

Hacker Bob Offline Code Walkthrough

How the project, agent pipeline, MCP memory, context management, and JSON state fit together

Generated for local code review

2026-05-01

Contents

1	How To Use This Guide	7
2	One-Page Mental Model	8
3	Why This Architecture Exists	9
3.1	Problem 1: Chat Context Is Too Small	9
3.2	Problem 2: Many Agents Need One Shared Memory	9
3.3	Problem 3: Agents Drift Unless Outputs Are Structured	9
3.4	Problem 4: Security Automation Needs Guardrails	10
4	Source Checkout Versus Installed Workspace	11
4.1	Source Repo	11
4.2	Installed Project	11
5	The Three Control Surfaces	12
5.1	1. Slash Commands And Skills	12
5.2	2. Agent Definitions	12
5.3	3. MCP Runtime	13
6	The Full Runtime Flow	14
7	The Finite-State Machine	16
7.1	HUNT -> CHAIN	16
7.2	CHAIN -> VERIFY	16
7.3	VERIFY -> GRADE	17
7.4	GRADE -> REPORT	17
8	Session State	18
9	JSON, JSONL, And Markdown	20
9.1	JSON	20
9.2	JSONL	20
9.3	Markdown	20
10	Core Artifact Table	21
11	MCP Server Architecture	22
11.1	Transport	22
11.2	Registry	22
11.3	Dispatch	23
11.4	Envelopes	23

12 The 39 MCP Tools By Role	24
12.1 Orchestrator Tools	24
12.2 Hunter Tools	24
12.3 Auth Tools	24
12.4 Chain Tools	24
12.5 Verification Tools	25
12.6 Evidence, Grade, Report Tools	25
13 Context Management In Detail	26
13.1 Context Is Not Shared By Default	26
13.2 The Root Orchestrator Gets Summaries	26
13.3 Hunters Get One Curated Brief	26
13.4 Verification Reads Structured Artifacts	26
13.5 Evidence Is Separate From Findings	27
13.6 Analytics Summarizes The Whole Pipeline	27
14 Agent Architecture	28
14.1 Root Orchestrator	28
14.2 Recon Agent	28
14.3 Hunter Agent	28
14.4 Chain Builder	29
14.5 Three Verifiers	29
14.6 Evidence Agent	29
14.7 Grader	29
14.8 Report Writer	30
15 Wave Lifecycle	31
15.1 Start A Wave	31
15.2 Spawn Background Hunters	31
15.3 Hunter Writes Handoff	31
15.4 SubagentStop Hook Validates Completion	32
15.5 Merge A Wave	32
16 How Findings Flow Through The Pipeline	34
16.1 1. Hunter Records A Finding	34
16.2 2. Chain Builder Tests Combinations	34
16.3 3. Verification Rounds Decide Survivors	35
16.4 4. Evidence Agent Collects Bounded Proof	35
16.5 5. Grader Produces A Verdict	35
17 HTTP, Auth, Egress, And Audit	37
17.1 HTTP Scan	37
17.2 Auth Storage	37
17.3 Egress Profiles	38
17.4 Geofence Warnings	38
18 Hooks And Guardrails	39
18.1 Session Write Guard	39
18.2 Hunter Stop Hook	39
18.3 Scope Guard Hooks	40
19 Pipeline Analytics And Debugging	41
19.1 Pipeline Events	41
19.2 Tool Telemetry	41
19.3 Pipeline Analytics Reader	42
20 How To Read The Repo Step By Step	43
20.1 Step 1: Read The Product Contract	43

20.2	Step 2: Understand Install And Lifecycle	43
20.3	Step 3: Read The Orchestrator Prompt	43
20.4	Step 4: Read The Agent Prompts	44
20.5	Step 5: Read MCP From Entry Point To Tool Handler	44
20.6	Step 6: Read Session State And Storage	45
20.7	Step 7: Read Waves And Hunter Context	45
20.8	Step 8: Read Findings, Chain, Verification, Evidence, Grade	46
20.9	Step 9: Read HTTP, Auth, Egress, Redaction	46
20.10	Step 10: Read Hooks And Tests	46
21	Useful Offline Commands	47
21.1	List Files	47
21.2	Find Tool Definitions	47
21.3	See Tool Registry Summary	47
21.4	Run Focused Tests	47
21.5	Run Full Test Suite	47
21.6	Check Generated Prompt Surfaces	47
21.7	Inspect Installed-Style MCP Server	47
21.8	Search For Artifact Ownership Rules	47
22	Suggested Flight Reading Plan	48
22.1	30 Minutes: Product Shape	48
22.2	45 Minutes: Orchestrator And Agents	48
22.3	60 Minutes: MCP Core	48
22.4	90 Minutes: State, Waves, Context	48
22.5	60 Minutes: Findings To Report	48
22.6	45 Minutes: Safety And Diagnostics	48
22.7	45 Minutes: Tests As Documentation	49
23	Annotated Golden Run	50
23.1	Step 0: Claude Code Loads The Control Surface	50
23.2	Step 1: Session Initialization	50
23.3	Step 2: Recon Agent Produces The Attack Surface	51
23.4	Step 3: Auth Either Stores Profiles Or Marks The Run Unauthenticated	52
23.5	Step 4: The Root Starts A Hunt Wave	52
23.6	Step 5: Hunters Run In Parallel	53
23.7	Step 6: Each Hunter Loads One Curated Context Packet	53
23.8	Step 7: Hunter Activity Creates Structured Evidence Trails	54
23.9	Step 8: Hunter Completion Uses Two Signals	54
23.10	Step 9: Resume Or Completion Merges The Wave	55
23.11	Step 10: The Root Decides Another Wave Or CHAIN	56
23.12	Step 11: Chain Builder Tests Combinations	56
23.13	Step 12: Three Verifiers Protect Against Agent Overclaiming	57
23.14	Step 13: Evidence Packs Are Written After Final Verification	57
23.15	Step 14: Grader Produces A Submit/Hold/Skip Decision	58
23.16	Step 15: Report Writer Produces The Human Output	58
23.17	Golden Run Reading Checklist	59
24	Deep Artifact Catalog	60
24.1	state.json	60
24.2	attack_surface.json	60
24.3	Recon Scratch Files	61
24.4	auth.json	61
24.5	traffic.jsonl	62
24.6	http-audit.jsonl	62
24.7	wave-N-assignments.json	63
24.8	handoff-wN-aN.json	63
24.9	handoff-wN-aN.md	64

24.10	live-dead-ends-wN-aN.jsonl	64
24.11	coverage.jsonl	64
24.12	findings.jsonl	65
24.13	findings.md	65
24.14	chain-attempts.jsonl	65
24.15	brutalist.json, balanced.json, verified-final.json	66
24.16	brutalist.md, balanced.md, verified-final.md	66
24.17	evidence-packs.json	66
24.18	evidence-packs.md	67
24.19	grade.json	67
24.20	grade.md	67
24.21	report.md	67
24.22	pipeline-events.jsonl	68
24.23	Tool Telemetry	68
24.24	Static Artifact Files	68
24.25	public-intel.json	69
25	Agent Responsibility Matrix	70
25.1	Root Orchestrator	70
25.2	Recon Agent	70
25.3	Hunter Agent	71
25.4	Chain Builder	72
25.5	Brutalist Verifier	72
25.6	Balanced Verifier	73
25.7	Final Verifier	73
25.8	Evidence Agent	74
25.9	Grader	74
25.10	Report Writer	75
26	State Machine And Phase Gates	76
26.1	Transition Contract	76
26.2	Gate Table	76
26.3	Phase Gate Debugging Question	77
26.4	State Machine Reading Exercise	77
27	Trace One Finding End To End	79
27.1	Forward Trace	79
27.1.1	1. Hunter Discovers The Issue	79
27.1.2	2. Hunter Logs Coverage	79
27.1.3	3. Hunter Records The Finding	80
27.1.4	4. Hunter Writes Handoff	80
27.1.5	5. Wave Merge Updates State	80
27.1.6	6. Chain Builder Tests The Chain Note	81
27.1.7	7. Verification Rounds Debate The Finding	81
27.1.8	8. Evidence Agent Creates A Safe Proof Pack	82
27.1.9	9. Grader Scores It	82
27.1.10	10. Report Writer Uses Only Survivors	83
27.2	Backward Trace	83
28	Debugging Playbook	84
28.1	The Run Says A Wave Is Pending	84
28.2	HUNT Will Not Transition To CHAIN	84
28.3	CHAIN Will Not Transition To VERIFY	85
28.4	VERIFY Will Not Transition To GRADE	85
28.5	The Report Mentions A Finding That Is Not In Final Verification	85
28.6	A Hunter Keeps Repeating Work	86
28.7	A Finding Is Missing From The Report	86
28.8	Auth Does Not Seem To Work	87

28.9 HTTP Requests Fail Repeatedly	87
28.10JSON And Markdown Disagree	87
29 Context Budget Cheat Sheet	89
29.1 Pattern 1: Compact State For The Root	89
29.2 Pattern 2: One Assigned Brief For Each Hunter	89
29.3 Pattern 3: Structured Artifacts For Later Agents	89
29.4 What Gets Summarized	90
29.5 What Must Stay Structured	90
29.6 Context Loss Recovery	90
30 Offline Exercises	92
30.1 Exercise 1: Find The Main Orchestrator Contract	92
30.2 Exercise 2: Find Who Writes <code>state.json</code>	92
30.3 Exercise 3: Find The Wave Token Logic	92
30.4 Exercise 4: Find The Hunter’s First Required Action	92
30.5 Exercise 5: Find Where HUNT -> CHAIN Can Block	93
30.6 Exercise 6: Find The “Include Every Finding” Rule	93
30.7 Exercise 7: Find Evidence Validation	93
30.8 Exercise 8: Find The Tool Registry Contract	93
30.9 Exercise 9: Find Write Guard Rules	94
30.10Exercise 10: Trace One Tool End To End	94
31 Code Map By Question	95
31.1 Where Does A Run Start?	95
31.2 Where Are Phases Defined?	95
31.3 Where Are MCP Tools Registered?	95
31.4 Where Is The Standard MCP Response Shape?	95
31.5 Where Are Session Paths Defined?	95
31.6 Where Is State Persisted And Normalized?	95
31.7 Where Are Waves Implemented?	95
31.8 Where Are Hunter Briefs Built?	95
31.9 Where Are Findings Implemented?	96
31.10Where Are Chain Attempts Implemented?	96
31.11Where Are Verification Rounds Implemented?	96
31.12Where Are Evidence Packs Implemented?	96
31.13Where Is Grading Implemented?	96
31.14Where Is HTTP Scanning And Audit Implemented?	96
31.15Where Is Auth Stored And Applied?	96
31.16Where Are Egress Profiles Implemented?	96
31.17Where Are Analytics Implemented?	96
31.18Where Are Safety Hooks?	97
31.19Where Are Prompt Contracts Tested?	97
31.20Where Are End-To-End Runtime Behaviors Tested?	97
32 Improvement Roadmap At A Glance	98
32.1 Quick Wins	98
32.2 Medium Refactors	98
32.3 Larger Architecture Improvements	98
32.4 Design Principles To Preserve	99
33 Improvement Suggestions	100
33.1 1. Add A Machine-Readable Artifact Schema Reference	100
33.2 2. Generate An Artifact Lifecycle Diagram From Tests Or Metadata	100
33.3 3. Add A Local “Explain This Session” Static Report	101
33.4 4. Make Context Budgets Visible In Hunter Briefs	101
33.5 5. Add A “Why This Phase Is Blocked” MCP Tool	102
33.6 6. Add Small Fixture Sessions For Offline Learning	102

33.7	7. Consider A Human-Friendly Session Index	103
33.8	8. Tighten Prompt-To-Tool Traceability	104
33.9	9. Add A Developer “Trace One Tool” Walkthrough	104
33.10	10. Make Report Trust Explicit In <code>report.md</code>	104
33.11	11. Add More Negative Fixtures For Agent Drift	105
33.12	12. Separate “Runtime Docs” From “Contributor Docs”	105
33.13	13. Add A Lightweight Local TUI Or HTML Dashboard	106
33.14	14. Keep Strong Boundaries Around Automatic Egress Rotation	106
33.15	15. Make “Context Is A Product Feature” A First-Class Doc Theme	107
34	FAQ	108
34.1	Where is the main entry point?	108
34.2	Why use MCP at all?	108
34.3	Why not just let agents write files directly?	108
34.4	Why JSONL for findings instead of one JSON file?	108
34.5	Why are Markdown files still present?	108
34.6	What happens if the root chat loses context?	108
34.7	What is <code>pending_wave</code> ?	108
34.8	Why forbid same-turn merge after spawning hunters?	108
34.9	What is a handoff token?	108
34.10	Why does the hunter also emit <code>BOB_HUNTER_DONE</code> ?	109
34.11	Who owns <code>attack_surface.json</code> ?	109
34.12	How does Bob avoid repeating work?	109
34.13	How does Bob decide which surfaces to hunt?	109
34.14	What is the difference between coverage and findings?	109
34.15	Why have three verification rounds?	109
34.16	Why does evidence come after final verification?	109
34.17	What makes evidence valid?	109
34.18	What determines SUBMIT, HOLD, or SKIP?	109
34.19	Can a no-finding session still reach REPORT?	110
34.20	Where is auth stored?	110
34.21	Does Bob block private IPs and localhost?	110
34.22	What is an egress profile?	110
34.23	Why is the scope guard a no-op?	110
34.24	What is <code>/bob-status</code> ?	110
34.25	What is <code>/bob-debug</code> ?	110
34.26	What is policy replay?	110
34.27	How are generated prompt surfaces kept in sync?	110
34.28	Which tests should I read first?	110
34.29	What should I trust if Markdown and JSON disagree?	111
35	Glossary	112
35.1	Agent	112
35.2	Orchestrator	112
35.3	MCP	112
35.4	Surface	112
35.5	Wave	112
35.6	Handoff	112
35.7	Coverage	112
35.8	Finding	112
35.9	Chain Attempt	112
35.10	Verification Round	112
35.11	Evidence Pack	112
35.12	Grade	112
35.13	Pipeline Analytics	112
36	File Map Cheat Sheet	113

36.1 User-Facing Docs	113
36.2 Claude-Facing Runtime	113
36.3 MCP Core	113
36.4 MCP State And Pipeline	113
36.5 HTTP, Auth, Recon Enrichment	113
36.6 Installer And Lifecycle	114
36.7 Tests	114

37 Final Reading Advice	115
--------------------------------	------------

1 How To Use This Guide

This guide is meant for a flight or any offline reading session. It assumes you have the repository checkout open locally and no internet connection.

Read it in two passes:

1. First pass: understand the system shape. Focus on the diagrams, the phase flow, and the artifact tables.
2. Second pass: keep the repo open and follow the file-by-file reading plan. When a section references a file, open it and skim the exact functions named there.

All paths in this guide are relative to the repository root:

`/Users/memehalis/sec/hacker-bob`

The most important idea is this:

`Claude chat coordinates.`

`Agents do bounded jobs.`

`The MCP server owns memory and validation.`

`JSON/JSONL artifacts are the durable source of truth.`

`Markdown is mostly for humans.`

2 One-Page Mental Model

Hacker Bob is not one monolithic script. It is a local Claude Code framework installed into a Claude Code project.

The source repo contains:

<code>.claude/</code>	Claude-facing commands, skills, agents, hooks, rules, knowledge
<code>mcp/</code>	Local MCP server and all runtime tool implementations
<code>scripts/</code>	Installer, config merge, release, prompt generation utilities
<code>bin/</code>	CLI entry point: <code>hacker-bob</code>
<code>test/</code>	Contract tests for MCP, prompts, installer, CLI, hooks
<code>testing/</code>	Policy replay harness for prompt/refusal diagnostics
<code>docs/</code>	Human docs and release notes

At runtime, a Bob hunt creates a local session directory:

```
~/bounty-agent-sessions/[target_domain]/
```

That session directory stores structured artifacts:

```
state.json
attack_surface.json
wave-1-assignments.json
handoff-w1-a1.json
coverage.jsonl
findings.jsonl
chain-attempts.jsonl
brutalist.json
balanced.json
verified-final.json
evidence-packs.json
grade.json
report.md
pipeline-events.jsonl
http-audit.jsonl
```

The slash command `/bob-hunt` is the orchestrator. It does not directly hunt. It coordinates the finite-state machine:

```
RECON -> AUTH -> HUNT -> CHAIN -> VERIFY -> GRADE -> REPORT
```

The heavy work is delegated to specialist agents:

```
recon-agent
hunter-agent
chain-builder
brutalist-verifier
balanced-verifier
final-verifier
evidence-agent
grader
report-writer
```

The MCP server exposes 39 `bounty_*` tools. Those tools validate inputs, write state atomically, enforce phase gates, redact sensitive data, summarize context, and return standard envelopes:

```
{
  "ok": true,
  "data": {},
  "meta": {
    "tool": "bounty_read_state_summary",
    "version": 1
  }
}
```


3 Why This Architecture Exists

The design is a response to four hard problems in agentic security work.

3.1 Problem 1: Chat Context Is Too Small

A full bounty run can include:

- recon output
- live hosts
- archived URLs
- JavaScript endpoints
- imported HTTP traffic
- auth profiles
- hundreds of HTTP scan records
- hunter notes
- coverage records
- findings
- chain attempts
- verification rounds
- evidence samples
- grading
- report drafts

Pasting all of that into every agent would overwhelm the context window and make agents repeat work. Bob solves this by storing durable state in MCP-owned JSON and JSONL files, then giving each agent a small slice.

The most important context-reduction tool is:

```
mcp/lib/hunter-brief.js
```

That file builds a curated hunter brief from:

- assigned surface
- coverage summary
- imported traffic summary
- HTTP audit summary
- circuit-breaker summary
- ranking summary
- public intel hints
- static scan hints
- auth hint
- bypass table
- curated hunter techniques

Hunters receive one assigned surface, not the entire session.

3.2 Problem 2: Many Agents Need One Shared Memory

The root orchestrator and subagents are separate Claude Code contexts. A background hunter might finish after the root chat has moved on or after context compaction. Shared state cannot live only in a message transcript.

Bob therefore uses:

```
~/bounty-agent-sessions/[domain]/
```

as the durable memory store, with MCP tools as the only trusted writers for critical artifacts.

3.3 Problem 3: Agents Drift Unless Outputs Are Structured

Without strict structure, an agent might say “I found something” in prose but forget to write the machine-readable artifact. Bob makes each phase write a specific artifact:

- hunters write `handoff-wN-aN.json`
- hunters record findings in `findings.jsonl`
- chain builder writes `chain-attempts.jsonl`
- verifiers write `brutalist.json`, `balanced.json`, `verified-final.json`
- evidence agent writes `evidence-packs.json`
- grader writes `grade.json`
- reporter writes `report.md`

Phase gates inspect those structured artifacts. Prose alone does not count.

3.4 Problem 4: Security Automation Needs Guardrails

Bob can send real HTTP requests and store sensitive test data. The repo has guardrails in several layers:

- MCP input validation in `mcp/lib/tool-validation.js`
- path safety in `mcp/lib/paths.js`
- atomic writes and session locks in `mcp/lib/storage.js`
- redaction in `mcp/redaction.js`, `mcp/lib/http-records.js`, and evidence validation
- write guard hook in `.claude/hooks/session-write-guard.sh`
- hunter stop validation in `.claude/hooks/hunter-subagent-stop.js`
- phase gates in `mcp/lib/phase-gates.js`
- egress profile handling in `mcp/lib/egress-profiles.js`

4 Source Checkout Versus Installed Workspace

This repo is usually the source package. It installs Bob into another Claude Code project directory.

4.1 Source Repo

The source repo is where you develop Bob:

```
/Users/memehalis/sec/hacker-bob
```

Here you run:

```
npm test
node scripts/generate-agent-tools.js --check
node scripts/generate-bountyagent-skill.js --check
```

4.2 Installed Project

An installed project is where a user actually runs Claude Code and /bob-hunt:

```
/path/to/user/project
```

The installer copies these into that target:

```
.claude/
mcp/
.mcp.json
testing/policy-replay/
```

Important installer files:

```
bin/hacker-bob.js
install.sh
scripts/install.js
scripts/merge-claude-config.js
scripts/lifecycle.js
mcp/lib/claude-config.js
```

The CLI has these main commands:

```
hacker-bob install <project-dir>
hacker-bob update <project-dir>
hacker-bob check-update <project-dir>
hacker-bob doctor <project-dir>
hacker-bob uninstall <project-dir>
```

The hacker-bob package under packages/hacker-bob/ is only a convenience alias. The canonical package is hacker-bob-cc.

5 The Three Control Surfaces

Bob has three main control surfaces.

5.1 1. Slash Commands And Skills

Files:

```
.claude/skills/bob-hunt/SKILL.md
.claude/skills/bob-status/SKILL.md
.claude/skills/bob-debug/SKILL.md
.claude/commands/bob-update.md
.claude/commands/bob-egress.md
```

Claude Code turns these into user-facing commands:

```
/bob-hunt
/bob-status
/bob-debug
/bob-update
/bob-egress
```

The most important file is `bob-hunt/SKILL.md`. It is the orchestrator prompt. It defines:

- accepted arguments
- flags such as `--no-auth` and `--egress`
- the finite-state machine
- which MCP tools are authoritative
- how to spawn each specialist agent
- when to wait
- when to stop
- when to merge a wave
- which artifacts to validate before moving forward

5.2 2. Agent Definitions

Files:

```
.claude/agents/recon-agent.md
.claude/agents/hunter-agent.md
.claude/agents/chain-builder.md
.claude/agents/brutalist-verifier.md
.claude/agents/balanced-verifier.md
.claude/agents/final-verifier.md
.claude/agents/evidence-agent.md
.claude/agents/grader.md
.claude/agents/report-writer.md
```

Each agent has:

- a role
- allowed tools
- model preference
- MCP server metadata when needed
- a final-output contract

Example: hunters do not get the `Write` tool. They can write durable hunt artifacts only through MCP tools such as:

```
bounty_record_finding
bounty_log_coverage
bounty_write_wave_handoff
```

5.3 3. MCP Runtime

Files:

```
mcp/server.js  
mcp/lib/transport.js  
mcp/lib/tool-registry.js  
mcp/lib/dispatch.js  
mcp/lib/tools/index.js  
mcp/lib/tools/*.js  
mcp/lib/*.js
```

Claude Code talks to `mcp/server.js` over stdio. The server exposes `bounty_*` tools. It validates, dispatches, records telemetry, and returns standard envelopes.

6 The Full Runtime Flow

This is the simplified execution path when the user runs:

```
/bob-hunt example.com
```

User

|

v

```
/bob-hunt skill
```

|

v

Root orchestrator prompt

|

```
+--> MCP: bounty_init_session
```

|

```
+--> Task: recon-agent
```

|

|

```
+--> writes attack_surface.json
```

|

```
+--> MCP: bounty_transition_phase AUTH
```

|

```
+--> AUTH tools: signup detect, temp email, auth store
```

|

```
+--> MCP: bounty_transition_phase HUNT
```

|

```
+--> MCP: bounty_start_wave
```

|

|

```
+--> writes wave-N-assignments.json
```

|

```
+--> sets state.pending_wave = N
```

|

```
+--> background hunter agents
```

|

|

```
+--> MCP: bounty_read_hunter_brief
```

|

```
+--> MCP: bounty_http_scan
```

|

```
+--> MCP: bounty_log_coverage
```

|

```
+--> MCP: bounty_record_finding
```

|

```
+--> MCP: bounty_write_wave_handoff
```

|

```
+--> final marker: BOB_HUNTER_DONE {...}
```

|

```
+--> hunter SubagentStop hook validates marker + handoff
```

|

```
+--> MCP: bounty_apply_wave_merge
```

|

|

```
+--> validates handoffs
```

|

```
+--> updates explored/dead_ends/leads/findings count
```

|

```
+--> clears pending_wave
```

|

```
+--> more waves or CHAIN
```

|

```
+--> chain-builder
```

|

|

```
+--> writes chain-attempts.jsonl
```

|

```
+--> brutalist verifier
```

```
+--> balanced verifier
```

```
+--> final verifier
```

|

```
+--> evidence-agent
|
|   +--> writes evidence-packs.json
|
+--> grader
|
|   +--> writes grade.json
|
+--> report-writer
|
|   +--> writes report.md
```

The orchestration principle:

The root orchestrator coordinates state and agents.

It does not do ad-hoc target testing outside the AUTH path.

7 The Finite-State Machine

The state machine is implemented in:

```
mcp/lib/session-state.js
mcp/lib/phase-gates.js
```

Valid phases:

```
RECON
AUTH
HUNT
CHAIN
VERIFY
GRADE
REPORT
EXPLORE
```

Allowed transitions:

```
RECON  -> AUTH
AUTH   -> HUNT
HUNT   -> CHAIN
CHAIN  -> VERIFY
VERIFY -> GRADE
GRADE  -> REPORT
GRADE  -> HUNT      # on HOLD
REPORT -> EXPLORE   # user asks for more hunting
EXPLORE -> CHAIN
```

Important transition gates:

7.1 HUNT -> CHAIN

Blocked when:

- `pending_wave` is still set
- attack surface cannot be read
- high or critical surfaces remain unexplored
- latest coverage contains unfinished `promising`, `needs_auth`, or `requeue` entries

Code:

```
mcp/lib/phase-gates.js
computeHuntToChainGate()
```

7.2 CHAIN -> VERIFY

Blocked when chain work is required but no terminal chain attempt exists.

Chain work is required when:

- there are multiple findings, or
- structured wave handoffs contain `chain_notes`

Terminal outcomes:

```
confirmed
denied
blocked
not_applicable
```

Code:


```
mcp/lib/phase-gates.js  
computeChainToVerifyGate()
```

7.3 VERIFY -> GRADE

Blocked when final reportable findings exist but valid evidence packs are missing.

Code:

```
mcp/lib/phase-gates.js  
computeVerifyToGradeGate()  
mcp/lib/evidence.js  
requireValidEvidencePacksForFinalReportableFindings()
```

7.4 GRADE -> REPORT

Also checks evidence validity. The report should only be trusted after final verification, evidence, and grade artifacts exist and validate.

8 Session State

The initial state is created by:

```
mcp/lib/session-state.js
buildInitialSessionState()
initSession()
```

Example simplified `state.json`:

```
{
  "target": "example.com",
  "target_url": "https://example.com",
  "phase": "HUNT",
  "hunt_wave": 2,
  "pending_wave": null,
  "total_findings": 1,
  "explored": ["api-main", "auth-flow"],
  "dead_ends": ["cdn.example.com static-only"],
  "waf_blocked_endpoints": ["/admin/export"],
  "lead_surface_ids": ["billing-api"],
  "scope_exclusions": [],
  "hold_count": 0,
  "auth_status": "authenticated"
}
```

Key fields:

`phase`

Current FSM phase.

`hunt_wave`

Last merged wave number.

`pending_wave`

Wave started but not reconciled yet. If this is not null, the correct path is usually `/bob-hunt resume <domain>`.

`explored`

Surface IDs completed by merged wave handoffs.

`dead_ends`

Durable endpoint/host notes that should not be retried blindly.

`waf_blocked_endpoints`

Endpoint classes where hunters hit hard blocking.

`lead_surface_ids`

Existing attack surface IDs that later waves should prioritize.

`total_findings`

Updated during wave merge from `findings.jsonl` summary.

`hold_count`

Incremented when GRADE sends the workflow back to HUNT.

`auth_status`

pending, authenticated, or unauthenticated.

There are two read styles:

bounty_read_session_state
Full public state arrays.

bounty_read_state_summary
Compact counts for routine orchestration.

This is one context-management pattern used throughout Bob: prefer summaries unless the full arrays are needed.

9 JSON, JSONL, And Markdown

Bob uses three artifact styles.

9.1 JSON

Use JSON for current authoritative documents:

```
state.json
attack_surface.json
wave-1-assignments.json
handoff-w1-a1.json
brutalist.json
balanced.json
verified-final.json
evidence-packs.json
grade.json
public-intel.json
```

JSON files are best when there is one current document to read and validate.

9.2 JSONL

Use JSONL for append-only event streams:

```
findings.jsonl
coverage.jsonl
http-audit.jsonl
traffic.jsonl
chain-attempts.jsonl
pipeline-events.jsonl
static-artifacts.jsonl
static-scan-results.jsonl
```

Each line is a complete JSON object. This is useful when agents keep appending records over time without rewriting a whole file.

9.3 Markdown

Use Markdown for human-readable mirrors:

```
findings.md
brutalist.md
balanced.md
verified-final.md
evidence-packs.md
grade.md
SESSION_HANDOFF.md
report.md
handoff-w1-a1.md
```

Important distinction:

Machine-readable orchestration should use JSON and JSONL.

Markdown is for humans and debugging.

The exception is `report.md`, which is the final human-facing output written by the report writer.

10 Core Artifact Table

Artifact	Format	Owner	Purpose
<code>state.json</code>	JSON	MCP	FSM phase, waves, explored surfaces, auth status
<code>attack_surface.json</code>	JSON	recon-agent	Recon output and hunter surface list
<code>traffic.jsonl</code>	JSONL	MCP	Imported HAR/Burp-style traffic
<code>http-audit.jsonl</code>	JSONL	MCP	Redacted record of Bob HTTP scan calls
<code>wave-N-assignments.json</code>	JSON	MCP	Which hunter owns which surface
<code>handoff-wN-aN.json</code>	JSON	MCP	Authoritative hunter final handoff
<code>handoff-wN-aN.md</code>	Markdown	MCP	Human mirror of hunter handoff
<code>coverage.jsonl</code>	JSONL	MCP	Endpoint/class/auth coverage records
<code>findings.jsonl</code>	JSONL	MCP	Proven findings recorded by hunters
<code>chain-attempts.jsonl</code>	JSONL	MCP	Tested chain hypotheses and outcomes
<code>brutalist.json</code>	JSON	MCP	Verification round 1
<code>balanced.json</code>	JSON	MCP	Verification round 2
<code>verified-final.json</code>	JSON	MCP	Final verification round
<code>evidence-packs.json</code>	JSON	MCP	Bounded evidence for final reportables
<code>grade.json</code>	JSON	MCP	SUBMIT/HOLD/SKIP verdict
<code>report.md</code>	Markdown	report-writer	Final output for the operator
<code>pipeline-events.jsonl</code>	JSONL	MCP	Phase/wave/finding/verification/grade events
tool telemetry	JSONL	MCP	Safe metadata for MCP tool health

11 MCP Server Architecture

Start here:

`mcp/server.js`

This file is a facade. It imports most runtime helpers and exports them for tests and hooks. When run as a process, it starts the stdio MCP server:

```
startServer()  
  -> startStdioServer({ tools: TOOLS, executeTool })
```

11.1 Transport

File:

`mcp/lib/transport.js`

Responsibilities:

- handle MCP initialize
- answer ping
- return `tools/list`
- execute `tools/call`
- support both framed `Content-Length` messages and raw JSON lines
- serialize tool envelopes back into MCP text content

11.2 Registry

File:

`mcp/lib/tool-registry.js`

The registry imports:

`mcp/lib/tools/index.js`

Each tool entry must define:

```
name  
description  
inputSchema  
handler  
role_bundles  
mutating  
global_preapproval  
network_access  
browser_access  
scope_required  
sensitive_output  
session_artifacts_written  
hook_required
```

This metadata drives:

- MCP tool exposure
- Claude settings permissions
- agent allowed tools
- prompt contract tests
- hooks for scope-sensitive tools
- docs and generated surfaces

11.3 Dispatch

File:

mcp/lib/dispatch.js

Flow:

```
executeTool(name, args)
  -> look up tool
  -> validate input schema
  -> call handler
  -> parse handler result
  -> classify data errors
  -> wrap in ok/error envelope
  -> record safe telemetry
```

11.4 Envelopes

File:

mcp/lib/envelope.js

Success:

```
{
  "ok": true,
  "data": {
    "version": 1
  },
  "meta": {
    "tool": "bounty_example",
    "version": 1
  }
}
```

Failure:

```
{
  "ok": false,
  "error": {
    "code": "STATE_CONFLICT",
    "message": "Wave merge requires pending_wave to be set"
  },
  "meta": {
    "tool": "bounty_apply_wave_merge",
    "version": 1
  }
}
```

The orchestrator prompt explicitly says: use `.data` on success and `.error` on failure. Do not infer success from random top-level fields.

12 The 39 MCP Tools By Role

The tools are registered in `mcp/lib/tools/index.js` and metadata is enforced by `mcp/lib/tool-registry.js`.

12.1 Orchestrator Tools

```
bounty_init_session
bounty_read_session_state
bounty_read_state_summary
bounty_transition_phase
bounty_start_wave
bounty_apply_wave_merge
bounty_wave_status
bounty_wave_handoff_status
bounty_merge_wave_handoffs
bounty_read_wave_handoffs
bounty_write_handoff
bounty_import_http_traffic
bounty_public_intel
bounty_list_findings
bounty_read_chain_attempts
bounty_read_verification_round
bounty_read_evidence_packs
bounty_read_grade_verdict
bounty_list_auth_profiles
bounty_read_tool_telemetry
bounty_read_pipeline_analytics
```

12.2 Hunter Tools

```
bounty_read_hunter_brief
bounty_http_scan
bounty_read_http_audit
bounty_import_static_artifact
bounty_static_scan
bounty_record_finding
bounty_list_findings
bounty_log_coverage
bounty_log_dead_ends
bounty_write_wave_handoff
bounty_list_auth_profiles
```

12.3 Auth Tools

```
bounty_signup_detect
bounty_temp_email
bounty_auto_signup
bounty_auth_store
bounty_list_auth_profiles
bounty_http_scan
```

12.4 Chain Tools

```
bounty_read_findings
bounty_read_wave_handoffs
bounty_read_chain_attempts
bounty_write_chain_attempt
bounty_read_http_audit
```


bounty_list_auth_profiles
bounty_http_scan

12.5 Verification Tools

bounty_read_findings
bounty_read_chain_attempts
bounty_read_verification_round
bounty_write_verification_round
bounty_read_http_audit
bounty_list_auth_profiles
bounty_http_scan

12.6 Evidence, Grade, Report Tools

Evidence:

bounty_read_findings
bounty_read_verification_round
bounty_read_http_audit
bounty_list_auth_profiles
bounty_http_scan
bounty_write_evidence_packs
bounty_read_evidence_packs

Grade:

bounty_read_findings
bounty_read_chain_attempts
bounty_read_verification_round
bounty_read_evidence_packs
bounty_write_grade_verdict
bounty_read_grade_verdict

Report:

bounty_read_findings
bounty_read_chain_attempts
bounty_read_verification_round
bounty_read_evidence_packs
bounty_read_grade_verdict

13 Context Management In Detail

This is the central design idea in the repo.

13.1 Context Is Not Shared By Default

The root orchestrator, hunters, verifiers, chain builder, grader, and reporter are separate agent runs. They do not automatically share all chat history.

Bob treats the chat transcript as unreliable for durable coordination. The shared memory is MCP-owned artifacts.

13.2 The Root Orchestrator Gets Summaries

The root orchestrator usually calls:

```
bounty_read_state_summary
bounty_wave_status
bounty_read_pipeline_analytics
```

These are compact. They avoid loading full arrays unless needed.

13.3 Hunters Get One Curated Brief

Hunter first action:

```
bounty_read_hunter_brief({
  target_domain: "...",
  wave: "w1",
  agent: "a1"
})
```

The returned brief includes one assigned surface and bounded summaries.

File:

```
mcp/lib/hunter-brief.js
```

Key context controls in that file:

- caps scalar strings
- caps arrays
- reports omitted counts
- includes only the assigned surface
- filters dead ends by relevant hosts
- summarizes coverage by latest endpoint/method/bug/auth key
- summarizes traffic relevant to the surface
- summarizes HTTP audit and circuit breaker state
- adds public intel and static scan hints when available
- gives auth guidance without exposing secrets
- includes curated technique hints from `.claude/knowledge/hunter-techniques.json`

This is how Bob avoids pasting the whole session into every hunter.

13.4 Verification Reads Structured Artifacts

Verifiers do not trust `findings.md`. They read:

```
bounty_read_findings
bounty_read_chain_attempts
bounty_read_verification_round
bounty_read_http_audit
```

Each verification round writes a full JSON result set. Balanced and final rounds must include every finding from the prior round so findings are not silently dropped.

Code:

```
mcp/lib/findings.js
writeVerificationRound()
requirePriorVerificationRound()
```

13.5 Evidence Is Separate From Findings

Hunters record claims in `findings.jsonl`. Final verification decides which claims are reportable. The evidence agent then collects bounded, redacted samples in `evidence-packs.json`.

This separation matters because raw PoC text is not enough for a trustworthy report. The evidence pack is the structured proof layer used by grading and report writing.

13.6 Analytics Summarizes The Whole Pipeline

File:

```
mcp/lib/pipeline-analytics.js
```

It reads artifacts and telemetry to answer:

- what phase is this session in?
- is a wave pending?
- are handoffs missing?
- did hunters get blocked?
- did tools fail often?
- did final verification drop all findings?
- are evidence packs missing?
- is grade/report present?
- did network reachability or egress look bad?

This powers `/bob-status` and `/bob-debug`.

14 Agent Architecture

Each agent is narrow on purpose.

14.1 Root Orchestrator

File:

`.claude/skills/bob-hunt/SKILL.md`

Responsibilities:

- parse command arguments and flags
- initialize or resume session
- transition phases
- spawn agents
- start and merge waves
- validate artifacts before proceeding
- choose next command or stop point

Not responsible for:

- ad-hoc target testing
- writing MCP-owned files directly
- synthesizing hunter handoff JSON
- repairing missing structured artifacts from Markdown

14.2 Recon Agent

File:

`.claude/agents/recon-agent.md`

It uses Bash only. It is intentionally MCP-free.

It has exactly seven collection Bash steps:

1. tool check
2. subdomain enum
3. live hosts
4. family discovery
5. CDX archived URLs
6. nuclei
7. JavaScript extraction

Then it writes:

`attack_surface.json`

This is one of the few agent-owned direct writes allowed in the session dir.

14.3 Hunter Agent

File:

`.claude/agents/hunter-agent.md`

Responsibilities:

- read assigned brief
- test one surface only
- use auth profiles when available
- prefer traffic-derived endpoints
- call `bounty_http_scan` for audited requests
- log meaningful coverage

- record only proven findings
- write exactly one wave handoff
- emit exactly one final marker

Important constraints:

- no `Write` tool
- do not directly write MCP-owned files
- do not scan arbitrary local static paths
- do not record weak standalone non-findings
- stop and request egress rotation after repeated first-party network failures

14.4 Chain Builder

File:

`.claude/agents/chain-builder.md`

It reads structured findings, handoffs, audit records, auth profiles, and prior chain attempts. It writes every tested hypothesis to:

`chain-attempts.jsonl`

It must not use Markdown handoffs as machine input.

14.5 Three Verifiers

Files:

`.claude/agents/brutalist-verifier.md`

`.claude/agents/balanced-verifier.md`

`.claude/agents/final-verifier.md`

Round 1: brutalist

- skeptical replay
- deny weak claims
- downgrade inflated severity

Round 2: balanced

- catch false negatives
- review brutalist decisions
- include every prior finding

Round 3: final

- re-run only reportable survivors with fresh requests
- include every prior finding
- final `reportable: true` results drive evidence collection

14.6 Evidence Agent

File:

`.claude/agents/evidence-agent.md`

Collects bounded, redacted evidence packs for final reportable findings only. It writes:

`evidence-packs.json`

14.7 Grader

File:

`.claude/agents/grader.md`

Scores final survivors on:

- impact
- proof quality
- severity accuracy
- chain potential
- report quality

Verdicts:

SUBMIT

HOLD

SKIP

14.8 Report Writer

File:

`.claude/agents/report-writer.md`

Writes:

`report.md`

It reads final verification, evidence packs, chain attempts, and grade. It should include only verified, evidenced, graded findings.

15 Wave Lifecycle

Waves are the core HUNT-phase coordination mechanism.

15.1 Start A Wave

The orchestrator computes assignments and calls:

```
bounty_start_wave({
  target_domain: "example.com",
  wave_number: 1,
  assignments: [
    { "agent": "a1", "surface_id": "api-main" },
    { "agent": "a2", "surface_id": "auth-flow" }
  ]
})
```

Code:

```
mcp/lib/waves.js
startWave()
```

Effects:

- validates phase is HUNT or EXPLORE
- validates no `pending_wave`
- validates wave number is `hunt_wave + 1`
- validates surface IDs exist in `attack_surface.json`
- generates a secret handoff token for each assignment
- writes `wave-N-assignments.json` with token hashes
- returns plain handoff tokens to the orchestrator
- sets `state.pending_wave = N`

The token is given only to the assigned hunter. The token prevents another agent from writing a fake authoritative handoff.

15.2 Spawn Background Hunters

The orchestrator spawns one `hunter-agent` per assignment with:

```
run_in_background: true
```

Each hunter prompt includes:

- domain
- wave
- agent ID
- assigned handoff token
- egress profile
- first action: call `bounty_read_hunter_brief`
- final action: call `bounty_write_wave_handoff`
- final marker: `BOB_HUNTER_DONE {...}`

15.3 Hunter Writes Handoff

Example simplified handoff:

```
{
  "target_domain": "example.com",
  "wave": "w1",
  "agent": "a2",
  "surface_id": "auth-flow",
  "surface_status": "partial",
```

```

    "provenance": "verified",
    "summary": "Tested signup, reset, invite, and OAuth callback flows.",
    "chain_notes": [
        "Password reset token leak may combine with weak session rotation."
    ],
    "dead_ends": [
        "/logout is static redirect only"
    ],
    "waf_blocked_endpoints": [],
    "lead_surface_ids": [
        "billing-api"
    ]
}

```

Code:

```

mcp/lib/waves.js
writeWaveHandoff()

```

15.4 SubagentStop Hook Validates Completion

File:

```
.claude/hooks/hunter-subagent-stop.js
```

When a hunter stops, the hook checks:

- did the final assistant message include `BOB_HUNTER_DONE?`
- is the marker valid JSON?
- does the marker include target, wave, agent, surface?
- does a structured handoff JSON exist?
- does the handoff match the marker?
- is the handoff provenance verified?

If validation fails, the hook blocks the subagent stop.

Important: the hook does not merge waves. It only validates completion and records telemetry.

15.5 Merge A Wave

After background hunters finish, the orchestrator calls:

```

bounty_apply_wave_merge({
  target_domain: "example.com",
  wave_number: 1,
  force_merge: false
})

```

Code:

```

mcp/lib/waves.js
applyWaveMerge()

```

Effects:

- reads assignments
- checks handoff readiness
- validates handoff payloads
- computes completed and partial surfaces
- includes live dead-end logs
- computes requeue surface IDs
- updates `explored`
- updates `dead_ends`

- updates `waf_blocked_endpoints`
- updates `lead_surface_ids`
- clears `pending_wave`
- sets `hunt_wave`
- updates `total_findings`
- appends a pipeline event

If not all handoffs are present and `force_merge` is false, it returns:

```
{  
  "status": "pending"  
}
```

and does not mutate session state.

16 How Findings Flow Through The Pipeline

16.1 1. Hunter Records A Finding

Tool:

bounty_record_finding

Code:

```
mcp/lib/findings.js  
recordFinding()
```

Simplified record:

```
{  
  "id": "F-1",  
  "target_domain": "example.com",  
  "title": "IDOR in invoice export exposes victim invoices",  
  "severity": "high",  
  "cwe": "CWE-639",  
  "endpoint": "https://app.example.com/api/invoices/export?id=123",  
  "description": "The endpoint accepts arbitrary invoice IDs.",  
  "proof_of_concept": "curl ...",  
  "response_evidence": "200 response includes victim invoice metadata",  
  "impact": "Attacker can read billing data for other accounts.",  
  "validated": true,  
  "wave": "w1",  
  "agent": "a2",  
  "surface_id": "billing-api",  
  "auth_profile": "attacker",  
  "dedupe_key": "..."  
}
```

`recordFinding()` also computes a dedupe key from endpoint, title/classification, auth context, and evidence fingerprint. Exact duplicates are not recorded unless `force_record` is true.

16.2 2. Chain Builder Tests Combinations

Tool:

bounty_write_chain_attempt

Code:

```
mcp/lib/chain-attempts.js
```

Simplified record:

```
{  
  "version": 1,  
  "ts": "2026-05-01T00:00:00.000Z",  
  "attempt_id": "C-1",  
  "target_domain": "example.com",  
  "finding_ids": ["F-1", "F-2"],  
  "surface_ids": ["billing-api", "auth-flow"],  
  "hypothesis": "Invite leak plus IDOR could expose invoices without victim action.",  
  "steps": [  
    "Used attacker profile to replay leaked invite context.",  
    "Requested victim invoice ID through export endpoint."  
  ],  
  "outcome": "denied",  
  "evidence_summary": "Export endpoint still required invoice ownership.",  
}
```

```

    "request_refs": ["http-audit:42"],
    "auth_profiles": ["attacker", "victim"]
}

```

16.3 3. Verification Rounds Decide Survivors

Each verification result has:

```

{
  "finding_id": "F-1",
  "disposition": "confirmed",
  "severity": "high",
  "reportable": true,
  "reasoning": "Fresh replay returned victim invoice metadata."
}

```

The final round decides which findings require evidence.

16.4 4. Evidence Agent Collects Bounded Proof

Tool:

bounty_write_evidence_packs

Simplified pack:

```

{
  "finding_id": "F-1",
  "sample_type": "cross-account invoice access",
  "sample_count": 3,
  "aggregate_counts": {
    "affected_accounts_sampled": 3,
    "private_fields_observed": 5
  },
  "representative_samples": [
    {
      "request_ref": "http-audit:42",
      "endpoint": "/api/invoices/export",
      "auth_profile": "attacker",
      "status": 200,
      "observed_fields": ["account_id", "email", "invoice_total"],
      "redacted_object_id": "inv_...789"
    }
  ],
  "sensitive_clusters": ["invoice metadata"],
  "replay_summary": "Attacker replay returned private invoice metadata.",
  "redaction_notes": "IDs and personal values redacted.",
  "report_snippet": "An attacker can enumerate invoice exports and receive private billing metadata."
}

```

Evidence validation rejects secrets, auth headers, cookies, tokens, huge raw responses, duplicate packs, packs for non-reportable findings, and missing packs for final reportable findings.

16.5 5. Grader Produces A Verdict

Simplified grade:

```

{
  "version": 1,
  "target_domain": "example.com",
  "verdict": "SUBMIT",
}

```

```

"total_score": 72,
"findings": [
  {
    "finding_id": "F-1",
    "impact": 25,
    "proof_quality": 20,
    "severity_accuracy": 12,
    "chain_potential": 5,
    "report_quality": 10,
    "total_score": 72,
    "feedback": null
  }
],
"feedback": null
}

```

Code enforces:

- per-finding total equals sum of rubric scores
- grade `total_score` equals the max per-finding score
- verdict matches thresholds and final reportability
- evidence packs are valid before grade write/read succeeds

17 HTTP, Auth, Egress, And Audit

17.1 HTTP Scan

Tool:

bounty_http_scan

Code:

```
mcp/lib/http-scan.js
mcp/lib/safe-fetch.js
mcp/lib/url-surface.js
mcp/lib/http-records.js
```

Responsibilities:

- require `target_domain`
- resolve egress profile
- optionally attach auth profile headers
- optionally block internal/private hosts when `block_internal_hosts: true`
- send request through safe fetch
- cap response bytes
- analyze response for tech, endpoints, secrets, security hints
- append redacted audit record to `http-audit.jsonl`

Important design choice:

Bob permits localhost, private, internal, metadata-style, and third-party hosts by default. This keeps authorized chaining flexible. A caller can opt into blocking internal hosts with:

```
{
  "block_internal_hosts": true
}
```

17.2 Auth Storage

Tool:

bounty_auth_store

Code:

```
mcp/lib/auth.js
```

Auth profiles are stored in:

```
~/bounty-agent-sessions/[domain]/auth.json
```

The file is written with mode 0600. Profiles can be named:

```
attacker
victim
legacy
custom names
```

Agents do not read `auth.json` directly. They call:

```
bounty_list_auth_profiles
```

for redacted summaries and use:

```
auth_profile: "attacker"
```

in `bounty_http_scan` calls.

17.3 Egress Profiles

Code:

```
mcp/lib/egress-profiles.js
.claude/hooks/bob-egress.js
.claude/commands/bob-egress.md
```

Config:

```
.claude/bob/egress-profiles.json
```

Default profile:

```
{
  "name": "default",
  "proxy_url": null,
  "region": null,
  "description": "Direct connection from this machine.",
  "enabled": true
}
```

Non-default profiles can use environment variable references:

```
{
  "name": "gr-residential",
  "proxy_url": "${BOB_EGRESS_GR_RESIDENTIAL_PROXY}",
  "region": "GR",
  "enabled": false
}
```

Proxy URLs are redacted and should not be printed into chat.

17.4 Geofence Warnings

HTTP audit summaries detect repeated first-party timeouts or connection errors. When a host repeatedly fails, Bob treats it as a reachability warning. The orchestrator should ask the operator before switching egress profiles.

18 Hooks And Guardrails

18.1 Session Write Guard

File:

```
.claude/hooks/session-write-guard.sh
```

Registered for:

Bash

Write

It blocks direct writes to MCP-owned files under:

```
~/bounty-agent-sessions/
```

Examples of blocked files:

```
state.json
findings.jsonl
coverage.jsonl
http-audit.jsonl
traffic.jsonl
chain-attempts.jsonl
brutalist.json
balanced.json
verified-final.json
evidence-packs.json
grade.json
pipeline-events.jsonl
handoff-w1-a1.json
```

Examples of allowed agent-owned files:

```
attack_surface.json
report.md
chains.md
*.txt
```

The hook catches:

- Write tool writes
- Bash redirects
- `tee`
- inline Python `open(..., "w")`
- `Path(...).write_text(...)`
- common inline script write patterns

18.2 Hunter Stop Hook

File:

```
.claude/hooks/hunter-subagent-stop.js
```

Registered for:

SubagentStop: hunter-agent

It blocks hunters from finishing unless their final marker and structured handoff are valid.

It also records metadata-only hunter run telemetry.

18.3 Scope Guard Hooks

Files:

```
.claude/hooks/scope-guard.sh  
.claude/hooks/scope-guard-mcp.sh
```

In this version, the Bash scope guard is intentionally permissive. MCP HTTP tools can block internal/private hosts only when the caller passes `block_internal_hosts: true`.

This is documented and tested, not accidental.

19 Pipeline Analytics And Debugging

Bob records two major diagnostic streams.

19.1 Pipeline Events

File:

```
~/bounty-agent-sessions/[domain]/pipeline-events.jsonl
```

Event types:

```
session_started
phase_transitioned
wave_started
hunter_stopped
wave_merge_pending
wave_merged
coverage_logged
finding_recorded
verification_written
evidence_written
grade_written
```

Example:

```
{
  "version": 1,
  "ts": "2026-05-01T00:00:00.000Z",
  "target_domain": "example.com",
  "type": "wave_merged",
  "phase": "HUNT",
  "wave_number": 1,
  "status": "merged",
  "source": "bounty_apply_wave_merge",
  "counts": {
    "assignments": 2,
    "handoffs": 2,
    "findings": 1
  }
}
```

19.2 Tool Telemetry

File:

```
~/bounty-agent-telemetry/tool-events.jsonl
~/bounty-agent-telemetry/agent-runs.jsonl
```

Code:

```
mcp/lib/tool-telemetry.js
```

Telemetry is intentionally metadata-only:

- tool name
- ok/failure
- elapsed time
- error code
- target domain
- wave/agent/surface context
- registry metadata

It avoids storing raw secret-bearing payloads.

19.3 Pipeline Analytics Reader

Tool:

`bounty_read_pipeline_analytics`

Code:

`mcp/lib/pipeline-analytics.js`

It can read one session or recent sessions and summarize:

- phase
- waves
- pending handoffs
- findings
- chain attempts
- final verification count
- evidence status
- egress/geofence warnings
- grade
- report presence
- health status
- bottlenecks
- next actions
- tool health
- hunter health

This is the core behind `/bob-status` and `/bob-debug`.

20 How To Read The Repo Step By Step

This is the recommended offline reading order.

20.1 Step 1: Read The Product Contract

Open:

```
README.md
CLAUDE.md
CONTRIBUTING.md
docs/FIRST_RUN.md
docs/TROUBLESHOOTING.md
```

Questions to answer:

- What does Bob install?
- What does the user run?
- Where does run state live?
- What is `bountyagent`?
- What is the difference between `hacker-bob-cc` and `hacker-bob`?

20.2 Step 2: Understand Install And Lifecycle

Open:

```
bin/hacker-bob.js
install.sh
scripts/install.js
scripts/merge-claude-config.js
scripts/lifecycle.js
mcp/lib/claude-config.js
```

Trace:

```
hacker-bob install /target
-> scripts/install.js installProject()
-> copy .claude agents/skills/hooks/rules/knowledge
-> copy mcp runtime
-> merge .mcp.json
-> merge .claude/settings.json
-> write .claude/bob/VERSION and install.json
```

Questions to answer:

- Which files are copied?
- Which config is merged instead of overwritten?
- How are hooks registered?
- How does uninstall know what is Bob-managed?

20.3 Step 3: Read The Orchestrator Prompt

Open:

```
.claude/skills/bob-hunt/SKILL.md
```

Do not skim too quickly. This file is the operational spec.

Mark these sections:

- Flags
- Hard Rules
- FSM

- Resume
- RECON
- AUTH
- HUNT
- CHAIN
- VERIFY
- GRADE
- REPORT
- EXPLORE

Questions to answer:

- Why does the root orchestrator avoid target testing?
- Why are hunter waves backgrounded?
- Why is same-turn wave merge forbidden after spawn?
- What artifacts are validated before moving phases?
- What happens after HOLD?

20.4 Step 4: Read The Agent Prompts

Open in this order:

```
.claude/agents/recon-agent.md
.claude/agents/hunter-agent.md
.claude/agents/chain-builder.md
.claude/agents/brutalist-verifier.md
.claude/agents/balanced-verifier.md
.claude/agents/final-verifier.md
.claude/agents/evidence-agent.md
.claude/agents/grader.md
.claude/agents/report-writer.md
```

For each agent, write down:

- What can it read?
- What can it write?
- Which MCP tools does it use?
- What must its final action be?
- Which artifact does it own?

20.5 Step 5: Read MCP From Entry Point To Tool Handler

Open:

```
mcp/server.js
mcp/lib/transport.js
mcp/lib/tool-registry.js
mcp/lib/dispatch.js
mcp/lib/envelope.js
mcp/lib/tool-validation.js
mcp/lib/tools/index.js
```

Trace one tool call:

```
MCP tools/call
-> transport handleMessage()
-> executeTool(name, args)
-> validateToolArguments()
-> tool.handler(args)
-> parseHandlerResult()
```

```
-> okEnvelope() or errorEnvelope()
-> safeRecordToolTelemetry()
```

Questions to answer:

- Where are tool schemas defined?
- Where are unknown arguments rejected?
- Where are errors classified?
- How does telemetry avoid changing tool behavior?

20.6 Step 6: Read Session State And Storage

Open:

```
mcp/lib/paths.js
mcp/lib/storage.js
mcp/lib/session-state.js
mcp/lib/phase-gates.js
```

Questions to answer:

- How does `target_domain` become a safe path?
- What is the session lock?
- Which transitions are allowed?
- Which transitions have gates?
- What happens when a lock is stale?

20.7 Step 7: Read Waves And Hunter Context

Open:

```
mcp/lib/assignments.js
mcp/lib/waves.js
mcp/lib/hunter-brief.js
mcp/lib/coverage.js
mcp/lib/ranking.js
```

Trace:

```
bounty_start_wave
-> write wave assignment with token hash
-> return plain token to orchestrator
```

```
hunter
-> bounty_read_hunter_brief
-> bounty_log_coverage
-> bounty_write_wave_handoff
```

```
bounty_apply_wave_merge
-> validate handoffs
-> update state
```

Questions to answer:

- Why are handoff tokens hashed on disk?
- What makes a handoff valid?
- How does a partial handoff get requeued?
- How does coverage create requeue surface IDs?
- How is hunter context capped?

20.8 Step 8: Read Findings, Chain, Verification, Evidence, Grade

Open:

```
mcp/lib/findings.js
mcp/lib/chain-attempts.js
mcp/lib/evidence.js
```

Questions to answer:

- How are duplicate findings suppressed?
- Why must balanced/final rounds include all prior findings?
- What makes evidence valid?
- How does grade consistency get enforced?
- Why can no-finding sessions still continue to SKIP and report?

20.9 Step 9: Read HTTP, Auth, Egress, Redaction

Open:

```
mcp/lib/http-scan.js
mcp/lib/safe-fetch.js
mcp/lib/url-surface.js
mcp/lib/http-records.js
mcp/lib/auth.js
mcp/lib/signup.js
mcp/lib/temp-email.js
mcp/lib/egress-profiles.js
mcp/redaction.js
```

Questions to answer:

- Why does `bounty_http_scan` require `target_domain`?
- What gets written to `http-audit.jsonl`?
- Where are proxy URLs redacted?
- How are auth profiles resolved?
- When are internal hosts blocked?

20.10 Step 10: Read Hooks And Tests

Open:

```
.claude/hooks/session-write-guard.sh
.claude/hooks/hunter-subagent-stop.js
test/test-write-guard.py
test/prompt-contracts.test.js
test/mcp-server.test.js
test/install-smoke.test.js
test/cli.test.js
```

Questions to answer:

- Which behaviors are contractual?
- Which prompt phrases are tests protecting?
- Which state machine gates have test coverage?
- Which security boundaries are tested?

21 Useful Offline Commands

Run from the repo root.

21.1 List Files

```
rg --files
```

21.2 Find Tool Definitions

```
rg "name: \"bounty_\" mcp/lib/tools
```

21.3 See Tool Registry Summary

```
node - <<'NODE'  
const { TOOL_REGISTRY } = require('./mcp/lib/tool-registry.js');  
for (const t of TOOL_REGISTRY) {  
  console.log(`${t.name} roles=${t.role_bundles.join(',') } writes=${t.session_artifacts_written.join(',') }`  
}  
NODE
```

21.4 Run Focused Tests

```
npm run test:mcp  
npm run test:prompts  
npm run test:install  
npm run test:cli  
npm run test:hooks
```

21.5 Run Full Test Suite

```
npm test
```

21.6 Check Generated Prompt Surfaces

```
node scripts/generate-agent-tools.js --check  
node scripts/generate-bountyagent-skill.js --check
```

21.7 Inspect Installed-Style MCP Server

```
node -e "const s=require('./mcp/server.js'); console.log(s.TOOLS.length)"
```

21.8 Search For Artifact Ownership Rules

```
rg "MCP-owned|handoff|evidence-packs|pending_wave|phase" .claude mcp test
```

22 Suggested Flight Reading Plan

22.1 30 Minutes: Product Shape

Read:

README.md
CLAUDE.md
docs/FIRST_RUN.md

Goal: understand what Bob is from the user's point of view.

22.2 45 Minutes: Orchestrator And Agents

Read:

.claude/skills/bob-hunt/SKILL.md
.claude/agents/*.md

Goal: understand the role contract for each agent.

22.3 60 Minutes: MCP Core

Read:

mcp/server.js
mcp/lib/transport.js
mcp/lib/tool-registry.js
mcp/lib/dispatch.js
mcp/lib/envelope.js
mcp/lib/tools/index.js

Goal: understand how Claude Code calls become validated local functions.

22.4 90 Minutes: State, Waves, Context

Read:

mcp/lib/session-state.js
mcp/lib/phase-gates.js
mcp/lib/waves.js
mcp/lib/hunter-brief.js
mcp/lib/coverage.js

Goal: understand how Bob manages many agents without relying on chat memory.

22.5 60 Minutes: Findings To Report

Read:

mcp/lib/findings.js
mcp/lib/chain-attempts.js
mcp/lib/evidence.js

Goal: understand how a raw hunter claim becomes a trusted final report.

22.6 45 Minutes: Safety And Diagnostics

Read:

.claude/hooks/*.js
.claude/hooks/*.sh
mcp/lib/pipeline-analytics.js
mcp/lib/tool-telemetry.js

Goal: understand how Bob catches drift, missing artifacts, and stuck sessions.

22.7 45 Minutes: Tests As Documentation

Read test names first:

```
rg "^(test|describe)\\(" test testing -n
```

Then inspect the tests that match the parts you still find confusing.

23 Annotated Golden Run

This chapter follows one fictional run for `example.test`. It is not a live target transcript. It is a deliberately clean path through the system so you can keep the moving parts in your head while reading the code.

Use this as the reference story:

```
/bob-hunt https://example.test --normal --egress default
```

The whole run is an artifact pipeline:

```
skill prompt
-> MCP session state
-> recon artifact
-> auth artifact
-> wave assignments
-> hunter briefs
-> HTTP audit, coverage, findings, handoffs
-> wave merge
-> chain attempts
-> three verification rounds
-> evidence packs
-> grade
-> report
```

If you only remember one thing, remember this: the chat is not the database. The session directory is the database, and MCP tools are the write API.

23.1 Step 0: Claude Code Loads The Control Surface

The operator types `/bob-hunt`. Claude Code loads:

```
.claude/skills/bob-hunt/SKILL.md
```

That skill is the root orchestrator contract. It says:

- which MCP tools the root may call
- which agents the root may spawn
- which phases are legal
- when waves can start and merge
- which artifacts are authoritative
- what the root must never do itself

Important detail: the root orchestrator has `Task`, `Read`, and selected MCP tools. It is intentionally not a hunter. It coordinates the state machine.

23.2 Step 1: Session Initialization

First meaningful write:

```
bounty_init_session({
  target_domain: "example.test",
  target_url: "https://example.test"
})
```

Code path:

```
mcp/lib/tools/init-session.js
-> mcp/lib/session-state.js
-> mcp/lib/storage.js
-> mcp/lib/pipeline-analytics.js
```

Artifact written:

~/bounty-agent-sessions/example.test/state.json

Simplified state:

```
{
  "target": "example.test",
  "target_url": "https://example.test",
  "phase": "RECON",
  "hunt_wave": 0,
  "pending_wave": null,
  "total_findings": 0,
  "explored": [],
  "dead_ends": [],
  "waf_blocked_endpoints": [],
  "lead_surface_ids": [],
  "scope_exclusions": [],
  "hold_count": 0,
  "auth_status": "pending"
}
```

Pipeline event appended:

```
pipeline-events.jsonl
event_type = session_started
source = bounty_init_session
```

The event log is not the state itself. It is the audit trail that explains how the state changed over time.

23.3 Step 2: Recon Agent Produces The Attack Surface

The root spawns:

```
Agent(subagent_type: "recon-agent", name: "recon", prompt: "DOMAIN=example.test SESSION=...")
```

Agent contract:

```
.claude/agents/recon-agent.md
```

The recon agent is unusual because it uses Bash and writes one authoritative agent-owned artifact directly:

```
attack_surface.json
```

It also creates recon scratch files such as:

```
subdomains.txt
live_hosts.txt
family_live.txt
all_urls.txt
nuclei_results.txt
js_endpoints.txt
js_secrets.txt
```

Those scratch files help build the final attack surface, but later orchestration mostly cares about `attack_surface.json`.

After recon, the root reads `attack_surface.json`. If it is missing or empty, the run stops. If it contains surfaces, the root transitions:

```
bounty_transition_phase({
  target_domain: "example.test",
  to_phase: "AUTH"
})
```

State change:

RECON -> AUTH

23.4 Step 3: Auth Either Stores Profiles Or Marks The Run Unauthenticated

In normal mode, AUTH attempts signup/login capture. The root may use:

```
bounty_signup_detect
bounty_temp_email
bounty_http_scan
bounty_auto_signup
bounty_auth_store
bounty_list_auth_profiles
```

The durable output is:

```
auth.json
```

Typical profiles:

```
attacker
victim
legacy
```

The important design point is that agents should not need raw secrets in their prompt context. They ask MCP for profile summaries and pass `auth_profile` into `bounty_http_scan`. The HTTP layer applies stored auth material.

If the operator used `--no-auth`, the root does not try signup. It transitions:

```
bounty_transition_phase({
  target_domain: "example.test",
  to_phase: "HUNT",
  auth_status: "unauthenticated"
})
```

Otherwise successful auth transitions with:

```
auth_status: "authenticated"
```

23.5 Step 4: The Root Starts A Hunt Wave

Before every wave, the root reads compact state and attack surface context:

```
bounty_read_state_summary
bounty_wave_status
Read attack_surface.json
```

It computes assignments. A simple wave 1 might be:

```
[
  { "agent": "a1", "surface_id": "api-main" },
  { "agent": "a2", "surface_id": "auth-flow" },
  { "agent": "a3", "surface_id": "billing-api" }
]
```

Then the root calls:

```
bounty_start_wave({
  target_domain: "example.test",
  wave_number: 1,
  assignments: [...]
})
```

Code path:

```
mcp/lib/tools/start-wave.js
-> mcp/lib/waves.js
```

Artifacts written:

```
wave-1-assignments.json
state.json
pipeline-events.jsonl
```

Important `state.json` mutation:

```
{
  "hunt_wave": 0,
  "pending_wave": 1
}
```

Important `wave-1-assignments.json` detail:

```
{
  "wave_number": 1,
  "assignments": [
    {
      "agent": "a1",
      "surface_id": "api-main",
      "handoff_token_sha256": "..."
    }
  ]
}
```

The plaintext handoff token is returned only in the MCP response to `bounty_start_wave`. It is not written to disk. The root injects each token into only the matching hunter prompt.

Why this matters:

- an agent cannot write another agent's authoritative handoff without the token
- the merge can distinguish verified handoffs from legacy/unverified handoffs
- the orchestrator does not have to trust Markdown claims

23.6 Step 5: Hunters Run In Parallel

The root spawns one background `hunter-agent` per assignment:

```
hunter-w1-a1 -> api-main
hunter-w1-a2 -> auth-flow
hunter-w1-a3 -> billing-api
```

The root must use background tasks for hunter waves. It must also respect the launch-turn barrier:

```
start wave
spawn hunters
report assignments
stop this turn
```

It must not start and merge the same wave in one turn. This prevents the root from merging before background agents have had a fair chance to write handoffs.

23.7 Step 6: Each Hunter Loads One Curated Context Packet

The first action for a normal hunter is:

```
bounty_read_hunter_brief({
  target_domain: "example.test",
  wave: "w1",
  agent: "a2"
})
```

Code path:

```
mcp/lib/tools/read-hunter-brief.js
-> mcp/lib/hunter-brief.js
```

The brief contains only what that hunter needs:

- its assigned surface
- valid surface IDs
- relevant traffic summary
- relevant HTTP audit summary
- coverage summary
- dead-end and WAF exclusions
- auth guidance
- ranking hints
- public intel hints
- static scan hints
- curated technique hints

This is the practical core of context management. The hunter does not receive the whole session. It receives one bounded packet.

23.8 Step 7: Hunter Activity Creates Structured Evidence Trails

During testing, the hunter uses MCP tools instead of writing files:

```
bounty_http_scan          -> http-audit.jsonl
bounty_log_coverage       -> coverage.jsonl
bounty_log_dead_ends      -> live-dead-ends-w1-a2.jsonl
bounty_record_finding     -> findings.jsonl and findings.md
```

If a hunter proves a finding, it records it immediately:

```
bounty_record_finding({
  target_domain: "example.test",
  wave: "w1",
  agent: "a2",
  surface_id: "billing-api",
  title: "IDOR in invoice export exposes victim invoice metadata",
  severity: "high",
  endpoint: "https://app.example.test/api/invoices/export?id=123",
  validated: true,
  ...
})
```

Code path:

```
mcp/lib/tools/record-finding.js
-> mcp/lib/findings.js
```

Artifacts:

```
findings.jsonl
findings.md
pipeline-events.jsonl
```

`findings.jsonl` is authoritative. `findings.md` is a human/debug mirror.

23.9 Step 8: Hunter Completion Uses Two Signals

At the end of its assigned surface, each hunter must call:

```
bounty_write_wave_handoff({
  target_domain: "example.test",
  wave: "w1",
```

```

    agent: "a2",
    surface_id: "billing-api",
    surface_status: "complete",
    handoff_token: "...",
    summary: "...",
    chain_notes: [...],
    dead_ends: [...],
    lead_surface_ids: [...]
  })

```

Artifacts:

```

handoff-w1-a2.json
handoff-w1-a2.md

```

Then the final assistant message must include:

```
BOB_HUNTER_DONE {"target_domain":"example.test","wave":"w1","agent":"a2","surface_id":"billing-api"}
```

The hook checks the marker and the structured handoff:

```
.claude/hooks/hunter-subagent-stop.js
```

Why two signals exist:

- the MCP handoff is the durable state input
- the final marker helps Claude Code/hook lifecycle know the subagent ended correctly
- the hook prevents “I am done” prose from replacing structured completion

23.10 Step 9: Resume Or Completion Merges The Wave

When all hunters complete, or when the operator later runs resume, the root reconciles the pending wave:

```
bounty_read_state_summary({ target_domain: "example.test" })
```

If `pending_wave` is 1, the root calls:

```

bounty_apply_wave_merge({
  target_domain: "example.test",
  wave_number: 1,
  force_merge: false
})

```

Code path:

```

mcp/lib/tools/apply-wave-merge.js
-> mcp/lib/waves.js

```

If not all handoffs are present, MCP returns:

```

{
  "status": "pending",
  "readiness": {
    "assignments_total": 3,
    "handoffs_total": 2,
    "missing_agents": ["a3"],
    "is_complete": false
  }
}

```

No state mutation happens in that case.

If ready, the merge updates:

```

state.explored
state.dead_ends

```

```
state.waf_blocked_endpoints
state.lead_surface_ids
state.pending_wave
state.hunt_wave
state.total_findings
```

Typical mutation:

```
{
  "hunt_wave": 1,
  "pending_wave": null,
  "total_findings": 1,
  "explored": ["api-main", "auth-flow", "billing-api"]
}
```

The merge also computes requeues from:

- partial handoffs
- missing handoffs
- invalid handoffs
- coverage entries with unfinished statuses

23.11 Step 10: The Root Decides Another Wave Or CHAIN

The root calls:

```
bounty_wave_status({ target_domain: "example.test" })
```

That status includes:

- finding counts
- coverage percentage
- unexplored high/critical surfaces
- open requeue surfaces
- HTTP audit health
- traffic import summary
- circuit breaker summary
- transition blockers

The HUNT -> CHAIN gate is enforced in:

```
mcp/lib/phase-gates.js
```

The transition blocks if:

- `pending_wave` is still set
- `attack_surface.json` cannot be read
- coverage cannot be read
- high or critical surfaces remain unexplored
- latest coverage still has open promising, needs-auth, or requeue entries

If the gate passes:

```
bounty_transition_phase({
  target_domain: "example.test",
  to_phase: "CHAIN"
})
```

23.12 Step 11: Chain Builder Tests Combinations

The root spawns:

```
Agent(subagent_type: "chain-builder", name: "chain", ...)
```

The chain builder reads:


```
bounty_read_findings
bounty_read_wave_handoffs
bounty_read_http_audit
bounty_list_auth_profiles
bounty_read_chain_attempts
```

It writes:

```
bounty_write_chain_attempt -> chain-attempts.jsonl
```

The CHAIN -> VERIFY gate blocks when a chain is required but no terminal chain attempt exists. A chain is required when there are multiple findings or any structured handoff includes `chain_notes`.

Terminal outcomes:

```
confirmed
denied
blocked
not_applicable
```

Non-terminal:

```
inconclusive
```

The point of this phase is not to make every finding worse. It is to eliminate speculative chains and record whether a plausible chain actually worked.

23.13 Step 12: Three Verifiers Protect Against Agent Overclaiming

After CHAIN, the root transitions:

```
CHAIN -> VERIFY
```

Then it runs three verifier agents:

```
brutalist-verifier -> brutalist.json
balanced-verifier  -> balanced.json
final-verifier     -> verified-final.json
```

Round behavior:

```
brutalist: attack the claims hard
balanced: catch brutalist false negatives or over-downgrades
final: re-run only reportable survivors with fresh requests
```

Important invariant:

```
balanced must include every brutalist finding
final must include every balanced finding
```

If a finding is omitted from a verifier result array, it is effectively lost. That is why the prompts repeat the “include every prior finding” rule.

23.14 Step 13: Evidence Packs Are Written After Final Verification

If final verification has reportable findings, the root spawns:

```
Agent(subagent_type: "evidence-agent", name: "evidence", ...)
```

The evidence agent reads:

```
bounty_read_findings
bounty_read_verification_round(round="final")
bounty_read_http_audit
bounty_list_auth_profiles
```

It writes:

```
bounty_write_evidence_packs -> evidence-packs.json and evidence-packs.md
```

Evidence is intentionally late in the pipeline. This prevents agents from collecting polished evidence for claims that did not survive verification.

The VERIFY -> GRADE gate validates evidence packs for final reportable findings. If a reportable finding is missing evidence, the transition blocks.

If no final finding is reportable, evidence can be skipped in a structured way, and the run still proceeds to GRADE and REPORT.

23.15 Step 14: Grader Produces A Submit/Hold/Skip Decision

The root transitions:

```
VERIFY -> GRADE
```

Then it spawns:

```
Agent(subagent_type: "grader", name: "grader", ...)
```

The grader reads:

```
bounty_read_findings
bounty_read_chain_attempts
bounty_read_verification_round(round="final")
bounty_read_evidence_packs
```

It writes:

```
bounty_write_grade_verdict -> grade.json and grade.md
```

Verdicts:

```
SUBMIT: score >= 40 and at least one medium-or-higher final reportable
```

```
HOLD:   score 20-39
```

```
SKIP:   score < 20 or no medium-or-higher final reportable
```

On HOLD, the FSM can go:

```
GRADE -> HUNT
```

state.hold_count increments so Bob does not loop forever without escalation.

23.16 Step 15: Report Writer Produces The Human Output

For SUBMIT or SKIP, the root transitions:

```
GRADE -> REPORT
```

Then it spawns:

```
Agent(subagent_type: "report-writer", name: "reporter", ...)
```

The report writer reads:

```
bounty_read_findings
bounty_read_chain_attempts
bounty_read_verification_round(round="final")
bounty_read_evidence_packs
bounty_read_grade_verdict
```

It writes:

```
report.md
```

report.md is the thing a human reads or adapts. It should be explainable from the structured artifacts behind it. If report.md says something that verified-final.json, evidence-packs.json, and grade.json do not support, trust the JSON and debug the report writer.

23.17 Golden Run Reading Checklist

When following this run in the code, read in this order:

```
.claude/skills/bob-hunt/SKILL.md
mcp/lib/session-state.js
.claude/agents/recon-agent.md
mcp/lib/waves.js
.claude/agents/hunter-agent.md
mcp/lib/hunter-brief.js
mcp/lib/findings.js
mcp/lib/phase-gates.js
.claude/agents/chain-builder.md
.claude/agents/*verifier.md
mcp/lib/evidence.js
.claude/agents/grader.md
.claude/agents/report-writer.md
```

The goal is not to memorize every helper function. The goal is to see where each state mutation is allowed to happen.

24 Deep Artifact Catalog

This is the map of “what file means what” in a session directory.

The default session root is:

`~/bounty-agent-sessions/[domain]/`

Path construction lives in:

`mcp/lib/paths.js`

24.1 `state.json`

Owner:

`MCP only`

Written by:

`bounty_init_session`
`bounty_transition_phase`
`bounty_start_wave`
`bounty_apply_wave_merge`

Read by:

`bounty_read_session_state`
`bounty_read_state_summary`
`bounty_wave_status`
`bounty_read_pipeline_analytics`
`phase gates`

Use it to answer:

- what phase is the session in?
- is a wave pending?
- how many waves were merged?
- what surfaces are explored?
- how many findings does the state think exist?
- did grading hold once or more?
- is auth pending, authenticated, or unauthenticated?

Trust level:

`authoritative for phase and coarse progress`

Important caution:

`total_findings` is a summary copied into state during merges. The authoritative finding records are still in `findings.jsonl`.

24.2 `attack_surface.json`

Owner:

`recon-agent`

Written by:

`.claude/agents/recon-agent.md`

Read by:

`root orchestrator`
`bounty_start_wave`
`bounty_wave_status`

bounty_read_hunter_brief
ranking and phase gate code

Use it to answer:

- what surfaces exist?
- what surface IDs can be assigned?
- which surfaces are high or critical?
- what evidence led recon to classify each surface?
- what high-value flows and bug-class hints should hunters consider?

Trust level:

authoritative for surface inventory

Important caution:

Runtime ranking should not rewrite this file. Ranking is computed from current state, traffic, coverage, audit, public intel, and static hints.

24.3 Recon Scratch Files

Examples:

recon-tools.txt
subdomains.txt
live_hosts.txt
family_candidates.txt
family_live.txt
all_urls.txt
nuclei_results.txt
js_urls.txt
js_endpoints.txt
js_secrets.txt

Owner:

recon-agent

Use them to answer:

- where did `attack_surface.json` come from?
- did a recon tool fail or return nothing?
- did JavaScript extraction produce useful endpoint hints?

Trust level:

supporting evidence for recon, not the main orchestration API

24.4 auth.json

Owner:

MCP auth tools

Written by:

bounty_auth_store
bounty_auto_signup

Read by:

bounty_list_auth_profiles
bounty_http_scan
agents that need profile summaries

Use it to answer:

- which profiles exist?
- was signup automated, assisted, or manually stored?
- are attacker and victim profiles both available?

Trust level:

authoritative for stored auth profiles

Important caution:

Do not paste raw auth material into prompts or reports. Agents should pass `auth_profile` into `bounty_http_scan`.

24.5 `traffic.jsonl`

Owner:

MCP import tool

Written by:

`bounty_import_http_traffic`

Read by:

`bounty_read_hunter_brief`

`bounty_wave_status`

`bounty_read_pipeline_analytics`

Use it to answer:

- what real application routes were imported from Burp/HAR-style history?
- what authenticated routes should hunters prioritize?
- did imported traffic cover the assigned surface?

Trust level:

authoritative imported traffic summary source

Important caution:

It is JSONL because imported request histories can be large and append-like.

24.6 `http-audit.jsonl`

Owner:

`bounty_http_scan`

Read by:

`hunters`

`chain builder`

`verifiers`

`evidence agent`

`analytics`

`wave status`

Use it to answer:

- what did Bob actually request?
- which egress profile was used?
- did target-owned hosts repeatedly timeout or fail?
- what request refs support evidence?
- did geofence warnings appear?

Trust level:

authoritative metadata trail for Bob HTTP requests

Important caution:

It should be redacted and bounded. It is not a raw packet capture.

24.7 wave-N-assignments.json

Owner:

bounty_start_wave

Read by:

bounty_read_hunter_brief

bounty_write_wave_handoff

bounty_wave_handoff_status

bounty_apply_wave_merge

Use it to answer:

- which agent owned which surface?
- what wave number was active?
- were handoff tokens required?

Trust level:

authoritative wave assignment record

Important caution:

It stores token hashes, not plaintext tokens.

24.8 handoff-wN-aN.json

Owner:

bounty_write_wave_handoff

Read by:

bounty_read_wave_handoffs

bounty_wave_handoff_status

bounty_apply_wave_merge

chain builder

phase gates for chain-note detection

SubagentStop hook validation

Use it to answer:

- did the hunter finish?
- was the surface complete or partial?
- what should be queued?
- what dead ends and WAF blocks were discovered?
- what chain notes should the chain builder consider?
- was the handoff token verified?

Trust level:

authoritative hunter completion record

Important caution:

Markdown handoffs are ignored for machine merge. A Markdown-only handoff is not a completed wave handoff.

24.9 handoff-wN-aN.md

Owner:

bounty_write_wave_handoff

Use it to answer:

- what did the hunter say in human-friendly prose?

Trust level:

human/debug mirror only

Important caution:

Do not build automation on it.

24.10 live-dead-ends-wN-aN.jsonl

Owner:

bounty_log_dead_ends

Read by:

bounty_apply_wave_merge
bounty_read_hunter_brief
analytics

Use it to answer:

- did a hunter log dead ends before reaching final handoff?
- should later hunters avoid a path even if a handoff was partial?

Trust level:

authoritative live exclusion log

24.11 coverage.jsonl

Owner:

bounty_log_coverage

Read by:

bounty_read_hunter_brief
bounty_wave_status
phase gates
wave merge requeue logic
analytics

Use it to answer:

- which endpoint, bug class, and auth-profile combinations were tested?
- what is still promising?
- what needs auth?
- what should be requeued?

Trust level:

authoritative coverage trail

Important caution:

Coverage is not a finding. It is a work ledger.

24.12 findings.jsonl

Owner:

bounty_record_finding

Read by:

root status tools

chain builder

verifiers

evidence agent

grader

report writer

analytics

Use it to answer:

- what claims did hunters prove live?
- which wave, agent, and surface produced each claim?
- what endpoint and auth profile were involved?
- what evidence did the hunter provide before verification?

Trust level:

authoritative hunter finding ledger

Important caution:

A finding here is not automatically reportable. It must survive verification, evidence, and grading.

24.13 findings.md

Owner:

bounty_record_finding

Use it to answer:

- what is the readable finding summary?

Trust level:

human/debug mirror only

24.14 chain-attempts.jsonl

Owner:

bounty_write_chain_attempt

Read by:

chain builder

verifiers

grader

report writer

phase gates

analytics

Use it to answer:

- were chain hypotheses tested?
- which findings were linked?
- did the chain work, fail, get blocked, or not apply?
- should chain impact be included in the final report?

Trust level:

authoritative chain test ledger

Important caution:

Only confirmed chain attempts should be used as impact evidence in reports.

24.15 brutalist.json, balanced.json, verified-final.json

Owner:

bounty_write_verification_round

Read by:

root orchestrator

later verifiers

evidence agent

grader

report writer

analytics

Use them to answer:

- did the finding survive skeptical replay?
- was severity changed?
- is the finding reportable?
- why was the finding denied or downgraded?

Trust level:

authoritative verification records

Important caution:

verified-final.json is the final reportability source. Earlier rounds explain the debate, but final drives evidence and grade.

24.16 brutalist.md, balanced.md, verified-final.md

Owner:

bounty_write_verification_round

Trust level:

human/debug mirrors only

24.17 evidence-packs.json

Owner:

bounty_write_evidence_packs

Read by:

bounty_read_evidence_packs

grader

report writer

phase gates

analytics

Use it to answer:

- is every final reportable finding covered by bounded evidence?
- what representative samples support the report?
- what aggregate counts prove impact without dumping raw sensitive data?
- what redaction decisions were made?

Trust level:

`authoritative pre-grade evidence layer`

Important caution:

This is where proof becomes report-safe. It must not contain secrets, cookies, tokens, passwords, raw large responses, or unbounded PII.

24.18 evidence-packs.md

Owner:

`bounty_write_evidence_packs`

Trust level:

`human/debug mirror only`

24.19 grade.json

Owner:

`bounty_write_grade_verdict`

Read by:

`bounty_read_grade_verdict`

`report writer`

`analytics`

Use it to answer:

- should the result be submitted, held, or skipped?
- what score did each final survivor receive?
- what feedback should drive a HOLD loop?

Trust level:

`authoritative verdict`

Important caution:

The code validates scoring arithmetic and verdict thresholds. If the grade is malformed, `bounty_read_grade_verdict` should fail instead of silently trusting it.

24.20 grade.md

Owner:

`bounty_write_grade_verdict`

Trust level:

`human/debug mirror only`

24.21 report.md

Owner:

`report-writer agent`

Read by:

`human operator`

`status/debug tools`

Use it to answer:

- what should be submitted or retained as the final closeout?

Trust level:

human output, but not the source of truth for orchestration

Important caution:

For a disputed claim, trace backward to `grade.json`, `evidence-packs.json`, `verified-final.json`, and `findings.jsonl`.

24.22 pipeline-events.jsonl

Owner:

MCP state-changing operations

Read by:

bounty_read_pipeline_analytics
/bob-status
/bob-debug

Use it to answer:

- what happened when?
- did a wave start or merge?
- did a phase transition happen?
- were there pending merges or failures?
- how did this session reach its current state?

Trust level:

authoritative event timeline, not authoritative current state

24.23 Tool Telemetry

Owner:

mcp/lib/tool-telemetry.js

Read by:

bounty_read_tool_telemetry
bounty_read_pipeline_analytics

Use it to answer:

- which MCP tools fail often?
- which agents hit stop-hook failures?
- are failures concentrated around auth, HTTP, handoff, evidence, or phase gates?

Trust level:

metadata-only diagnostic source

24.24 Static Artifact Files

Artifacts:

static-imports/SA-N.txt
static-artifacts.jsonl
static-scan-results.jsonl

Owner:

bounty_import_static_artifact
bounty_static_scan

Use them to answer:

- what token-contract or static source was imported?
- what bounded static scan hints should hunter briefs include?

Trust level:

authoritative for imported static-analysis leads

Important caution:

Hunters must import pasted content through MCP. They must not scan arbitrary local paths.

24.25 public-intel.json

Owner:

bounty_public_intel

Read by:

bounty_read_hunter_brief

ranking

analytics

Use it to answer:

- what public program or disclosure hints were collected?
- which bug classes or flows may be worth prioritizing?

Trust level:

prioritization hint, not evidence

25 Agent Responsibility Matrix

This section describes each agent as an operating contract: what it receives, what it may write, and what it must not decide.

25.1 Root Orchestrator

File:

`.claude/skills/bob-hunt/SKILL.md`

Phase:

all phases

Primary context:

- command arguments
- compact MCP state
- wave status
- pipeline analytics
- artifact existence and validation results

May write through MCP:

- session state
- wave assignments
- phase transitions
- session handoff notes

May spawn:

- recon agent
- hunter agents
- chain builder
- verifiers
- evidence agent
- grader
- report writer

Must not:

- hunt directly
- send target HTTP requests except AUTH signup/login calls
- synthesize handoff JSON
- manually edit MCP-owned artifacts
- merge a wave in the same turn it launched hunters

Best file to read:

`.claude/skills/bob-hunt/SKILL.md`

Failure signals:

- invalid phase transition
- pending wave not reconciled
- phase gate blockers
- missing attack surface
- missing verification/evidence/grade artifacts

25.2 Recon Agent

File:

`.claude/agents/recon-agent.md`

Phase:

RECON

Primary context:

- domain
- session directory

May write:

`attack_surface.json`
`recon scratch files`

Must not:

- use MCP tools
- keep probing beyond the seven Bash collection steps
- create multiple competing attack-surface artifacts

Best code/doc pair:

`.claude/agents/recon-agent.md`
`mcp/lib/attack-surface.js`

Failure signals:

- missing tools in `recon-tools.txt`
- empty `live_hosts.txt`
- missing or malformed `attack_surface.json`
- attack surface contains no surfaces

25.3 Hunter Agent

File:

`.claude/agents/hunter-agent.md`

Phase:

HUNT or EXPLORE

Primary context:

- spawn prompt with target, wave, agent, handoff token, egress profile
- one `bounty_read_hunter_brief` result

May write through MCP:

`http-audit.jsonl`
`coverage.jsonl`
`live-dead-ends-wN-aN.jsonl`
`findings.jsonl`
`findings.md`
`handoff-wN-aN.json`
`handoff-wN-aN.md`
`static artifacts and scan results`

Must not:

- test unassigned first-party surfaces as primary scope
- manually write handoff, findings, coverage, traffic, audit, or static files
- record weak standalone non-findings
- continue hammering unreachable or WAF-blocked hosts
- finish without a structured handoff in normal wave mode

Best code/doc pair:

.claude/agents/hunter-agent.md
mcp/lib/hunter-brief.js
mcp/lib/waves.js
mcp/lib/findings.js

Failure signals:

- no BOB_HUNTER_DONE marker
- marker does not match handoff
- handoff token invalid
- handoff missing or malformed
- `surface_status` is partial and should requeue
- coverage shows promising work left open

25.4 Chain Builder

File:

.claude/agents/chain-builder.md

Phase:

CHAIN

Primary context:

- findings
- structured handoffs
- HTTP audit
- auth profile summaries
- prior chain attempts

May write through MCP:

chain-attempts.jsonl

Must not:

- invent chains from prose alone
- read Markdown handoffs or `findings.md` as machine input
- leave required chains as only `inconclusive`

Best code/doc pair:

.claude/agents/chain-builder.md
mcp/lib/chain-attempts.js
mcp/lib/phase-gates.js

Failure signals:

- CHAIN -> VERIFY blocked by missing terminal chain attempt
- chain evidence says “could” but no `confirmed`, `denied`, `blocked`, or `not_applicable` outcome exists

25.5 Brutalist Verifier

File:

.claude/agents/brutalist-verifier.md

Phase:

VERIFY round 1

Primary context:

- findings
- chain attempts

- HTTP audit
- auth profile summaries

May write through MCP:

`brutalist.json`
`brutalist.md`

Must not:

- accept claims without replay or reasoning
- write verifier Markdown directly
- deny solely because auth expired

Failure signals:

- missing `brutalist.json`
- empty results when findings exist
- malformed disposition/severity/reportable fields

25.6 Balanced Verifier

File:

`.claude/agents/balanced-verifier.md`

Phase:

VERIFY round 2

Primary context:

- findings
- brutalist round
- chain attempts
- HTTP audit
- auth profile summaries

May write through MCP:

`balanced.json`
`balanced.md`

Must not:

- drop findings from the brutalist round
- blindly pass through denied/downgraded results if there is a clear false negative

Failure signals:

- result count is less than brutalist result count
- a brutalist finding ID is absent from balanced results

25.7 Final Verifier

File:

`.claude/agents/final-verifier.md`

Phase:

VERIFY round 3

Primary context:

- findings
- balanced round
- chain attempts

- HTTP audit
- auth profile summaries

May write through MCP:

`verified-final.json`
`verified-final.md`

Must not:

- drop findings from the balanced round
- use stale results for reportable survivors without fresh replay

Failure signals:

- reportable finding lacks fresh replay reasoning
- final results omit a balanced finding
- evidence phase has nothing to cover because final incorrectly dropped all reportables

25.8 Evidence Agent

File:

`.claude/agents/evidence-agent.md`

Phase:

`after final verification, before GRADE`

Primary context:

- findings
- final verification
- HTTP audit
- auth profile summaries

May write through MCP:

`evidence-packs.json`
`evidence-packs.md`

Must not:

- create, modify, or remove findings
- grade
- write reports
- store raw secrets, cookies, tokens, or unbounded sensitive data

Failure signals:

- missing pack for a final reportable finding
- unsafe evidence content
- sample count too high
- evidence pack references a non-reportable finding

25.9 Grader

File:

`.claude/agents/grader.md`

Phase:

`GRADE`

Primary context:

- findings

- chain attempts
- final verification
- evidence packs

May write through MCP:

`grade.json`
`grade.md`

Must not:

- submit automatically
- score speculative chain impact
- issue **SUBMIT** without a medium-or-higher final reportable finding

Failure signals:

- arithmetic does not add up
- verdict contradicts thresholds
- **SUBMIT** without evidence or final reportable medium-or-higher finding

25.10 Report Writer

File:

`.claude/agents/report-writer.md`

Phase:

REPORT

Primary context:

- findings
- chain attempts
- final verification
- evidence packs
- grade verdict

May write directly:

`report.md`

Must not:

- report denied, blocked, inconclusive, or not-applicable chains as impact
- use hunter severity if verification changed it
- invent vulnerability sections in no-findings reports
- include methodology filler

Failure signals:

- report contains a claim absent from final verification
- report includes evidence absent from evidence packs
- report severity differs from final verification without explanation

26 State Machine And Phase Gates

The FSM is simple on paper:

```
RECON -> AUTH -> HUNT -> CHAIN -> VERIFY -> GRADE -> REPORT
                                         |
                                         v
                                REPORT -> EXPLORE -> CHAIN
```

But the important logic is not the arrow. The important logic is what must be true before an arrow is allowed.

Code:

```
mcp/lib/session-state.js
mcp/lib/phase-gates.js
```

26.1 Transition Contract

Every transition goes through:

bounty_transition_phase

That tool:

1. reads `state.json`
2. checks the current phase
3. checks the requested next phase is legal
4. runs any phase-specific gate
5. writes `state.json`
6. appends a `phase_transitioned` event

If the gate blocks, the tool returns a `STATE_CONFLICT` style failure. The root should use the blocker message to decide what to do next.

26.2 Gate Table

RECON -> AUTH

Legal after recon if `attack_surface.json` exists and root decides recon found surfaces.
The MCP transition itself does not create `attack_surface.json`.

AUTH -> HUNT

Requires `auth_status`.
Allowed values: `authenticated` or `unauthenticated`.

HUNT -> CHAIN

Requires no `pending_wave`.
Requires readable `attack_surface.json`.
Requires readable coverage.
Blocks if high/critical surfaces remain unexplored.
Blocks if latest coverage contains `promising`, `needs_auth`, or `requeue work`.
Can be overridden only with an explicit `override_reason`.

CHAIN -> VERIFY

Computes whether chain testing is required.
Chain testing is required when there are multiple findings or structured handoff `chain_notes`.
If required, at least one terminal chain attempt must exist.
Can be overridden only with an explicit `override_reason`.

VERIFY -> GRADE

Requires valid evidence packs for all final reportable findings.

If there are no final reportable findings, structured evidence skip is allowed.

GRADE -> REPORT

Reuses the evidence-pack validity gate.

GRADE -> HUNT

Allowed on HOLD.

Increments hold_count.

REPORT -> EXPLORE

Allowed only when the user asks to continue hunting after a report.

EXPLORE -> CHAIN

Uses the same wave system as HUNT, then returns to CHAIN.

26.3 Phase Gate Debugging Question

When a phase transition fails, ask:

Is this blocked by state, artifact validity, missing work, or operator intent?

Examples:

pending_wave is not null

-> state says a wave is open

-> inspect wave assignments and handoff status

unexplored_high_surfaces

-> attack_surface.json has HIGH/CRITICAL surfaces not in state.explored

-> run another wave or explicitly override with a reason

open_requeue_coverage

-> latest coverage has promising, needs_auth, or requeue entries

-> run another wave focused on those surfaces

chain_attempts_missing

-> chain context says a chain should be tested

-> rerun chain builder or override only with operator acceptance

evidence_packs_invalid

-> final reportable findings are not all covered by valid evidence

-> rerun evidence agent

26.4 State Machine Reading Exercise

Open:

mcp/lib/session-state.js

Find:

allowedTransitions

Then open:

mcp/lib/phase-gates.js

Trace these functions:

computeHuntToChainGate

computeChainToVerifyGate

```
computeVerifyToGradeGate  
formatTransitionBlockers
```

That is the complete “why can or cannot we move forward” core.

27 Trace One Finding End To End

This chapter traces a fictional finding F-1 forward and backward.

Finding:

F-1: IDOR in invoice export exposes victim invoice metadata

27.1 Forward Trace

27.1.1 1. Hunter Discovers The Issue

The hunter is assigned:

```
wave = w1
agent = a2
surface_id = billing-api
```

It reads:

```
bounty_read_hunter_brief
```

It sees imported traffic for:

```
GET /api/invoices/export?id=123
```

It tests with:

```
bounty_http_scan({
  target_domain: "example.test",
  method: "GET",
  url: "https://app.example.test/api/invoices/export?id=123",
  auth_profile: "attacker",
  wave: "w1",
  agent: "a2",
  surface_id: "billing-api",
  egress_profile: "default"
})
```

The request is audited in:

```
http-audit.jsonl
```

27.1.2 2. Hunter Logs Coverage

The hunter records that this endpoint and bug class were tested:

```
bounty_log_coverage({
  target_domain: "example.test",
  wave: "w1",
  agent: "a2",
  surface_id: "billing-api",
  entries: [
    {
      "endpoint": "/api/invoices/export",
      "method": "GET",
      "bug_class": "IDOR",
      "auth_profile": "attacker",
      "status": "tested",
      "evidence_summary": "Export accepted arbitrary invoice id"
    }
  ]
})
```

Artifact:

coverage.jsonl

27.1.3 3. Hunter Records The Finding

Tool:

bounty_record_finding

Artifact:

findings.jsonl

Simplified line:

```
{
  "id": "F-1",
  "target_domain": "example.test",
  "wave": "w1",
  "agent": "a2",
  "surface_id": "billing-api",
  "title": "IDOR in invoice export exposes victim invoice metadata",
  "severity": "high",
  "endpoint": "https://app.example.test/api/invoices/export?id=123",
  "auth_profile": "attacker",
  "validated": true
}
```

MCP also updates the human mirror:

findings.md

27.1.4 4. Hunter Writes Handoff

Tool:

bounty_write_wave_handoff

Artifact:

handoff-w1-a2.json

Important fields:

```
{
  "wave": "w1",
  "agent": "a2",
  "surface_id": "billing-api",
  "surface_status": "complete",
  "provenance": "verified",
  "summary": "Tested invoice export and related billing endpoints.",
  "chain_notes": [
    "Invoice IDOR may combine with account enumeration from auth-flow."
  ]
}
```

Then the final marker appears:

BOB_HUNTER_DONE {"target_domain":"example.test","wave":"w1","agent":"a2","surface_id":"billing-api"}

The hook checks marker-to-handoff consistency.

27.1.5 5. Wave Merge Updates State

Tool:

bounty_apply_wave_merge

State after merge:

```
{
  "pending_wave": null,
  "hunt_wave": 1,
  "total_findings": 1,
  "explored": ["billing-api"]
}
```

Pipeline event:

wave_merged

27.1.6 6. Chain Builder Tests The Chain Note

Because the handoff includes `chain_notes`, CHAIN likely requires a terminal attempt.

Tool:

bounty_write_chain_attempt

Artifact:

chain-attempts.jsonl

Possible record:

```
{
  "attempt_id": "C-1",
  "finding_ids": ["F-1"],
  "surface_ids": ["billing-api", "auth-flow"],
  "hypothesis": "Account enumeration can help target invoice IDs for export.",
  "outcome": "denied",
  "evidence_summary": "Enumeration did not reveal invoice IDs.",
  "auth_profiles": ["attacker", "victim"]
}
```

Even a denied attempt is useful. It prevents speculative chain language in the report.

27.1.7 7. Verification Rounds Debate The Finding

Round 1:

brutalist.json

Maybe:

```
{
  "finding_id": "F-1",
  "disposition": "confirmed",
  "severity": "high",
  "reportable": true,
  "reasoning": "Fresh replay with attacker profile returned victim invoice metadata."
}
```

Round 2:

balanced.json

Passes through or adjusts.

Round 3:

verified-final.json

Final source for whether evidence must be collected:

```
{
  "finding_id": "F-1",
  "disposition": "confirmed",
  "severity": "high",
  "reportable": true,
  "reasoning": "Fresh final replay confirms cross-account invoice metadata exposure."
}
```

27.1.8 8. Evidence Agent Creates A Safe Proof Pack

Tool:

bounty_write_evidence_packs

Artifact:

evidence-packs.json

Simplified pack:

```
{
  "finding_id": "F-1",
  "sample_type": "cross-account invoice export",
  "sample_count": 3,
  "representative_samples": [
    {
      "request_ref": "http-audit:42",
      "endpoint": "/api/invoices/export",
      "auth_profile": "attacker",
      "status": 200,
      "observed_fields": ["account_id", "email", "invoice_total"],
      "redacted_object_id": "inv_...789"
    }
  ],
  "sensitive_clusters": ["invoice metadata"],
  "replay_summary": "Attacker profile returned victim invoice metadata.",
  "redaction_notes": "Identifiers and personal values redacted.",
  "report_snippet": "An attacker can export invoice metadata for other accounts by changing the invoice id."
}
```

27.1.9 9. Grader Scores It

Tool:

bounty_write_grade_verdict

Artifact:

grade.json

Possible verdict:

```
{
  "verdict": "SUBMIT",
  "total_score": 72,
  "findings": [
    {
      "finding_id": "F-1",
      "impact": 25,
      "proof_quality": 20,
      "severity_accuracy": 12,
      "chain_potential": 5,
      "report_quality": 10,
    }
  ]
}
```

```

    "total_score": 72
  }
]
}

```

27.1.10 10. Report Writer Uses Only Survivors

Artifact:

report.md

The report should cite:

- final severity from `verified-final.json`
- bounded samples from `evidence-packs.json`
- only confirmed chain attempts from `chain-attempts.jsonl`
- grade verdict from `grade.json`

27.2 Backward Trace

If you are reading `report.md` and want to verify a claim, walk backward:

report.md

```

-> grade.json
-> evidence-packs.json
-> verified-final.json
-> balanced.json
-> brutalist.json
-> findings.jsonl
-> http-audit.jsonl
-> handoff-w1-a2.json
-> wave-1-assignments.json
-> attack_surface.json
-> state.json

```

Ask these questions:

1. Does `grade.json` include the finding?
2. Does `verified-final.json` mark it `reportable: true`?
3. Does `evidence-packs.json` include a pack for it?
4. Does the evidence pack reference request audit entries?
5. Does `findings.jsonl` show which hunter recorded it?
6. Does the hunter handoff match the assigned surface?
7. Does state show the wave was actually merged?

That backward trace is the fastest way to find hallucinated report content.

28 Debugging Playbook

This section is written for offline use. Start with the symptom, then inspect the listed artifacts or files.

28.1 The Run Says A Wave Is Pending

Inspect:

```
state.json
wave-N-assignments.json
handoff-wN-aN.json files
bounty_wave_handoff_status
bounty_read_pipeline_analytics
```

Likely causes:

- one background hunter has not finished
- a hunter ended without the final marker
- handoff JSON is missing
- handoff JSON is malformed
- marker and handoff disagree
- an unexpected agent wrote a handoff

Code to read:

```
mcp/lib/waves.js
.claude/hooks/hunter-subagent-stop.js
```

What to do:

- wait for hunters if they are still running
- resume the session after completion
- use force-merge only when the missing/invalid handoff is understood and the reason is explicit

28.2 HUNT Will Not Transition To CHAIN

Inspect:

```
bounty_wave_status
state.json
attack_surface.json
coverage.jsonl
```

Likely blockers:

```
pending_wave
attack_surface_unavailable
coverage_unavailable
unexplored_high_surfaces
open_requeue_coverage
```

Code to read:

```
mcp/lib/phase-gates.js
computeHuntToChainGate
```

What to do:

- merge the pending wave
- run another wave for high/critical unexplored surfaces
- run another wave for promising or requeue coverage
- override only if the operator intentionally accepts the gap

28.3 CHAIN Will Not Transition To VERIFY

Inspect:

```
findings.jsonl
handoff-wN-aN.json
chain-attempts.jsonl
```

Likely blocker:

```
chain_attempts_missing
```

This means the repo detected reason to test chainability, usually multiple findings or handoff `chain_notes`, but no terminal chain attempt was recorded.

Code to read:

```
mcp/lib/phase-gates.js
computeChainRequirement
computeChainToVerifyGate
mcp/lib/chain-attempts.js
```

What to do:

- rerun the chain builder with the blocker text
- make sure it records `confirmed`, `denied`, `blocked`, or `not_applicable`
- do not treat `inconclusive` as satisfying the gate

28.4 VERIFY Will Not Transition To GRADE

Inspect:

```
verified-final.json
evidence-packs.json
bounty_read_evidence_packs
```

Likely blocker:

```
evidence_packs_invalid
```

Common causes:

- final reportable finding has no pack
- evidence pack references a non-reportable finding
- evidence contains unsafe token-like or secret-like content
- representative samples are missing or unbounded

Code to read:

```
mcp/lib/evidence.js
mcp/lib/phase-gates.js
```

What to do:

- rerun evidence agent
- remove unsafe fields
- ensure every final reportable finding has exactly the needed bounded proof

28.5 The Report Mentions A Finding That Is Not In Final Verification

Inspect:

```
report.md
verified-final.json
evidence-packs.json
grade.json
```

Likely causes:

- report writer used stale hunter findings instead of final verification
- report writer included a denied or downgraded non-reportable finding
- report writer included chain impact from a denied or blocked chain attempt

Code/doc to read:

`.claude/agents/report-writer.md`

What to do:

- regenerate the report writer output
- instruct it to use only final reportable findings and evidence packs

28.6 A Hunter Keeps Repeating Work

Inspect:

```
coverage.jsonl
bounty_read_hunter_brief
dead_ends in state.json
waf_blocked_endpoints in state.json
```

Likely causes:

- coverage was not logged before pivots
- dead ends were not merged yet
- hunter brief caps hid some older context
- assigned surface is too broad

Code to read:

```
mcp/lib/hunter-brief.js
mcp/lib/coverage.js
```

What to do:

- make hunters log coverage after meaningful tests
- split broad surfaces in recon if needed
- tune hunter brief summaries if context caps are too aggressive

28.7 A Finding Is Missing From The Report

Trace:

```
findings.jsonl
brutalist.json
balanced.json
verified-final.json
evidence-packs.json
grade.json
report.md
```

Possible disappearance points:

- hunter never recorded it
- brutalist denied it
- balanced failed to pass it through
- final failed to pass it through
- final marked it non-reportable
- evidence pack missing
- grade skipped it
- report writer omitted it

Most important invariant:

balanced includes every brutalist result

final includes every balanced result

If that invariant breaks, inspect:

```
mcp/lib/findings.js
.claude/agents/balanced-verifier.md
.claude/agents/final-verifier.md
test/mcp-server.test.js
```

28.8 Auth Does Not Seem To Work

Inspect:

```
auth.json
bounty_list_auth_profiles
http-audit.jsonl
```

Likely causes:

- profile was never stored
- profile expired
- profile name mismatch
- request did not pass `auth_profile`
- target changed auth requirements

What to do:

- use `bounty_list_auth_profiles` before replay
- store a fresh profile with `bounty_auth_store`
- make sure hunters/verifiers use `auth_profile`: "attacker" and "victim" where appropriate

28.9 HTTP Requests Fail Repeatedly

Inspect:

```
http-audit.jsonl
bounty_wave_status
bounty_read_pipeline_analytics
circuit_breaker_summary in hunter brief
```

Likely causes:

- target unavailable
- WAF or rate limit
- egress profile blocked
- geofence or regional behavior
- internal host reachability issue

What to do:

- do not silently rotate egress
- log blocked coverage and dead ends
- ask the operator to resume with a specific `--egress` profile if appropriate

28.10 JSON And Markdown Disagree

Rule:

JSON and JSONL win for orchestration.

Markdown is human/debug output unless explicitly documented otherwise.

Examples:

findings.jsonl wins over findings.md
handoff-wN-aN.json wins over handoff-wN-aN.md
verified-final.json wins over verified-final.md
evidence-packs.json wins over evidence-packs.md
grade.json wins over grade.md

Exception:

report.md is the final human-facing report.

Even then, report trust comes from the structured artifacts behind it.

29 Context Budget Cheat Sheet

Bob's architecture is mostly about preventing context collapse.

The naive version of this project would paste everything into every agent:

```
all recon
all traffic
all requests
all findings
all handoffs
all previous discussion
all reports
```

That would fail because agents would lose the important part, repeat work, and silently drift away from the state machine.

Bob uses three context patterns instead.

29.1 Pattern 1: Compact State For The Root

The root usually reads:

```
bounty_read_state_summary
bounty_wave_status
bounty_read_pipeline_analytics
```

It does not need every request or every endpoint. It needs to know:

- what phase is active
- whether a wave is pending
- whether gates are blocked
- whether another agent must run
- whether an artifact is missing

This is why `readStateSummary()` exists next to full `readSessionState()`.

29.2 Pattern 2: One Assigned Brief For Each Hunter

Each hunter reads:

```
bounty_read_hunter_brief
```

That brief is a lossy but purposeful packet. It should answer:

- what am I assigned?
- what should I prioritize?
- what has already been tested?
- what should I avoid?
- what auth can I use?
- what request history is relevant?
- what static or public hints matter?

The hunter should not read the whole session directory. It should not decide its own surface. It should not consume all other hunters' context.

29.3 Pattern 3: Structured Artifacts For Later Agents

Later agents read JSON/JSONL:

```
chain builder -> findings, handoffs, audit, auth summaries
verifiers     -> findings, chain attempts, audit, auth summaries
evidence      -> final verification, findings, audit, auth summaries
```

```
grader          -> final verification, evidence packs, chain attempts
reporter        -> final verification, evidence packs, grade
```

This prevents the final report from depending on the root chat transcript.

29.4 What Gets Summarized

Good candidates for summary:

- old HTTP audit entries
- imported traffic
- coverage history
- dead ends
- public intel
- static scan hints
- telemetry

Bad candidates for lossy summary:

- final verification dispositions
- grade verdict
- evidence pack validation
- phase
- pending wave
- handoff token verification result

The rule of thumb:

```
Summarize prioritization context.
Preserve decision state.
```

29.5 What Must Stay Structured

These should remain structured and machine-validated:

```
state
wave assignments
handoffs
coverage records
findings
chain attempts
verification rounds
evidence packs
grade
pipeline events
tool telemetry
```

If these become prose-only, many-agent coordination breaks.

29.6 Context Loss Recovery

If the root chat loses context, the recovery path is:

```
/bob-hunt resume example.test
```

The root does not need the old chat. It reads:

```
bounty_read_state_summary
```

Then:

```
if pending_wave:
    try bounty_apply_wave_merge
```

```
else:  
    continue from state.phase
```

That is why the durable state model matters. Resume is possible because the chat is disposable.

30 Offline Exercises

These are designed for a flight. Do the “Task” first, then check the answer.

30.1 Exercise 1: Find The Main Orchestrator Contract

Task:

```
sed -n '1,260p' .claude/skills/bob-hunt/SKILL.md
```

Questions:

- what tools can the root call?
- what are the hard rules?
- where is the FSM defined?
- what does the launch-turn barrier forbid?

Answer:

The root orchestrator contract is the `bob-hunt` skill. It owns phase coordination, agent spawning, wave start/merge decisions, and resume behavior. It must not hunt directly, synthesize handoffs, or write MCP-owned artifacts. The launch-turn barrier forbids merging or checking handoff status in the same turn that spawned hunters.

30.2 Exercise 2: Find Who Writes `state.json`

Task:

```
rg -n "state.json|writeSessionStateDocument|session_artifacts_written" mcp/lib
```

Answer:

`state.json` is written by MCP state/wave tools:

```
bounty_init_session
bounty_transition_phase
bounty_start_wave
bounty_apply_wave_merge
```

The core implementation is in:

```
mcp/lib/session-state.js
mcp/lib/waves.js
```

Agents should not write it directly.

30.3 Exercise 3: Find The Wave Token Logic

Task:

```
rg -n "handoff_token|sha256|provenance" mcp/lib/waves.js
```

Answer:

`bounty_start_wave` generates a plaintext token per assignment and writes only its SHA-256 hash to `wave-N-assignments.json`. `bounty_write_wave_handoff` validates the supplied token. Verified handoffs get `provenance: "verified"`.

This prevents unauthenticated handoff spoofing between agents.

30.4 Exercise 4: Find The Hunter’s First Required Action

Task:

```
rg -n "first action|bounty_read_hunter_brief|Call `bounty_read_hunter_brief`" .claude/agents/hunter-agent.m
```

Answer:

The hunter must call `bounty_read_hunter_brief` first in normal wave mode. The orchestrator prompt injects wave, agent, target, token, and egress profile, but the brief is where the assigned context packet comes from.

30.5 Exercise 5: Find Where HUNT -> CHAIN Can Block

Task:

```
sed -n '1,180p' mcp/lib/phase-gates.js
```

Answer:

`computeHuntToChainGate()` blocks on:

```
pending_wave
attack_surface_unavailable
coverage_unavailable
unexplored_high_surfaces
open_requeue_coverage
```

This is why Bob cannot simply decide “enough hunting” based on chat vibes.

30.6 Exercise 6: Find The “Include Every Finding” Rule

Task:

```
rg -n "EVERY finding|silently dropped|missing from your results" .claude/agents
```

Answer:

Balanced verifier must include every brutalist result. Final verifier must include every balanced result. If a verifier omits a finding, it can disappear from the pipeline.

30.7 Exercise 7: Find Evidence Validation

Task:

```
rg -n "requireValidEvidence|representative_samples|secret|token|final reportable" mcp/lib/evidence.js
```

Answer:

Evidence validation ensures final reportable findings are covered and evidence is bounded/redacted. This is the gate before grade/report work.

30.8 Exercise 8: Find The Tool Registry Contract

Task:

```
sed -n '1,180p' mcp/lib/tool-registry.js
```

Answer:

Every tool entry must declare metadata such as:

```
role_bundles
mutating
network_access
browser_access
scope_required
sensitive_output
session_artifacts_written
hook_required
```

This supports tool governance, prompt generation, and contract tests.

30.9 Exercise 9: Find Write Guard Rules

Task:

```
sed -n '1,180p' .claude/hooks/session-write-guard.sh
sed -n '1,180p' test/test-write-guard.py
```

Answer:

The write guard blocks direct writes to MCP-owned artifacts such as state, findings, handoffs, evidence packs, traffic, audit, coverage, and pipeline events. It allows recon-owned `attack_surface.json`.

30.10 Exercise 10: Trace One Tool End To End

Task:

Trace `bounty_write_wave_handoff`.

Read:

```
mcp/lib/tools/write-wave-handoff.js
mcp/lib/tool-registry.js
mcp/lib/dispatch.js
mcp/lib/waves.js
.claude/agents/hunter-agent.md
.claude/hooks/hunter-subagent-stop.js
test/mcp-server.test.js
```

Answer:

The agent prompt requires the handoff. The tool module defines the schema and metadata. The registry validates tool shape. Dispatch wraps execution in the standard envelope. `waves.js` validates assignment, token, payload, and writes JSON/Markdown. The stop hook checks that the final marker matches the structured handoff. Tests cover missing, malformed, and mismatched cases.

31 Code Map By Question

Use this section when you know the question but not the file.

31.1 Where Does A Run Start?

```
.claude/skills/bob-hunt/SKILL.md  
mcp/lib/tools/init-session.js  
mcp/lib/session-state.js
```

31.2 Where Are Phases Defined?

```
mcp/lib/constants.js  
mcp/lib/session-state.js  
mcp/lib/phase-gates.js
```

31.3 Where Are MCP Tools Registered?

```
mcp/lib/tools/index.js  
mcp/lib/tool-registry.js  
mcp/lib/dispatch.js  
mcp/server.js
```

31.4 Where Is The Standard MCP Response Shape?

```
mcp/lib/envelope.js  
mcp/lib/dispatch.js
```

Look for:

```
{ ok, data, meta }  
{ ok: false, error, meta }
```

31.5 Where Are Session Paths Defined?

```
mcp/lib/paths.js
```

31.6 Where Is State Persisted And Normalized?

```
mcp/lib/session-state.js  
mcp/lib/storage.js
```

31.7 Where Are Waves Implemented?

```
mcp/lib/waves.js  
mcp/lib/assignments.js  
mcp/lib/tools/start-wave.js  
mcp/lib/tools/apply-wave-merge.js  
mcp/lib/tools/write-wave-handoff.js
```

31.8 Where Are Hunter Briefs Built?

```
mcp/lib/hunter-brief.js  
mcp/lib/ranking.js  
mcp/lib/http-records.js  
mcp/lib/coverage.js
```

31.9 Where Are Findings Implemented?

mcp/lib/findings.js
mcp/lib/tools/record-finding.js
mcp/lib/tools/read-findings.js
mcp/lib/tools/list-findings.js

31.10 Where Are Chain Attempts Implemented?

mcp/lib/chain-attempts.js
mcp/lib/tools/write-chain-attempt.js
mcp/lib/tools/read-chain-attempts.js

31.11 Where Are Verification Rounds Implemented?

mcp/lib/findings.js
mcp/lib/tools/write-verification-round.js
mcp/lib/tools/read-verification-round.js

31.12 Where Are Evidence Packs Implemented?

mcp/lib/evidence.js
mcp/lib/tools/write-evidence-packs.js
mcp/lib/tools/read-evidence-packs.js

31.13 Where Is Grading Implemented?

mcp/lib/tools/write-grade-verdict.js
mcp/lib/tools/read-grade-verdict.js
.claude/agents/grader.md

31.14 Where Is HTTP Scanning And Audit Implemented?

mcp/lib/http-scan.js
mcp/lib/safe-fetch.js
mcp/lib/http-records.js
mcp/lib/tools/http-scan.js
mcp/redaction.js

31.15 Where Is Auth Stored And Applied?

mcp/lib/auth.js
mcp/lib/signup.js
mcp/auto-signup.js
mcp/lib/tools/auth-store.js
mcp/lib/tools/auto-signup.js
mcp/lib/tools/list-auth-profiles.js

31.16 Where Are Egress Profiles Implemented?

mcp/lib/egress-profiles.js
.claude/commands/bob-egress.md
.claude/hooks/bob-egress.js
.claude/bob/egress-profiles.json

31.17 Where Are Analytics Implemented?

mcp/lib/pipeline-analytics.js
mcp/lib/tool-telemetry.js

`.claude/skills/bob-status/SKILL.md`
`.claude/skills/bob-debug/SKILL.md`

31.18 Where Are Safety Hooks?

`.claude/hooks/session-write-guard.sh`
`.claude/hooks/hunter-subagent-stop.js`
`.claude/hooks/scope-guard.sh`
`.claude/hooks/scope-guard-mcp.sh`

31.19 Where Are Prompt Contracts Tested?

`test/prompt-contracts.test.js`
`scripts/generate-agent-tools.js`
`scripts/generate-bountyagent-skill.js`

31.20 Where Are End-To-End Runtime Behaviors Tested?

`test/mcp-server.test.js`
`test/cli.test.js`
`test/install-smoke.test.js`
`test/policy-replay.test.js`

32 Improvement Roadmap At A Glance

The detailed improvement notes below explain each suggestion. This roadmap groups them by likely payoff and implementation shape.

32.1 Quick Wins

Add a question-based code map to the repo docs.

Why:

New readers search by question, not by module name. This guide now includes that map, but a shorter checked-in docs page would help contributors.

Add an artifact ownership table to `README.md` or `docs/developer`.

Why:

The JSON/JSONL/Markdown split is central. A single table reduces accidental direct writes and prompt drift.

Add report provenance comments to `report.md`.

Why:

A local-only footer or HTML comment can make reports traceable back to `verified-final.json`, `evidence-packs.json`, and `grade.json`.

32.2 Medium Refactors

Add `bounty_explain_phase_gate`.

Why:

The gate logic already exists. A read-only explainer would make blocked transitions understandable without first attempting a transition that fails.

Add fixture sessions.

Why:

Small synthetic sessions would make tests, documentation, and offline learning much easier. They would also support future static session explanation tools.

Add an “explain this session” script.

Why:

`bounty_read_pipeline_analytics` already computes much of the needed summary. A static Markdown/HTML report could turn raw artifacts into a readable postmortem.

Expose context budget metadata in hunter briefs.

Why:

When a brief is heavily compressed, the hunter and operator should know. Numeric budget metadata would make missed context easier to diagnose.

32.3 Larger Architecture Improvements

Generate artifact schemas or schema docs from runtime validators.

Why:

The validators are the real contract today. Generated docs or JSON Schemas would make that contract visible without duplicating it by hand.

Generate lifecycle diagrams from tool metadata.

Why:

The tool registry already knows role bundles and written artifacts. A generated graph would show phase -> agent -> tool -> artifact -> gate.

Build a local dashboard.

Why:

Multi-agent state is hard to inspect from raw JSON. A local HTML/TUI dashboard could show waves, handoffs, coverage, findings, verification, evidence, and telemetry in one place.

32.4 Design Principles To Preserve

Keep egress rotation operator-controlled.

Why:

Egress affects authorization boundaries, geofencing, rate limits, and program rules. Agents should surface reachability problems, not silently route around them.

Keep Markdown non-authoritative for machine decisions.

Why:

Markdown is easy for agents to overfit or hallucinate. JSON/JSONL gives the pipeline validation, dedupe, phase gates, and deterministic recovery.

Keep context bounded by role.

Why:

The main value of the architecture is not just parallel agents. It is parallel agents with durable shared state and small, role-specific context packets.

33 Improvement Suggestions

This section is intentionally written as engineering notes, not as a criticism of the current design. The project already has a coherent architecture: narrow agents, MCP-owned state, structured artifacts, write guards, phase gates, and contract tests. The suggestions below are the places where the system could become easier to understand, easier to debug, or harder for agents to misuse.

33.1 1. Add A Machine-Readable Artifact Schema Reference

Suggestion:

Create a checked-in schema reference for the major session artifacts:

```
docs/artifact-schemas.md
docs/schemas/state.schema.json
docs/schemas/attack-surface.schema.json
docs/schemas/wave-handoff.schema.json
docs/schemas/finding.schema.json
docs/schemas/verification-round.schema.json
docs/schemas/evidence-packs.schema.json
docs/schemas/grade.schema.json
```

Why it makes sense:

Right now, artifact schemas are mostly encoded in normalizer functions:

```
mcp/lib/session-state.js
mcp/lib/waves.js
mcp/lib/findings.js
mcp/lib/evidence.js
mcp/lib/chain-attempts.js
```

That is good for enforcement, but harder for a new reader to inspect offline. JSON Schemas or a generated docs page would make the state model easier to learn and reduce accidental prompt/code drift.

Tradeoff:

Schemas can become stale unless generated from or tested against the same normalizers. The right approach would be to add tests that compare sample fixtures against both the schema and runtime validators.

Priority:

High for maintainability and onboarding.

33.2 2. Generate An Artifact Lifecycle Diagram From Tests Or Metadata

Suggestion:

Add a small script that emits a Mermaid or DOT graph showing:

```
phase -> agent -> MCP tool -> artifact -> next phase gate
```

Example output:

```
HUNT -> hunter-agent -> bounty_record_finding -> findings.jsonl
HUNT -> hunter-agent -> bounty_write_wave_handoff -> handoff-wN-aN.json
HUNT -> bounty_apply_wave_merge -> state.json
CHAIN -> chain-builder -> bounty_write_chain_attempt -> chain-attempts.jsonl
VERIFY -> final-verifier -> bounty_write_verification_round -> verified-final.json
VERIFY -> evidence-agent -> bounty_write_evidence_packs -> evidence-packs.json
GRADE -> grader -> bounty_write_grade_verdict -> grade.json
REPORT -> report-writer -> report.md
```

Why it makes sense:

Bob’s architecture is easy to understand once you see the artifact chain, but the chain is currently spread across prompts, tool metadata, and tests. A graph would make the workflow explainable at a glance and would help contributors see where a new tool or artifact belongs.

Tradeoff:

If hand-maintained, it will drift. Prefer generating it from `TOOL_MANIFEST`, agent frontmatter, and a small curated phase map.

Priority:

High for documentation value; medium implementation effort.

33.3 3. Add A Local “Explain This Session” Static Report

Suggestion:

Add a command or script that reads one session directory and writes a static HTML or Markdown explanation:

```
node scripts/explain-session.js example.com > session-explained.md
```

It should explain:

- current phase
- why Bob is or is not allowed to transition
- wave history
- missing or invalid handoffs
- findings flow
- verification outcomes
- evidence readiness
- grade/report trust
- relevant telemetry bottlenecks

Why it makes sense:

`/bob-status` is intentionally compact and `/bob-debug` is interactive. A static explanation would be useful offline, in PR reviews, and when handing a session to another operator. It would also make the JSON state easier to learn because it could link every conclusion back to the artifact that produced it.

Tradeoff:

It overlaps with `bounty_read_pipeline_analytics`. The implementation should reuse `mcp/lib/pipeline-analytics.js` rather than inventing a second analyzer.

Priority:

Medium-high, especially for teaching and postmortems.

33.4 4. Make Context Budgets Visible In Hunter Briefs

Suggestion:

Expand `bounty_read_hunter_brief` output with a small `context_budget` object:

```
{
  "context_budget": {
    "surface_chars": 3210,
    "coverage_items": 18,
    "traffic_items": 22,
    "audit_items": 12,
    "knowledge_chars": 2400,
    "truncated": true
  }
}
```

Why it makes sense:

The repo already caps surface arrays and reports `surface_limits`. A broader budget summary would make it obvious when a hunter is receiving a heavily compressed view. That helps explain missed context and helps tune caps without guesswork.

Tradeoff:

More metadata in the brief consumes a little context. Keep it compact and numeric.

Priority:

Medium. It directly supports the project's context-management model.

33.5 5. Add A “Why This Phase Is Blocked” MCP Tool

Suggestion:

Add a read-only tool:

`bounty_explain_phase_gate`

Inputs:

```
{
  "target_domain": "example.com",
  "from_phase": "HUNT",
  "to_phase": "CHAIN"
}
```

Output:

```
{
  "allowed": false,
  "blockers": [
    {
      "code": "unexplored_high_surfaces",
      "message": "HIGH or CRITICAL attack surfaces remain unexplored",
      "surface_ids": ["billing-api"]
    }
  ],
  "next_actions": [
    "Run another wave for billing-api or transition with an explicit override reason."
  ]
}
```

Why it makes sense:

The gate logic already exists in `mcp/lib/phase-gates.js`. Today, blocked transitions surface as `STATE_CONFLICT` errors from `bounty_transition_phase`. A dedicated read-only explainer would make orchestration and `/bob-status` clearer without requiring a failed transition attempt.

Tradeoff:

It adds another MCP tool and must be reflected in registry metadata, prompt contracts, and generated allowed tools.

Priority:

Medium-high for operator experience.

33.6 6. Add Small Fixture Sessions For Offline Learning

Suggestion:

Create sanitized fixture sessions under a `test/docs` path:

```
docs/fixtures/sessions/clean-submit/  
docs/fixtures/sessions/no-findings-skip/  
docs/fixtures/sessions/pending-wave/  
docs/fixtures/sessions/missing-evidence/
```

Each fixture should include realistic but fake artifacts:

```
state.json  
attack_surface.json  
coverage.jsonl  
findings.jsonl  
handoff-w1-a1.json  
verified-final.json  
evidence-packs.json  
grade.json  
pipeline-events.jsonl
```

Why it makes sense:

Reading artifact examples is the fastest way to understand the system. Tests construct many synthetic sessions already, but they are embedded in test code. Curated fixtures would make the artifact model concrete without exposing real target data.

Tradeoff:

Fixtures can create maintenance overhead. Keep them minimal and validate them in tests.

Priority:

High for offline understanding; medium for production runtime.

33.7 7. Consider A Human-Friendly Session Index

Suggestion:

Maintain or generate an index file per session:

```
session-index.json
```

Possible contents:

```
{  
  "target_domain": "example.com",  
  "session_dir": "...",  
  "current_phase": "VERIFY",  
  "artifacts": {  
    "state": "state.json",  
    "attack_surface": "attack_surface.json",  
    "findings": "findings.jsonl",  
    "final_verification": "verified-final.json"  
  },  
  "latest_events": ["wave_merged", "phase_transitioned", "verification_written"]  
}
```

Why it makes sense:

The session directory contains many files. An index would help humans and tools navigate without guessing which artifact is current.

Tradeoff:

If written as another authoritative artifact, it could create consistency problems. Safer version: generate it on demand from existing artifacts.

Priority:

Medium.

33.8 8. Tighten Prompt-To-Tool Traceability

Suggestion:

For each agent prompt, include a short “artifact contract” block in a consistent format:

Reads:

- bounty_read_findings -> findings.jsonl

Writes:

- bounty_write_verification_round -> verified-final.json

Must not:

- write verifier markdown directly

Why it makes sense:

The information already exists, but it is written in prose and frontmatter. A consistent block would help future prompt edits and make prompt-contract tests simpler to maintain.

Tradeoff:

Prompts are already carefully size-limited. The block should be compact or generated.

Priority:

Medium.

33.9 9. Add A Developer “Trace One Tool” Walkthrough

Suggestion:

Add a doc page that traces a single tool end to end, for example `bounty_write_wave_handoff`:

agent prompt

- > MCP tool schema
- > tool registry metadata
- > executeTool validation
- > waves.js writeWaveHandoff
- > assignment token validation
- > JSON and Markdown writes
- > SubagentStop validation
- > wave merge consumption
- > tests

Why it makes sense:

This project is easiest to understand through one complete vertical slice. `bounty_write_wave_handoff` is a good candidate because it touches prompts, MCP, token provenance, files, hooks, and tests.

Tradeoff:

Documentation maintenance. Keep it focused on one representative path.

Priority:

High for new maintainers.

33.10 10. Make Report Trust Explicit In `report.md`

Suggestion:

Have the report writer include a small internal footer or comment with source artifact references:

Generated from:
- verified-final.json
- evidence-packs.json
- chain-attempts.jsonl
- grade.json

For no-findings reports, include:

Final verification: 0 reportable findings
Grade: SKIP
Evidence: skipped

Why it makes sense:

The system already treats report trust as dependent on final verification, evidence, and grade. Making that visible in the report helps another human understand why the report can or cannot be trusted.

Tradeoff:

Bug bounty submissions should stay clean. This could be included as an HTML comment or a local-only appendix, not necessarily in the final pasted report.

Priority:

Medium.

33.11 11. Add More Negative Fixtures For Agent Drift

Suggestion:

Add tests and fixtures for common bad behaviors:

- hunter writes Markdown handoff only
- hunter marker does not match structured handoff
- balanced verifier drops a finding
- evidence pack includes a token-like string
- grade says SUBMIT with no medium-or-higher final reportable
- report exists but grade is missing

Why it makes sense:

Many of these are already tested in `test/mcp-server.test.js`. Curated negative fixtures would make the failure modes more discoverable and could feed the future static session explainer.

Tradeoff:

More fixtures mean more files. Keep them short and synthetic.

Priority:

Medium.

33.12 12. Separate “Runtime Docs” From “Contributor Docs”

Suggestion:

Split documentation into two mental tracks:

docs/operator/
 first-run.md
 status-debug.md
 egress.md

docs/developer/
 architecture.md
 artifact-schemas.md

```
adding-a-tool.md
prompt-contracts.md
release.md
```

Why it makes sense:

The repo serves two audiences:

- operators running Bob against authorized targets
- contributors changing Bob itself

Mixing those can make onboarding harder. A clearer split would reduce cognitive load.

Tradeoff:

Docs navigation must stay simple. Too many docs can be worse than too few.

Priority:

Low-medium.

33.13 13. Add A Lightweight Local TUI Or HTML Dashboard

Suggestion:

Generate a local dashboard from `bounty_read_pipeline_analytics`:

```
hacker-bob doctor-session example.com --html
```

The dashboard could show:

- phase timeline
- waves and handoff readiness
- surface coverage
- findings flow
- verification disposition table
- evidence coverage
- tool failures
- egress/geofence warnings

Why it makes sense:

The data already exists. A dashboard would make multi-agent state easier to inspect without reading raw JSON. It would also be useful during demos and debugging.

Tradeoff:

This is polish, not core correctness. It should not replace JSON artifacts or MCP analytics.

Priority:

Low-medium unless operator debugging becomes a major bottleneck.

33.14 14. Keep Strong Boundaries Around Automatic Egress Rotation

Suggestion:

Keep the current operator-controlled egress model. If adding future automation, make it recommendation-only by default:

```
"default egress has repeated first-party failures; operator may resume with --egress gr-residential"
```

Why it makes sense:

Egress changes can affect authorization, rate limits, geofencing, program rules, and legal/ethical boundaries. The current design is conservative: agents detect reachability problems and ask the operator rather than silently rotating.

Tradeoff:

Manual intervention slows autonomous runs, but it preserves operator control.

Priority:

High as a design principle.

33.15 15. Make “Context Is A Product Feature” A First-Class Doc Theme

Suggestion:

Add a short architecture doc specifically about context management:

`docs/developer/context-management.md`

It should explain:

- why the root orchestrator uses summaries
- why hunters get one assigned brief
- why arrays are capped
- why Markdown is not authoritative
- how telemetry stays metadata-only
- how resume works after context loss

Why it makes sense:

The central technical achievement of this repo is not just “many agents.” It is many agents coordinated through bounded context and structured durable state. Making that explicit will help future AI-generated changes preserve the architecture instead of accidentally expanding prompts or stuffing too much data into agent context.

Tradeoff:

Documentation does not enforce behavior by itself. Pair this with prompt contract tests around context caps and brief shape.

Priority:

High for long-term agent quality.

34 FAQ

34.1 Where is the main entry point?

There are several, depending on what you mean:

- CLI entry: `bin/hacker-bob.js`
- installer: `scripts/install.js`
- MCP server: `mcp/server.js`
- `/bob-hunt` orchestration prompt: `.claude/skills/bob-hunt/SKILL.md`
- MCP tool registry: `mcp/lib/tool-registry.js`

For runtime behavior, start with `/bob-hunt` and `mcp/server.js`.

34.2 Why use MCP at all?

Because separate agents need shared memory, validation, and durable state. Without MCP, everything would depend on chat history and fragile prose. MCP turns key actions into typed tool calls that write structured local artifacts.

34.3 Why not just let agents write files directly?

Direct file writes create drift. An agent might write malformed JSON, skip a required field, overwrite someone else's data, or invent state. MCP tools validate and normalize writes. The write guard blocks direct writes to MCP-owned artifacts.

34.4 Why JSONL for findings instead of one JSON file?

Append-only logs are safer for multi-step agent work. Each record is one JSON line. Appending a finding or coverage record does not require rewriting a whole array and does not risk losing older records if a later write fails.

34.5 Why are Markdown files still present?

They are human/debug mirrors. For example, `findings.md` is easier to read than `findings.jsonl`, but orchestration should trust `findings.jsonl`.

34.6 What happens if the root chat loses context?

The user can run:

```
/bob-hunt resume <domain>
```

The orchestrator reads MCP state and artifacts, sees the current phase and pending wave, and continues from there.

34.7 What is `pending_wave`?

It means a wave was started but not merged. Hunters may still be running, or some handoffs may be missing. Bob should not start another wave or move to CHAIN while `pending_wave` is set.

34.8 Why forbid same-turn merge after spawning hunters?

Hunters run in the background. The root turn that spawns them should not immediately assume their handoffs exist. The launch-turn barrier prevents the orchestrator from racing ahead before background work finishes.

34.9 What is a handoff token?

When `bounty_start_wave` creates assignments, it generates a token for each assigned hunter. The token hash is stored in `wave-N-assignments.json`; the plain token is returned only to the orchestrator so it can put it in that hunter's prompt. `bounty_write_wave_handoff` verifies the token before accepting the handoff.

34.10 Why does the hunter also emit BOB_HUNTER_DONE?

The marker lets the SubagentStop hook confirm that the final message is a proper hunter completion. The hook then checks that the structured handoff exists and matches the marker. The marker alone is not enough.

34.11 Who owns attack_surface.json?

The recon agent writes it directly. This is one of the explicitly allowed agent-owned session files. After that, MCP reads it and validates surface IDs.

34.12 How does Bob avoid repeating work?

Several mechanisms:

- `state.explored`
- `coverage.jsonl`
- dead-end and WAF-blocked endpoint lists
- hunter brief coverage summaries
- wave merge requeue logic
- finding dedupe keys
- HTTP audit summaries and circuit breaker summaries

34.13 How does Bob decide which surfaces to hunt?

The recon agent gives each surface a priority. MCP ranking can compute runtime ranking from attack surface data, imported traffic, and public intel hints. The README says ranking is runtime prioritization, not a durable rewrite in normal status/brief reads.

34.14 What is the difference between coverage and findings?

Coverage records what was tested and what remains promising, blocked, or worth requeueing. Findings are proven vulnerability claims. Coverage can exist without findings.

34.15 Why have three verification rounds?

To reduce false positives:

- Brutalist verifier tries to kill weak findings.
- Balanced verifier catches false negatives or over-corrections.
- Final verifier freshly replays reportable survivors.

The final round is the gate for evidence collection.

34.16 Why does evidence come after final verification?

Evidence collection can touch the target again. Bob should collect formal evidence only for findings that survived final verification, not for every weak hunter claim.

34.17 What makes evidence valid?

`evidence-packs.json` must cover every final reportable finding, avoid secrets and raw sensitive values, stay bounded, avoid duplicate finding IDs, and contain representative samples plus summaries.

34.18 What determines SUBMIT, HOLD, or SKIP?

The grader scores final survivors on five axes. Code enforces score math and verdict thresholds:

- SUBMIT: score at least 40 and at least one medium-or-higher reportable finding
- HOLD: score 20-39 when reportability exists
- SKIP: score below 20 or no medium-or-higher reportable finding

34.19 Can a no-finding session still reach REPORT?

Yes. If final verification has no reportables, evidence can be skipped, the grader writes a terminal **SKIP**, and the reporter writes a no-findings closeout.

34.20 Where is auth stored?

In:

```
~/bounty-agent-sessions/[domain]/auth.json
```

The file is mode 0600. Agents should not read it directly. They use `bounty_list_auth_profiles` and `auth_profile` on `bounty_http_scan`.

34.21 Does Bob block private IPs and localhost?

Not by default. Bob allows them by default for authorized labs, internal scopes, VPN targets, SSRF chains, and user-authorized pivots. HTTP tools can block them when called with `block_internal_hosts: true`.

34.22 What is an egress profile?

A named route for Bob HTTP scan calls. The default profile is direct local connection. Non-default profiles can point to operator-managed proxies via environment variable references.

34.23 Why is the scope guard a no-op?

The current design keeps scope decisions permissive at the hook layer and puts optional blocking into MCP HTTP calls. This is tested behavior, not missing code.

34.24 What is /bob-status?

A compact, read-only status command. It reads analytics, state summary, wave status, and key artifacts to tell the operator where the latest or selected session stands.

34.25 What is /bob-debug?

A deeper read-only debugger. It starts with telemetry and analytics, then can inspect artifacts and narrow transcript windows. With `--deep`, it can run the local policy replay diagnostics when a policy/refusal issue is identified.

34.26 What is policy replay?

The `testing/policy-replay/` harness can replay minimized transcript cases against agent prompts to diagnose false refusals, policy stalls, or unsafe compliance risks. It is not part of normal hunting.

34.27 How are generated prompt surfaces kept in sync?

Scripts:

```
scripts/generate-agent-tools.js
scripts/generate-bountyagent-skill.js
```

Prompt contract tests ensure generated files and registry metadata agree.

34.28 Which tests should I read first?

Start with:

```
test/prompt-contracts.test.js
test/mcp-server.test.js
test/install-smoke.test.js
test/test-write-guard.py
```

The test names are a good architecture outline.

34.29 What should I trust if Markdown and JSON disagree?

Trust the MCP-owned structured JSON/JSONL artifact. Markdown mirrors are for humans and debugging.

35 Glossary

35.1 Agent

A Claude Code subagent with a narrow prompt and tool list.

35.2 Orchestrator

The root `/bob-hunt` flow. It coordinates agents and state transitions.

35.3 MCP

Model Context Protocol. In Bob, it is the local server that owns structured state and exposes `bounty_*` tools.

35.4 Surface

An attackable unit from `attack_surface.json`, such as an API, auth flow, admin panel, upload feature, CMS, GraphQL endpoint, static property, or token contract artifact.

35.5 Wave

A batch of hunter assignments. Each assignment maps one agent ID to one surface ID.

35.6 Handoff

A structured end-of-wave summary written by a hunter through MCP.

35.7 Coverage

Durable records of endpoint/bug-class/auth-profile tests and whether each was tested, blocked, promising, needs auth, or should be requeued.

35.8 Finding

A proven vulnerability claim recorded by a hunter.

35.9 Chain Attempt

A tested hypothesis that findings or handoff notes combine into stronger impact.

35.10 Verification Round

One of brutalist, balanced, or final. Each round records dispositions for findings.

35.11 Evidence Pack

Bounded redacted proof for a final reportable finding.

35.12 Grade

The structured `SUBMIT/HOLD/SKIP` verdict.

35.13 Pipeline Analytics

Metadata-only summary of session health, bottlenecks, and next actions.

36 File Map Cheat Sheet

36.1 User-Facing Docs

README.md
docs/FIRST_RUN.md
docs/TROUBLESHOOTING.md
docs/ROADMAP.md
CHANGELOG.md
docs/releases/

36.2 Claude-Facing Runtime

.claude/skills/
.claude/agents/
.claude/commands/
.claude/hooks/
.claude/rules/
.claude/knowledge/
.claude/bypass-tables/

36.3 MCP Core

mcp/server.js
mcp/lib/transport.js
mcp/lib/tool-registry.js
mcp/lib/dispatch.js
mcp/lib/envelope.js
mcp/lib/tool-validation.js
mcp/lib/tools/index.js

36.4 MCP State And Pipeline

mcp/lib/paths.js
mcp/lib/storage.js
mcp/lib/session-state.js
mcp/lib/phase-gates.js
mcp/lib/waves.js
mcp/lib/assignments.js
mcp/lib/coverage.js
mcp/lib/findings.js
mcp/lib/chain-attempts.js
mcp/lib/evidence.js
mcp/lib/pipeline-analytics.js
mcp/lib/tool-telemetry.js

36.5 HTTP, Auth, Recon Enrichment

mcp/lib/http-scan.js
mcp/lib/safe-fetch.js
mcp/lib/url-surface.js
mcp/lib/http-records.js
mcp/lib/auth.js
mcp/lib/signup.js
mcp/lib/temp-email.js
mcp/lib/egress-profiles.js
mcp/lib/public-intel.js
mcp/lib/static-artifacts.js

mcp/lib/ranking.js
mcp/lib/hunter-brief.js
mcp/redaction.js

36.6 Installer And Lifecycle

bin/hacker-bob.js
install.sh
scripts/install.js
scripts/lifecycle.js
scripts/merge-claude-config.js
scripts/release-check.js
scripts/generate-agent-tools.js
scripts/generate-bountyagent-skill.js

36.7 Tests

test/mcp-server.test.js
test/prompt-contracts.test.js
test/install-smoke.test.js
test/cli.test.js
test/package.test.js
test/update-check.test.js
test/policy-replay.test.js
test/test-write-guard.py

37 Final Reading Advice

When reading this repo, do not start by trying to understand every helper function. Follow the data:

```
/bob-hunt
-> state.json
-> attack_surface.json
-> wave assignments
-> hunter brief
-> coverage/findings/handoff
-> wave merge
-> chain attempts
-> verification rounds
-> evidence packs
-> grade
-> report
```

Then follow the enforcement:

```
tool registry
-> tool validation
-> session locks
-> phase gates
-> write guard
-> hunter stop hook
-> prompt contract tests
```

If you understand those two paths, the rest of the repo becomes much easier: install scripts move the framework into place, docs explain the user flow, telemetry explains failures, and tests keep the contract from drifting.