: [reason in 10 words]

```
  - Do not remove functional instructions — only add or refine
  - Preserve all immutable sections exactly as they appear in the original

STEP 4 — SELF-REVIEW
Before outputting, verify:
  a) All immutable sections are unchanged
  b) Total length increase is within 20% limit
  c) Every change has a traceable reason
  d) The revised prompt is internally consistent (no contradictions)
  e) The core task definition is unchanged

STEP 5 — OUTPUT
Return a JSON object:
{
  agent_name: string,
  previous_version: string,
  new_version: string (increment minor for wording changes, major for structural),
  revised_prompt: string (complete prompt text with annotations),
  clean_prompt: string (same prompt WITHOUT annotation comments),
  change_log: [
    {
      change_type: 'clarification'|'example'|'constraint'|'reorder'|'removal',
      section_changed: string,
      previous_text: string (original wording),
      new_text: string (new wording),
      analysis_finding: string (which finding drove this change),
      expected_metric_impact: string
    }
  ],
  immutable_sections_preserved: boolean,
  length_delta_pct: float,
  approval_tier: 'async-4h'|'async-24h'|'sync-required'|'sync-plus-test',
  confidence: 'low'|'medium'|'high',
  human_review_notes: string (specific things reviewer should check)
}
```

8.4 Versioning Protocol

Every prompt change must increment the version number and be recorded in the prompt_version_registry table. This is mandatory. Without a complete version history, the feedback loop cannot attribute performance changes to specific modifications, and the entire system loses its analytical integrity.

Use semantic versioning adapted for prompts:

- v1.0 → v1.1: Minor wording refinements, example additions (auto-deploy eligible)
- v1.1 → v1.2: Edge case handling additions, format changes (async-24h approval)
- v1.x → v2.0: Structural changes to reasoning approach (sync-required approval)
- v2.0 → v3.0: Core task reframing or significant expansion (sync + test required)

Chapter 9: The Loop Orchestrator — Keeping It All Running

The Loop Orchestrator is the operations manager of your ML Ops system. It does not analyze, optimize, or execute tasks — it makes sure every other component runs on schedule, detects when something is broken, and escalates before small problems become data-corrupting failures.

9.1 Orchestrator Responsibilities by Cadence

Cadence	Actions
Every 4 hours (business hours)	Health check: verify Action Logger fired in last 4h; alert if silent
Daily 6:00 AM	Trigger Outcome Collector; run anomaly detection; generate daily health report
Daily 9:00 AM	Review daily health report; send summary to notification channel
Weekly Sunday 11 PM	Trigger Pattern Analyzer; evaluate if optimization is warranted; trigger Optimizer if yes
Weekly Monday 9 AM	Deploy approved prompt updates; update version registry; notify team
Monthly 1st of month	Generate monthly performance report; archive data; review scoring rubric validity

9.2 Anomaly Detection Rules

The Orchestrator must detect and alert on these anomaly conditions. Every alert should include: the anomaly type, the affected agent, the current metric value, the expected range, and a suggested first diagnostic step.

Anomaly	Detection Condition	Severity	Auto-Action
Logger silence	No new records in agent_task_log for >4h during business hours	Critical	Alert immediately, pause optimizer
Score collapse	7-day rolling avg drops >20 points vs prior 7-day avg	High	Alert, pause optimizer, flag for human review
Cost spike	Daily cost_usd >50% above 7-day average	High	Alert, check for routing misconfiguration

Scoring drought	>20% of records older than 72h with null outcome_score	Medium	Alert, check Outcome Collector logs
Version drift	Agent running an unregistered prompt_version	Medium	Alert, verify version registry is current
Memory store growth stop	No new records for >24h when agents are active	Critical	Alert, check Supabase connection
Optimizer loop	Same agent optimized more than 2x in 7 days	High	Pause optimizer, escalate to human

9.3 The Orchestrator Prompt — Full System Prompt

Loop Orchestrator — Complete System Prompt

You are the Loop Orchestrator for an OpenClaw ML Ops system. You manage scheduling, health monitoring, anomaly detection, and escalation for the five-agent feedback loop. You do not analyze, optimize, or execute tasks.

AVAILABLE TOOLS:
- supabase_query: read from memory store
- trigger_agent: fire any sub-agent by name
- send_notification: Slack/email alerts
- update_agent_config: deploy approved prompt updates
- log_orchestrator_event: write orchestration events to audit log

DAILY ROUTINE (triggered at 6:00 AM):
1. Run health checks (see HEALTH CHECK PROTOCOL below)
2. Trigger Outcome Collector with current timestamp
3. Wait 15 minutes for Outcome Collector to complete
4. Query for Outcome Collector summary report
5. Generate daily_health record (see schema below)
6. If any CRITICAL or HIGH anomalies detected, send immediate alert
7. Send daily summary at 9:00 AM

WEEKLY ROUTINE (triggered Sunday 11:00 PM):
1. Run health checks
2. Trigger Pattern Analyzer with date_range = last 14 days
3. Wait 10 minutes for Pattern Analyzer to complete
4. Retrieve analysis report from pattern_analysis_log
5. If analysis_report.optimization_warranted = true AND

```
    analysis_report.confidence_level IN ('medium','high'):
    - Trigger Prompt Optimizer with analysis report + current prompt
    - Route optimizer output to appropriate approval workflow
    - Log the optimization trigger event
6. If optimization_warranted = false: log 'no optimization warranted', continue
7. Generate weekly_summary report and send to notification channel

HEALTH CHECK PROTOCOL:
Run after every trigger. Check all conditions in the anomaly detection table.
For each anomaly detected:
  a) Log the anomaly with full context to audit log
  b) Send notification with: anomaly_type, agent_name, current_value,
     expected_range, suggested_first_diagnostic_step
  c) If severity = Critical: pause optimizer for all agents
  d) If severity = High: pause optimizer for affected agent only
  e) Set system_status = 'degraded' in health record

DEPLOYMENT PROTOCOL (triggered Monday 9:00 AM):
1. Query for optimizer outputs with status = 'approved'
2. For each approved update:
   a) Retrieve clean_prompt from optimizer output
   b) Use update_agent_config to deploy new prompt
   c) Insert new record in prompt_version_registry
   d) Set previous version is_active = false
   e) Log deployment event with full change_log
   f) Send deployment notification

OUTPUT: After every routine, output a JSON status:
{ routine: string, completed_at: ISO timestamp, actions_taken: [],
  anomalies_detected: [], system_status: 'healthy'|'degraded'|'critical' }
```

PART IV

ADVANCED MACHINE LEARNING TECHNIQUES

Chapter 10: Multi-Agent Reinforcement Patterns

Single-agent feedback loops optimize one agent in isolation. But in most real deployments, multiple agents work in sequence — an outreach agent generates a reply, a qualification agent scores that reply, a booking agent books the meeting. The performance of Agent B depends on the output quality of Agent A. Multi-agent reinforcement addresses this dependency.

10.1 The Upstream/Downstream Dependency Problem

When Agent A produces poor output, Agent B receives poor input. If you optimize Agent B without knowing that Agent A is the root cause, you will chase a ghost — making changes to B that will never solve a problem that originates in A. This is one of the most common failure modes in multi-agent deployments.

> **Detection Rule**
> If Agent B's outcome scores are low AND Agent A's output quality is low, optimize Agent A first. Never optimize a downstream agent until you have verified that upstream output quality is above 60 average score.

10.2 Pipeline Score Attribution

For a three-agent pipeline (A → B → C), you need to know which agent is responsible for the final outcome. Implement pipeline score attribution by tracking the task_id chain:

```
// Extend the Task Run Record schema with pipeline fields:
{
  ...standard_fields,
  pipeline_id:           UUID (shared across all agents in one pipeline run),
  parent_task_id:        UUID | null (task_id of the upstream agent that
produced input),
  pipeline_position:     integer (1, 2, 3... position in the pipeline),
  pipeline_final_score:  integer | null (set by Outcome Collector on final
agent only)
}

// Attribution query: which agent has highest correlation with final outcome?
SELECT
  agent_name,
```

```
  pipeline_position,
  CORR(outcome_score, final.pipeline_final_score) as attribution_score
FROM agent_task_log atl
JOIN (
  SELECT pipeline_id, pipeline_final_score
  FROM agent_task_log
  WHERE pipeline_final_score IS NOT NULL
) final ON atl.pipeline_id = final.pipeline_id
GROUP BY agent_name, pipeline_position;
```

10.3 Cross-Agent Context Passing

As each agent in a pipeline improves, it produces better outputs that benefit all downstream agents. To amplify this compounding effect, implement context passing — where the quality score of upstream outputs is included in the context of downstream agents:

Cross-Agent Context Injection Prompt Addition

Add this section to any downstream agent's system prompt:

UPSTREAM CONTEXT:
The input you are receiving was produced by {{upstream_agent_name}} running prompt version {{upstream_prompt_version}}. This upstream agent currently achieves an average outcome score of {{upstream_avg_score}}/100.
Known strengths of this upstream agent: {{upstream_top_wins}}
Known weaknesses to compensate for: {{upstream_known_failures}}

Use this context to calibrate your handling — if the upstream agent is known to under-specify certain fields, compensate in your processing.

10.4 Ensemble Patterns

For high-stakes tasks where a single agent run is too risky, implement an ensemble pattern: run 2-3 agent variants in parallel, score them using a judge agent, and select the best output. The feedback loop then tracks which variant wins most often and optimizes toward that variant's configuration.

Ensemble Judge Agent — System Prompt

You are an output quality judge. You will receive 2-3 versions of a completed agent task output. Your job is to select the best one based on the success criteria provided, and explain your selection.

```
EVALUATION CRITERIA (provided at runtime):
  {{success_criteria_list}}

FOR EACH OUTPUT, score it 1-10 on:
  - Accuracy: does it correctly complete the requested task?
  - Relevance: is every element of the output relevant to the goal?
  - Quality: would this output achieve the success metric?
  - Efficiency: is it appropriately concise without sacrificing quality?

Output JSON:
{
  winner: integer (1, 2, or 3 — which variant won),
  winner_score: float (average of 4 criteria),
  scores: [{variant: 1, accuracy: N, relevance: N, quality: N, efficiency: N}],
  winner_rationale: string (2 sentences why this one won),
  improvement_note: string (what would make even the winner better)
}
```

Chapter 11: A/B Testing Automation at Scale

Manual A/B testing is slow, inconsistent, and impossible to scale. Automated A/B testing lets you run controlled experiments on every prompt change, collect statistically valid results, and promote winners or roll back losers — all without human involvement in the test itself.

11.1 How Automated A/B Testing Works in OpenClaw

When the Prompt Optimizer produces a new prompt version, instead of immediately deploying it to all traffic, the Orchestrator routes a defined percentage of runs to the new version while keeping the rest on the current version. Both versions run simultaneously, the Outcome Collector scores both, and the Pattern Analyzer declares a winner when statistical significance is reached.

Phase	Action	Who Performs It
1. Deploy Test	Route 20% of traffic to new version, 80% to current	Orchestrator
2. Collect Data	Log all runs with version tag; score outcomes normally	Logger + Outcome Collector
3. Monitor	Daily check: is sample size reached? Any anomalies?	Orchestrator
4. Declare Winner	When N >= 30 per variant and confidence >= 90%, pick winner	Pattern Analyzer
5. Full Rollout	Route 100% to winner; archive loser with performance data	Orchestrator
6. Document	Write A/B result to experiment log with full context	Orchestrator

11.2 A/B Test Configuration in the Task Run Record

Extend the metadata field to support A/B test tracking without changing the core schema:

```
// In metadata field of Task Run Record, add:
{
  ...other_metadata,
  ab_test_id:      string | null,  // 'exp-outreach-2024-03-01' or null if no
active test
  ab_variant:      'control' | 'treatment' | null,
```

```
  ab_traffic_pct:  integer | null   // what % of traffic this variant is
receiving
}

// A/B winner declaration query:
SELECT
  metadata->>'ab_variant' as variant,
  COUNT(*) as n,
  ROUND(AVG(outcome_score), 2) as avg_score,
  ROUND(STDDEV(outcome_score), 2) as std_dev
FROM agent_task_log
WHERE metadata->>'ab_test_id' = 'exp-outreach-2024-03-01'
  AND outcome_score IS NOT NULL
GROUP BY 1;
```

Chapter 12: Negative Example Injection

Few-shot examples in prompts typically show the model what good outputs look like. Negative example injection adds the inverse: carefully curated examples of bad outputs with explanations of why they failed. This is one of the highest-leverage optimization techniques in the system, and the most underused.

12.1 Why Negative Examples Work

Language models are trained to predict what comes next based on patterns. When you show a model an example of a poor output and explain why it is poor, you activate pattern-avoidance — the model weights its generation away from those patterns. This is especially powerful for failure modes that are subtle and recurring, where positive-only examples do not provide sufficient contrast.

12.2 How to Build Your Negative Example Library

Your feedback loop automatically generates a negative example library — every run scoring below 25 is a candidate negative example. The process for curating and injecting these examples:

12. Query your memory store weekly for runs scoring < 25
13. For each low-scoring run, retrieve the output_preview and the outcome_data
14. Use an analysis agent to annotate why the output failed (see prompt below)
15. Store the annotated failure in a negative_examples table
16. Include the top 3 most relevant negative examples in your agent's system prompt

Failure Annotator Agent — System Prompt

```
You are a failure analysis agent. You will receive an agent output that performed
poorly (outcome score < 25) and the outcome data explaining why it failed.
Your job is to produce a concise failure annotation suitable for injection
into the agent's system prompt as a negative few-shot example.

Input:
  agent_output: {{output_text}}
  outcome_score: {{score}}
  outcome_data: {{outcome_data}}
  task_context: {{task_type}} for {{metadata_summary}}

Produce:
{
  failure_summary: string (one sentence: what the output did wrong),
  failure_category: string (one of: too_long | too_short | off_topic |
    wrong_tone | missing_cta | factual_error | format_violation |
    irrelevant_personalization | generic | other),
  negative_example_text: string (formatted for prompt injection):
    Format: 'BAD EXAMPLE — DO NOT do this: [output snippet]
             WHY IT FAILED: [failure_summary]
             WHAT TO DO INSTEAD: [specific alternative approach]',
  reusability_score: integer 1-10 (how broadly applicable is this failure lesson),
  inject_in_prompt_section: string (which section of the prompt to add this near)
}
```

Chapter 13: Cost-Quality Frontier Optimization

Every agent configuration sits somewhere on the cost-quality curve. Moving up the curve increases quality and cost. Moving down reduces cost but may sacrifice quality. The Cost-Quality Frontier is the set of configurations where you cannot reduce cost without sacrificing quality, and cannot improve quality without increasing cost. Your goal is to stay on this frontier — not above it (wasting money on quality you don't need) and not below it (sacrificing quality without saving enough to justify it).

13.1 Mapping Your Current Position

Use this query to plot your cost-quality position for each agent and task type:

```
-- Cost-quality position by agent and model
```

```
SELECT
  agent_name,
  model_used,
  task_type,
  COUNT(*) as runs,
  ROUND(AVG(outcome_score), 2) as quality_score,
  ROUND(AVG(cost_usd) * 1000, 4) as avg_cost_per_1k_runs,
  ROUND(AVG(outcome_score) / NULLIF(AVG(cost_usd), 0), 0) as quality_per_dollar
FROM agent_task_log
WHERE outcome_score IS NOT NULL
  AND run_timestamp > NOW() - INTERVAL '30 days'
GROUP BY 1, 2, 3
HAVING COUNT(*) >= 20
ORDER BY agent_name, quality_per_dollar DESC;
```

13.2 Model Downgrade Decision Framework

Not all tasks need Sonnet. Use this decision tree for every task type to determine the optimal model:

Question	Yes → Action	No → Action
Does the task require multi-step reasoning or synthesis?	Use Sonnet	Proceed to next question
Is the output format complex (structured JSON, long-form prose)?	Use Sonnet for first draft; Haiku for revision	Proceed to next question
Does Haiku produce outputs that score within 8 points of Sonnet for this task?	Use Haiku — acceptable quality delta	Use Sonnet — quality gap is too large
Is this task in a user-facing, high-stakes context?	Use Sonnet regardless of cost	Haiku is acceptable if score delta < 10
Are you running > 1000 of this task type per month?	A/B test Haiku vs Sonnet immediately — savings are significant	Continue with current model

Chapter 14: Cross-Agent Knowledge Transfer

Your feedback loop generates knowledge — specific, validated, hard-won insights about what works in your market, your niche, and your use case. Cross-agent knowledge transfer is the practice of deliberately sharing that knowledge between agents instead of letting each agent learn everything from scratch.

14.1 The Knowledge Transfer Use Cases

Source Agent	Knowledge to Transfer	Target Agent
Outreach agent	Which industries respond best to which openers	Lead qualification agent — weight qualifying questions by industry
Content agent	Which hook styles drive highest engagement	Outreach agent — use proven hooks in email subject lines
Support agent	Most common resolution strategies by issue type	Product agent — surface known workarounds proactively
Qualification agent	Which signals predict high-value customers	Outreach agent — prioritize leads with high-value signals
Content agent	Best performing CTAs by content category	Email agent — use same CTA patterns in nurture sequences

14.2 The Knowledge Transfer Prompt

Knowledge Transfer Agent — System Prompt

You are a knowledge synthesis agent. You will receive two analysis reports:
one from a source agent and one from a target agent. Your job is to identify
insights from the source agent that could be applied to improve the target agent.

PROCESS:
1. Read source_analysis_report and extract all top_wins with priority='high'
2. For each high-priority win, evaluate:
   a) Is this insight specific to the source agent's task, or is it generalizable?
   b) If generalizable, how would it apply to the target agent's context?
   c) What specific change to the target agent's prompt would apply this insight?
3. Produce transfer recommendations for insights that pass the generalizability test

Output JSON:
```
{
  source_agent: string,
  target_agent: string,
  transfer_recommendations: [
    {
      source_insight: string,
      target_application: string,
      target_prompt_change: string (exact text to add/change),
```

```
      confidence: 'low'|'medium'|'high',
      rationale: string
    }
  ]
}
```

PART V

REAL-WORLD DEPLOYMENTS

Chapter 15: Use Case — Cold Outreach Engine

Cold outreach is the highest-ROI application of OpenClaw ML Ops. The performance ceiling for a non-ML outreach agent is typically 3-8% reply rate. An ML Ops outreach engine, after 90 days of loop operation, consistently achieves 12-25% reply rates in most B2B markets. Here is the complete deployment blueprint.

15.1 System Configuration

Component	Configuration
Primary metric	Reply rate (replies / messages sent, rolling 7-day)
Secondary metrics	Meeting booked rate, positive reply rate, unsubscribe rate
Outcome lag	72 hours (give prospects time to reply before scoring)
Loop cadence	Weekly analysis, bi-weekly optimization
A/B test traffic split	80% control / 20% treatment during test phases
Model routing	Haiku for personalization variable extraction; Sonnet for message drafting
Minimum sample	100 sends per variant before declaring A/B winner

15.2 Essential Metadata Fields

- lead_industry — the industry of the recipient company
- lead_company_size — employee count bucket (1-10, 11-50, 51-200, 201-1000, 1000+)
- lead_title_seniority — IC, manager, director, VP, C-level
- personalization_source — manual, LinkedIn, website, news, referral
- message_template — which template variant was used (A, B, C...)
- send_day_of_week — Tuesday and Thursday outperform Monday/Friday in most markets
- send_time_bucket — morning (6-9am), midday (10am-1pm), afternoon (2-5pm)
- campaign_name — to segment by campaign strategy

15.3 What the Loop Will Learn

Over 90 days, your outreach loop will discover patterns specific to your market that no general-purpose playbook can provide. Common high-value discoveries:

- **Industry-specific opener effectiveness:** SaaS founders respond to ROI-framing; agency owners respond to time-saving framing; enterprise buyers respond to risk-reduction framing.
- **Title-level response patterns:** CEOs at small companies respond to peer-positioning; VPs at large companies respond to business case framing; ICs respond to workflow improvement framing.
- **Optimal message length:** Most markets have a sweet spot — typically 60-90 words for cold email. The loop will find your specific sweet spot through outcome correlation.
- **Personalization depth threshold:** There is a point at which additional personalization stops improving reply rate. The loop finds this threshold, preventing over-personalization that increases cost without improving outcomes.

15.4 Stage-by-Stage Evolution

Stage	Timeline	Loop State	Expected Reply Rate
Cold Start	Days 1-14	Manual logging, no optimization	Baseline: 3-6%
First Signal	Days 15-30	First pattern analysis run; initial optimization	4-8%
Compounding	Days 31-60	Regular weekly cycles; first segmentation variants	7-14%
Maturity	Days 61-90	Multi-variant routing; negative example injection active	12-20%
Optimization	Day 90+	Full ML Ops: A/B testing, cross-agent transfer, ensemble	18-28%

Chapter 16: Use Case — Content Intelligence System

Content generation at scale without a feedback loop produces content that performs consistently mediocre. With a feedback loop, your content agent discovers what resonates specifically with your audience and compounds that knowledge into every subsequent piece. This is how creators using AI go from average engagement to viral content.

16.1 System Configuration

Component	Configuration
Primary metric	Engagement rate (interactions / reach) — platform-specific formula
Secondary metrics	Saves rate (highest-intent signal), share rate, profile visits, follower growth
Outcome lag	168 hours (7 days — allow full engagement curve to play out)
Loop cadence	Bi-weekly analysis (need 2 weeks of engagement data per cycle)
Platforms	Track separately — TikTok, Instagram Reels, LinkedIn, Twitter/X all have different patterns
Model routing	Haiku for hook generation (high volume); Sonnet for long-form caption drafts
Content taxonomy	Hook style × Content type × CTA type — at minimum 3-dimensional metadata

16.2 Hook Style Taxonomy for Metadata Tracking

Hook style is the single highest-leverage variable in content performance. Track it precisely:

Hook Style Code	Example	Typical Performance Context
QUESTION	'Why do most founders fail at marketing?'	High engagement in educational niches
BOLD_CLAIM	'Cold email is dead. Here is what works now.'	Strong in contrarian / thought leadership niches
NUMBER	'7 things I learned building to $1M'	Consistently strong across most B2B audiences
STORY_START	'I almost quit last Tuesday. Here is what happened.'	Very strong on TikTok; weaker on LinkedIn
THREAT	'If you are still doing X, you are losing money'	High click, mixed engagement — monitor unsubscribes

CURIOSITY_GAP	'The one thing nobody tells you about OpenClaw...'	Platform-dependent; strong on TikTok, moderate elsewhere
RESULT_FIRST	'I went from $0 to $50K MRR. Here is the exact playbook.'	Strong for entrepreneur audiences

Chapter 17: Use Case — Lead Qualification Loop

Lead qualification is a high-stakes, long-feedback-cycle ML Ops application. The feedback loop has a 30-60 day minimum before it produces reliable signal because the outcome (does the lead close?) takes weeks to materialize. But the compounding ROI is exceptional — a qualification agent that learns to identify your highest-value leads can increase close rates by 30-50%.

17.1 The Multi-Stage Scoring Problem

Lead qualification has multiple intermediate outcomes before the final outcome (close). Design your scoring to capture the full funnel:

Funnel Stage	Observable Outcome	Score Contribution
Initial qualification	Lead accepts meeting / responds positively	30% of final score
Discovery call	Lead shows genuine pain; budget confirmed; timeline confirmed	40% of final score
Proposal	Proposal requested; decision maker engaged	20% of final score
Close	Deal won / lost / ghosted	10% of final score (the actual close is downstream)

This multi-stage scoring approach lets you get feedback within 7-14 days (meeting booked) rather than waiting 30-60 days for a close event — giving the feedback loop much faster signal without sacrificing accuracy.

Chapter 18: Use Case — Customer Support Optimizer

Customer support is the fastest feedback cycle use case in this guide. Outcomes are measurable within hours, not days — which means your support feedback loop can run on a

3x-weekly analysis cadence and produces visible improvements within the first week of operation.

18.1 System Configuration

Component	Configuration
Primary metric	First-contact resolution rate (FCR) — resolved without follow-up ticket
Secondary metrics	CSAT score, resolution time, escalation rate, re-open rate
Outcome lag	4-24 hours (check for follow-up tickets and CSAT response)
Loop cadence	3x per week analysis; weekly optimization
Issue taxonomy	Billing, Technical, Onboarding, Feature Request, Complaint, Refund — at minimum
Customer tier tracking	Free, Trial, Paid, Enterprise — routing rules differ by tier

18.2 Support ML Ops Loop Unique Advantage

Unlike outreach or content, support has an immediate feedback signal (was the ticket re-opened?) AND a downstream signal (CSAT score). This means you can run two separate optimization passes: optimize for FCR using the immediate signal, and optimize for satisfaction using the CSAT signal. These two objectives can occasionally conflict — the loop will surface this conflict and you can make an explicit choice about which to prioritize.

Chapter 19: Use Case — Token Cost Reduction System

Token cost optimization is the feedback loop that funds all other feedback loops. The cost savings it generates can be reinvested into higher quality models for tasks that truly require them. This is not a standalone use case — it is a system-wide optimization layer that runs across all your agents.

19.1 The Cost Reduction Loop Architecture

Unlike other feedback loops that optimize a single agent, the cost reduction loop operates at the system level. It analyzes all agents simultaneously and generates cost optimization recommendations across your entire OpenClaw deployment.

Optimization Type	Typical Savings	Implementation Complexity
Model downgrade (Sonnet → Haiku for eligible tasks)	40-70% cost reduction on downgraded tasks	Low — change model routing config
Prompt compression (remove redundant instructions)	10-25% token reduction	Medium — requires careful testing
Input compression (summarize lengthy inputs before processing)	20-40% on input-heavy tasks	Medium — add pre-processing step
Output length constraints (enforce word limits on outputs)	15-30% output token reduction	Low — add constraint to prompt
Caching (reuse outputs for identical or near-identical inputs)	50-80% on repetitive tasks	High — requires similarity detection
Batch processing (group small tasks into single large call)	10-20% on per-call overhead	Medium — requires queue architecture

Cost Reduction Target by Maturity Stage

Month 1 (Cold Start): Baseline established — no optimization yet
Month 2 (First Cycle): Target 15-25% cost reduction through model routing fixes
Month 3 (Compounding): Target 35-50% cost reduction through routing + compression
Month 4+ (Optimized): Target 60-75% cost reduction vs. Day 1 baseline

Actual results from SPRINT member deployments: 65-90% cost reduction at 6 months. This is the 97% token reduction pattern documented in ScaleUP Media case studies.

PART VI

REFERENCE

Chapter 20: Complete System Prompt Library

This chapter contains every system prompt in the OpenClaw ML Ops system in final, deployable form. Each prompt is labeled with its agent role, trigger condition, recommended model, and key variables it expects. Copy these directly into OpenClaw — they are written to work without modification for most use cases.

PROMPT 1 — Action Logger

Attribute	Value
Role	Post-task webhook — fires after every agent run
Model	Claude Haiku (low complexity, high volume)
Trigger	Post-task action on every tracked agent
Variables	{{agent_name}}, {{prompt_version}}, {{model_used}}, {{input_tokens}}, {{output_tokens}}, {{cost_usd}}, {{execution_time_ms}}, {{task_type}}, {{full_input}}, {{full_output}}, {{metadata_json}}

Action Logger — Final Deployable Prompt

You are a precision data logging agent. Log this completed task run to the memory store. Produce NO output except the logging status. Do not summarize, evaluate, or comment.

Construct a Task Run Record with these exact values:
task_id: [generate UUID v4]
run_timestamp: [current UTC in ISO 8601]
agent_name: {{agent_name}}
prompt_version: {{prompt_version}}
model_used: {{model_used}}
input_tokens: {{input_tokens}}
output_tokens: {{output_tokens}}
cost_usd: {{cost_usd}}
execution_time_ms: {{execution_time_ms}}
task_type: {{task_type}}
input_preview: [first 200 chars of {{full_input}}]

output_preview: [first 300 chars of {{full_output}}]
output_word_count: [word count of {{full_output}}]
metadata: [parse {{metadata_json}} as JSON object]
outcome_score: null
outcome_data: null
outcome_collected_at: null
outcome_source: null

Write this record to the agent_task_log table using the supabase_insert tool.
On success, output ONLY: {"status": "logged", "task_id": "[the UUID]"}
On failure, retry once. If retry fails: {"status": "error", "message": "[error]"}
No other output under any circumstances.

PROMPT 2 — Outcome Collector

Attribute	Value
Role	Daily outcome retrieval and scoring agent
Model	Claude Haiku (structured data retrieval, rule-based scoring)
Trigger	Daily cron — 6:00 AM UTC
Variables	No input variables — agent queries its own data sources

Outcome Collector — Final Deployable Prompt

You are an outcome tracking agent. Run daily at 6:00 AM.
Retrieve outcome data for unscored runs and write scores back to the log.

STEP 1 — Query unscored records:
Use supabase_query: SELECT * FROM agent_task_log
WHERE outcome_score IS NULL
  AND run_timestamp < NOW() - INTERVAL '2 hours'
  AND run_timestamp > NOW() - INTERVAL '7 days'
ORDER BY run_timestamp ASC LIMIT 100;

STEP 2 — For each record, route to outcome source by task_type:
  outreach → CRM: search by metadata.lead_id for reply/booking events
  content → Platform API: retrieve by metadata.post_id for engagement data
  support → Helpdesk API: retrieve by metadata.ticket_id for resolution/CSAT

qualification → CRM: retrieve by metadata.deal_id for deal stage
data_processing → Check metadata.completion_webhook_received
If no outcome data available yet for a record, skip it (set skipped=true).

STEP 3 — Score each retrieved outcome using the scoring formula for that task_type.
Store: outcome_score (0-100), outcome_data (raw metrics JSON),
outcome_source (system name), outcome_collected_at (now UTC).

STEP 4 — Batch update scored records via supabase_update.
Update by task_id. Batch size: 25 records per call.

STEP 5 — Output summary JSON only:
{records_processed, records_scored, records_skipped_no_data, records_errored,
avg_score_of_scored_batch, errors: [{task_id, error_message}]}
Send to orchestrator_callback tool with event_type: 'outcome_collection_complete'

PROMPT 3 — Pattern Analyzer

Attribute	Value
Role	Weekly performance intelligence and pattern extraction
Model	Claude Sonnet (requires nuanced reasoning and statistical interpretation)
Trigger	Weekly cron — Sunday 11:00 PM, triggered by Orchestrator
Variables	{{agent_name}}, {{date_range_start}}, {{date_range_end}}, {{task_data_json}}

Pattern Analyzer — Final Deployable Prompt

You are a performance intelligence analyst for an OpenClaw ML Ops system.
Analyze the provided task run dataset and produce a structured analysis report.

Agent: {{agent_name}} | Period: {{date_range_start}} to {{date_range_end}}
Dataset: {{task_data_json}}

EXECUTE IN ORDER:

1. VALIDATE: Confirm records >= 50, prompt versions >= 2, scoring coverage >= 70%.
If records < 50: return {error: 'insufficient_data', count: N} and stop.

2. BASELINE: Calculate avg_score, score_distribution (by decile), failure_rate (score < 30), top_score_rate (score > 75), avg_cost, by prompt_version.

3. VERSION COMPARISON: For each version with >= 20 records:
avg_score, std_dev, p25/p50/p75, failure_rate, top_score_rate.
Rank versions. Flag delta > 10 points as high priority.

4. CONTEXT PATTERNS: For each metadata field in >= 50% of records:
group by value, calc avg_score per group.
Flag groups scoring > 15 points from overall avg.

5. FAILURE ANALYSIS: Isolate score < 30 records.
Find top 3 conditions: appear in > 30% of failures AND < 30% of successes.
Calculate odds_ratio per condition. Flag odds_ratio > 2.0.

6. COST EFFICIENCY: Per model: avg_score, avg_cost, score_per_dollar.
Flag task_types where downgrade saves > 40% cost for < 5 score delta.

7. RECOMMENDATIONS: Per finding, produce:
{category, pattern, evidence (specific numbers), recommendation (exact change),
expected_impact, priority: high|medium|low, effort: low|medium|high}

8. OUTPUT: Complete Analysis Report JSON (Schema B).
Set optimization_warranted = true if any priority:high recommendation exists.
Write executive_summary starting with single most important finding.
Insert record into pattern_analysis_log via supabase_insert.

PROMPT 4 — Prompt Optimizer

Attribute	Value
Role	Automated prompt improvement based on analysis findings
Model	Claude Sonnet (requires careful reasoning and prompt engineering skill)
Trigger	Triggered by Orchestrator when Pattern Analyzer returns optimization_warranted=true
Variables	{{agent_name}}, {{current_prompt}}, {{analysis_report_json}}, {{success_metric}}, {{immutable_sections}}

Prompt Optimizer — Final Deployable Prompt

You are an expert prompt engineer. Improve the agent system prompt based on the analysis findings. Your output will be reviewed before deployment.

Agent: {{agent_name}}
Current prompt: {{current_prompt}}
Analysis report: {{analysis_report_json}}
Success metric: {{success_metric}}
Immutable sections (do NOT change): {{immutable_sections}}

PROTOCOL:
1. CONSTRAINT CHECK: Mark all immutable sections. These are untouchable.

2. MAP: For each priority:high recommendation in the analysis report,
identify the prompt section responsible and determine change type:
clarification | example_add | example_replace | constraint_add | reorder

3. DRAFT: Write the complete revised prompt. Rules:
- Max 20% length increase
- Annotate every change: # OPT [NEW_VERSION]: [reason, 10 words max]
- Do not remove functional instructions — refine only
- Do not change immutable sections

4. SELF-REVIEW: Verify immutability, length limit, consistency, traceability.

5. DETERMINE APPROVAL TIER:
- < 5% wording change only: async-4h
- Example additions: async-24h
- New instruction sections: sync-required
- Core reasoning changes or > 20% length delta: sync-plus-test

6. OUTPUT JSON:
{agent_name, previous_version, new_version, revised_prompt (annotated),
clean_prompt (no annotations), change_log [{change_type, section_changed,
previous_text, new_text, analysis_finding, expected_metric_impact}],
immutable_sections_preserved (boolean), length_delta_pct (float),
approval_tier, confidence: low|medium|high, human_review_notes}

PROMPT 5 — Loop Orchestrator

Attribute	Value
Role	Master scheduler, health monitor, and deployment controller
Model	Claude Haiku (scheduling and routing logic — Sonnet not needed)
Trigger	Two cron schedules: Daily 6AM + Sunday 11PM
Variables	{{routine_type}}: 'daily' or 'weekly' — determines which routine to run

Loop Orchestrator — Final Deployable Prompt

You are the Loop Orchestrator for an OpenClaw ML Ops system.
Manage scheduling, health monitoring, and deployment for the feedback loop.
You do not analyze data or optimize prompts — those are other agents' jobs.

Current routine: {{routine_type}}

IF routine_type = 'daily':
1. Run HEALTH CHECKS (see below)
2. Trigger outcome_collector_agent
3. Wait 15 min, retrieve its summary via supabase_query (latest record)
4. Write daily_health record to supabase
5. If CRITICAL/HIGH anomalies: send_notification immediately
6. Schedule daily summary notification for 9:00 AM

IF routine_type = 'weekly':
1. Run HEALTH CHECKS
2. For each active agent: trigger pattern_analyzer_agent
3. Wait 10 min, retrieve analysis reports
4. For each report where optimization_warranted = true
   AND confidence IN ('medium','high'):
   - Trigger prompt_optimizer_agent with that agent's report + current prompt
   - Route optimizer output to approval workflow based on approval_tier
5. Send weekly summary with all analysis highlights

HEALTH CHECKS — run these queries and flag anomalies:
Logger silence: no records in last 4h during 6AM-8PM window → CRITICAL
Score collapse: 7-day avg dropped > 20 pts vs prior 7 days → HIGH
Cost spike: daily cost > 50% above 7-day avg → HIGH
Scoring drought: > 20% records > 72h old with null score → MEDIUM
Version drift: records with unregistered prompt_version → MEDIUM
Optimizer loop: same agent optimized 2+ times in 7 days → HIGH → PAUSE OPTIMIZER

DEPLOYMENT (runs Monday 9AM as separate 'deployment' routine_type):

```
1. Query optimizer outputs with status='approved'
2. For each: deploy via update_agent_config, update prompt_version_registry,
   set old version is_active=false, log deployment, send notification

OUTPUT after every run:
{routine, completed_at, actions_taken: [], anomalies: [],
 system_status: 'healthy'|'degraded'|'critical'}
```

PROMPT 6 — Failure Annotator

Failure Annotator — Final Deployable Prompt

```
You are a failure analysis agent. You receive a low-scoring agent output
and explain why it failed in a format suitable for negative example injection.

Input:
  agent_name: {{agent_name}}
  output_text: {{output_text}}
  outcome_score: {{score}} (confirmed below 25)
  outcome_data: {{outcome_data}}
  task_context: {{task_type}} | {{metadata_summary}}

ANALYZE:
1. What specifically about this output caused the poor outcome?
   Consider: length, tone, structure, accuracy, relevance, missing elements,
   excessive elements, wrong framing, mismatched context.
2. Categorize the failure: too_long | too_short | off_topic | wrong_tone |
   missing_cta | factual_error | format_violation | irrelevant_personalization |
   generic | over_engineered | other
3. What is the minimum change that would have prevented this failure?
4. How broadly applicable is this lesson (1-10)?
   10 = applies to nearly all runs; 1 = extremely context-specific

OUTPUT JSON:
{
  failure_summary: string (one sentence, starts with what the output DID),
  failure_category: string,
  root_cause: string (what in the prompt or context caused this pattern),
  reusability_score: integer 1-10,
```

```
  negative_example_block: string (ready to paste into system prompt):
    format: 'BAD EXAMPLE — avoid this pattern:
            [verbatim output snippet, max 80 words]
            WHY THIS FAILED: [failure_summary]
            WHAT TO DO INSTEAD: [specific actionable alternative]',
  prompt_section_to_inject_near: string
}
```

PROMPT 7 — Knowledge Transfer Agent

Knowledge Transfer Agent — Final Deployable Prompt

```
You are a cross-agent knowledge synthesis specialist. You identify insights
from one agent's performance data that can be applied to improve another agent.

Source agent analysis: {{source_analysis_json}}
Target agent name: {{target_agent_name}}
Target agent current prompt: {{target_prompt}}
Target agent success metric: {{target_success_metric}}

PROCESS:
1. Extract all priority:high and priority:medium wins from source_analysis
2. For each win, test for generalizability:
   - Is this insight specific to the source agent's task type? (skip if yes)
   - Could a similar pattern exist in the target agent's domain?
   - Would applying this insight risk changing the target agent's core function?
3. For each generalizable insight, draft the target prompt change:
   - Identify the target prompt section to modify
   - Write the specific text addition or change
   - Estimate confidence that transfer will improve target metric

OUTPUT JSON:
{
  source_agent: string,
  target_agent: string,
  transfer_recommendations: [
    {source_insight, target_application, target_prompt_section,
     proposed_change_text, confidence: low|medium|high, rationale}
  ],
```

```
  total_recommendations: integer,
  high_confidence_count: integer
}
If no generalizable insights found: {transfer_recommendations: [], reason: string}
```

PROMPT 8 — A/B Test Judge

A/B Test Judge — Final Deployable Prompt

```
You are an A/B test result evaluator for an OpenClaw ML Ops system.
Determine whether a prompt experiment has produced a conclusive winner.

Test ID: {{ab_test_id}}
Agent: {{agent_name}}
Control version: {{control_version}} | Treatment version: {{treatment_version}}
Test data: {{ab_test_data_json}}

EVALUATION PROTOCOL:
1. Verify minimum sample: control >= 30 scored runs AND treatment >= 30 scored runs.
   If not: return {status: 'insufficient_data', control_n: N, treatment_n: N}

2. Calculate for each variant:
   avg_score, std_dev, failure_rate (score<30), top_score_rate (score>75),
   avg_cost, score_per_dollar

3. Calculate statistical significance:
   score_delta = treatment_avg - control_avg
   If abs(score_delta) < 5: inconclusive (noise range)
   If abs(score_delta) >= 5 AND n >= 50 per variant: sufficient evidence
   If abs(score_delta) >= 10: strong evidence, declare winner

4. Determine recommendation:
   - score_delta > 5 (treatment wins): promote treatment to 100% traffic
   - score_delta < -5 (control wins): roll back treatment, archive with learnings
   - abs(score_delta) < 5: extend test for another cycle

5. OUTPUT JSON:
{status: 'conclusive'|'inconclusive'|'insufficient_data',
 winner: 'control'|'treatment'|null, score_delta: float,
```

```
confidence: 'high'|'medium'|'low', recommendation: string,
key_finding: string (one sentence — what this test revealed),
archive_note: string (what to record for future reference)}
```

Chapter 21: First-Time Setup Wizard

Use this prompt to bootstrap your first ML Ops deployment. Run it in a Claude Sonnet chat with your agent context filled in. It will output a complete, customized configuration package — database schema, scoring formula, metadata fields, and initial system prompts — ready to deploy.

First-Time Setup Wizard Prompt (Run in Claude Chat)

I am setting up an OpenClaw ML Ops feedback loop for my first time.
Help me build a complete, customized configuration for my specific agent.

MY AGENT DETAILS:
Agent name: [YOUR AGENT NAME]
What it does: [ONE SENTENCE — be specific]
Volume: approximately [N] runs per [day/week]
Current prompt version: v1.0
Model currently used: [claude-haiku / claude-sonnet]
Primary success metric: [the ONE number that defines success]
How I measure that metric: [WHERE does that number come from?]
Outcome lag: [how long after the agent runs before I know if it worked?]
Downstream system: [CRM name / social platform / helpdesk / etc.]
Key context variables: [what factors vary between runs — industry, campaign, etc.]

Based on this, please produce ALL of the following:

1. SUPABASE SCHEMA: The exact SQL CREATE TABLE statements I need, customized for my agent type with the right metadata fields as JSONB keys.

2. SCORING FORMULA: A precise, numeric scoring formula for my specific metric, including base scores and modifiers for my outcome type.

3. METADATA FIELD LIST: The exact 6-10 metadata fields I should capture for my agent type, with the data type and why each one matters for ML Ops.

4. OUTCOME COLLECTOR CONFIG: The exact query logic and data extraction steps for my specific downstream system ({{downstream_system}}).

5. LOOP CADENCE RECOMMENDATION: Recommended schedule for Outcome Collector, Pattern Analyzer, and Prompt Optimizer given my volume and outcome lag.

6. COLD START PLAN: A specific 2-week plan for manual logging and scoring that will get me to 50 scored records as fast as possible.

7. FIRST OPTIMIZATION HYPOTHESIS: Based on the agent type and metric I described, what are the top 3 things most likely to drive performance improvement in my first optimization cycle? Give me specific prompt changes to test.

Output everything as implementation-ready configurations.
No general advice — I want exact specifications I can copy and deploy.

Chapter 22: Troubleshooting and Common Failures

Every ML Ops deployment runs into problems. The table below documents the most common failures, their root causes, and the exact fixes. Review this chapter before escalating any issue — 90% of problems are on this list.

22.1 Data Pipeline Failures

Symptom	Root Cause	Fix
No new records in agent_task_log	Action Logger webhook misconfigured or Supabase connection down	Check webhook URL and API key; verify Supabase table exists and RLS policy allows inserts
Records missing fields (NULLs where values expected)	OpenClaw variable mapping incorrect	Audit variable names in Logger prompt against available OpenClaw task variables
Duplicate records for same run	Logger firing multiple times (retry logic creating duplicates)	Add UNIQUE constraint on task_id; use upsert instead of insert in Logger
outcome_score never populates	Outcome Collector not reaching downstream system OR source data not available	Test Outcome Collector manually; verify API credentials for downstream tool
Very low scoring coverage (< 40%)	Outcome lag too short — collecting before outcomes exist	Extend outcome window from 2h to 24h or 48h depending on your outcome lag
All records show same prompt_version	Version string not being updated when prompt changes	Add version update to your deployment checklist; consider auto-incrementing in Logger

22.2 Analysis and Optimization Failures

Symptom	Root Cause	Fix
Pattern Analyzer produces 'no patterns found'	Insufficient data OR all versions performing similarly	Wait for more data; or introduce deliberate variation by testing a significantly different prompt
Optimizer makes harmful changes (scores drop post-optimization)	Optimizer applied a pattern that was spurious (noise, not signal)	Increase minimum sample to 100; add 14-day rolling window requirement; manually revert to previous version
Same recommendations repeat every cycle	Root cause is in data collection, not the prompt	Audit metadata completeness; check if scoring formula is correct; verify

		Outcome Collector is querying the right source
Optimizer tries to change immutable sections	Immutable sections not clearly marked in the prompt	Add explicit [IMMUTABLE] tags to protected sections; add constraint to Optimizer prompt
Analysis reports show inconsistent results week to week	Small sample size creating high variance	Require minimum 14-day analysis window; use 30-day window for slow-feedback agents
Confidence always 'low' in analysis reports	Records count is below 50 threshold consistently	Either increase agent volume or extend analysis window to accumulate more records per cycle

22.3 Orchestration Failures

Symptom	Root Cause	Fix
Orchestrator health alerts not firing	Notification webhook misconfigured	Test webhook manually; verify Slack/email integration in OpenClaw
Weekly cycle not triggering	Cron schedule misconfigured or Orchestrator failing silently	Add Orchestrator execution log; verify cron syntax; add a watchdog alert if weekly trigger hasn't fired
Optimizer triggered but no approval notification sent	Approval workflow misconfigured	Add explicit notification step after Optimizer runs; log approval_tier in Orchestrator output
Multiple optimizations in same week (loop)	Missing optimizer loop detection	Add check: if same agent optimized in last 7 days, skip and alert instead

Chapter 23: 90-Day Maturity Roadmap

This roadmap gives you a concrete week-by-week implementation plan. Follow it sequentially. Resist the temptation to skip phases — each phase builds the foundation for the next.

Phase 1: Foundation (Weeks 1-2)

Phase 1 Goals: Build the data infrastructure and start collecting clean data

Week 1:

- Day 1-2: Answer the 4 mindset questions from Chapter 1 for your target agent
- Day 2-3: Set up Supabase with all three tables (Chapter 4 SQL)
- Day 3-4: Configure the Action Logger on your target agent (Chapter 5)
- Day 4-5: Test logger by running 10 agent tasks — verify all records appear
- Day 5-7: Design and document your scoring formula (Chapter 6 formulas)

Week 2:

- Day 8-10: Configure Outcome Collector against your downstream system
- Day 10-12: Run Outcome Collector manually — verify scores populate correctly
- Day 12-14: Manual review: read every output from first 30 runs
  - Score 20 of them manually and compare to Outcome Collector scores
  - Calibrate the scoring formula if automated scores don't match intuition

Phase 1 Exit Criteria: 30+ records logged, 20+ records scored, scoring accuracy calibrated

Phase 2: First Loop Cycle (Weeks 3-5)

Phase 2 Goals: Run your first automated loop cycle and produce first optimization

Week 3:

- Continue logging and collecting. Aim for 50 scored records by end of week.
- Deploy the Loop Orchestrator on daily schedule.
- Monitor daily health reports — verify no anomalies.

Week 4:

- Run Pattern Analyzer for the first time (manually trigger if < 50 records).
- Review the analysis report carefully. Do the findings make intuitive sense?
- If yes: proceed to Prompt Optimizer.
- If no: debug data quality before optimizing.

Week 5:
Run Prompt Optimizer on first analysis report.
Review proposed changes manually — this is the most important human review step.
Deploy new prompt version (v1.0 → v1.1 or v1.0 → v2.0).
Note the deployment date — this is your first optimization event.

Phase 2 Exit Criteria: First optimization deployed, version v1.x in production, Orchestrator running on automated schedule

Phase 3: Compounding (Weeks 6-10)

Phase 3 Goals: Establish automated weekly cadence; launch first A/B test

Weeks 6-7:
Loop running on full automated schedule.
Review weekly analysis reports — validate findings make sense.
Compare performance of v1.1 vs v1.0 — measure the impact of first optimization.
Begin building negative example library (Chapter 12).

Weeks 8-9:
Launch first A/B test (Chapter 11).
Deploy v2.0 as treatment variant at 20% traffic.
Monitor daily — watch for anomalies in either variant.

Week 10:
A/B test should have 30+ scored runs per variant by now.
Run A/B Test Judge agent — declare winner.
Roll out winner to 100% traffic. Archive loser with learnings.

Phase 3 Exit Criteria: 200+ total scored records, first A/B test completed, measurable performance improvement vs. Week 1 baseline

Phase 4: Optimization (Weeks 11-13)

Phase 4 Goals: Full autonomous operation; introduce advanced techniques

Week 11:
Negative example injection active in system prompt.
Context pattern segmentation: build variant prompts for top-performing segments.

Week 12:
Model routing optimization: run cost-quality analysis.
Implement any model downgrade opportunities identified.
Verify score delta is within acceptable range after downgrade.

Week 13:
If you have 2+ agents with feedback loops running: implement cross-agent knowledge transfer (Chapter 14).
Generate first monthly performance report.
Compare Week 13 baseline metrics to Week 1 baseline metrics.
Document the performance delta — this is your compounding return.

Phase 4 Exit Criteria: Full autonomous loop running, cost reduction documented, performance improvement vs. Day 1 clearly quantified

Expected Performance Benchmarks

Metric	Week 1 Baseline	Week 13 Target	Typical Range Achieved
Outcome score (avg)	Baseline established	+15 to +30 points	Depends heavily on starting baseline
Failure rate (score < 30)	Baseline established	50-70% reduction	Most deployments hit 50%+ reduction by week 13
Token cost per run	Baseline established	40-65% reduction	Routing optimization alone typically 40-50%
Human tuning time	High (manual reviews daily)	Low (anomaly review only)	80-90% reduction in hands-on optimization time
Prompt version cadence	1 version (v1.0)	4-6 versions	Each version typically 5-15 points above predecessor
A/B tests completed	0	2-3 tests	Each test adds 1 data point to your knowledge base

The Compounding Advantage — Final Word

After 90 days of operation, your OpenClaw ML Ops system will have:

- A proprietary dataset of hundreds or thousands of real-world outcomes from YOUR market
- A system prompt refined through multiple optimization cycles on REAL performance data

- A negative example library that prevents your specific failure modes
- A cost structure 40-70% lower than Day 1 with equal or better performance
- A competitive moat that deepens every week the loop runs

No competitor starting today can replicate this without running the same number of cycles. Time is the input that cannot be shortcut. Start the loop. Let it run. Trust the data.

ScaleUP Media · THE SPRINT Program · @mattganzak

Build. Measure. Learn. Compound.