ChatGPT

Sheratan `sauber_main` (commit 8a33fe2a2894625841e700336812ccbbdd4970e1) – Systemzustands-Audit

1 Aktuelle Architektur (10 bullets)

1. **Source-of-Truth vs. Mirror:** Die aktive Codebasis befindet sich unter `sauber_main/core/` und `sauber_main/runtime/`. In der Datei *zurechtfinden.md* wird ausdrücklich erwähnt, dass `core/` das laufende System ist und dass `mesh/core/` nur ein Migrations-Mirror ist – Worker sollen **nicht** in `mesh/core/` implementieren [1].

2. **Core-App & Lifespan:** `core/main.py` startet eine FastAPI-App (Port 8001) und initialisiert alle Subsysteme (Datenbank, State-Machine, Baseline-Tracker, Anomaly-Detector, Self-Diagnostic-Engine, Dispatcher, ChainRunner). Der Lifespan initialisiert die Datenbank, lädt den Systemzustand, startet den Diagnostik-Loop und übergibt den Dispatcher/ChainRunner an die event-Loops.

3. **Dispatcher-Schicht:** Die Klasse `Dispatcher` orchestriert Job-Dispatch. Sie holt Pending Jobs aus der Storage, dedupliziert (Idempotency), prüft Abhängigkeiten, sortiert nach Priorität und ruft `WebRelayBridge.enqueue_job()` auf, wenn Rate-Limits eingehalten werden. In `core/main.py` wird der Dispatcher im Hintergrund-Thread gestartet.

4. **ChainRunner:** Der `ChainRunner` läuft ebenfalls im Hintergrund und erzeugt Jobs aus Chain-Spezifikationen. Der `job_chain_manager` verwaltet Chain-Kontexte und Follow-up-Specs.

5. **Decision Trace & State-Machine:** Sheratan verwendet einen deterministischen State-Machine (`core/state_machine.py`) und ein Decision-Trace-Schema. Die JSON-Schema für `decision_trace_v1` verlangt, dass jeder Trace-Eintrag ein `state`-Objekt mit `context_refs` **und** `constraints` enthält [2]. Fehlende `constraints` führen zu Schema-Fehlern (s. Blocker 1).

6. **WebRelay Bridge:** Die `WebRelayBridge` übersetzt Jobs in ein einheitliches Datei-Format für die externe LLM-Worker-Umgebung. Beim Dispatch ruft sie `LedgerClient.charge()` auf, um Token aus dem Mesh-Ledger zu buchen. Wenn das Konto des Auftraggebers nicht existiert, druckt der Code eine Warnung („Ledger charge failed …“) [3].

7. **Storage & DB:** `core/storage.py` implementiert ein File-basiertes Speichersystem (JSON-Dateien). Der Refactoring-Plan sieht einen Umstieg auf SQLite vor; das Decision-Trace-Schema und die Datenmodelldefinitionen sind bereits vorbereitet [4]. Die Datenmodelle definieren Jobs, Hosts und Rate-Limit-Configs.

8. **Runtime-Zonen:** Die Laufzeit-Daten liegen in `runtime/` (Mission- und Job-Dateien, Chains, Baselines, Logs). Diese Zone ist der „Design-For-Reflection“-Ort (Baselines, Health-Reports, Anomalies) [5] .

9. **Mesh & External Directory:** Eine Parallelstruktur `mesh/` und `external/` existiert. Laut Migrations-Plan bleibt die **alte Struktur (`v_core`, `v_mesh`, `v_mini`, `runtime`) aktiv**, während `mesh/` und `external/` Kopien enthalten – das System läuft weiterhin aus der alten Struktur [6] . Diese Mirror-Verzeichnisse sind für die spätere Migration vorgesehen und dürfen aktuell nicht verändert werden.

10. **Security Gates & Gateway-Konzept:** Das Refactoring-Plan sieht ein Gate-System (G0 bis G4) vor, das im `mesh/core/gates`-Verzeichnis implementiert wird [7] . Dieses Gate-Layer bildet eine klar definierte „Gateway/Policy-Boundary“ zwischen externen Aufrufen und internen Mission/Task-Ausführungen und ersetzt langfristig das alte `sheratan-gateway`-Container-Konzept.

2 Funktionsstatus („Ampel“)

Subsystem	Status	Begründung
Boot	🟢 (grün)	`core/main.py` startet, initialisiert Datenbank und Diagnostik-Module; der Dispatcher und ChainRunner laufen im Hintergrund; ein Smoke-Test (`tests/smoke_e2e_job.py`) erzeugt Mission/Task/Job und wird erfolgreich abgeschlossen.
API	🟢 (grün)	FastAPI-Endpoints für Mission/Task/Job CRUD, Dispatch, Sync und System-Health laufen stabil. CORS ist aktiviert, und `/api/system/health` prüft Ports der internen Dienste.
DB/Storage	🟢 (grün)	Das File-basierte Storage funktioniert (Jobs, Tasks, Missions, Chains). SQLite-Schema in `docs/SHERATAN_REFACTORING_PLAN.md` ist vorbereitet, aber noch nicht im aktiven Pfad.
Jobs/ Dispatcher	🟡 (gelb)	Dispatcher arbeitet, aber es gibt offene TODOs: deduplizierter Rate-Limiter, Job-Dependencies, Retry-Backoff. Bei fehlenden Ledger-Konten bricht die Abrechnung ab (siehe Blocker 2).
Worker/Mesh	🟠 (orange)	Der WebRelay-Mechanismus schreibt Job-Dateien in `webrelay_out/`, aber es existiert noch kein automatisierter LLM-Worker; Offgrid-Hosts/Broker laufen nur als Konzept. Heartbeat-Endpoints existieren, aber es gibt keine echte Host-Registrierung.
Decision Trace/State Machine	🟡 (gelb)	State-Machine und Trace-Logger sind aktiv. Allerdings werden einige Trace-Einträge ohne `constraints` erstellt, was zu Schema-Validierungsfehlern führt [2] (Blocker 1).

3 Top-5 Blocker (Root-Cause → Minimaler Fix)

#	Blocker	Root-Cause / Evidence	Minimaler Patch-Ort
1	*Schema-Validation Fehler – „`constraints` is a required property“*	Das Decision-Trace-Schema verlangt, dass jedes `state`-Objekt `context_refs` und `constraints` enthält [2]. Mehrere Trace-Logger-Aufrufe übergeben nur `context_refs`.	In `core/main.py` existiert bereits `normalize_trace_state()` (Zeilen 15-22) – dieser fügt notfalls ein leeres `constraints` ein. **Lösung:** Alle Trace-Logger-Aufrufe sollten `normalize_trace_state()` auf `state` anwenden oder manuell ein leeres `constraints` setzen.
2	*Ledger-Charge schlägt fehl („Account 'smoke-user' does not exist“)*	Die `WebRelayBridge` ruft bei Dispatch `ledger.charge(payer, worker_id, cost, job_id)` auf. Wenn der Ledger keine Benutzerkonten kennt, führt dies zu einer Fehlermeldung (Zeilen 169–177) [3].	Kurzfristig kann man `cost=0` für interne Smoke-Tests setzen oder fehlende Benutzerkonten in der Ledger-JSON (`mesh/registry/ledger.json`) anlegen. Besser wäre, die Ledger-Initialisierung optional zu machen und bei fehlendem Account eine Warnung statt Exception zu erzeugen.
3	*Konfusion durch Parallel-Verzeichnisstrukturen*	Das Migrations-Map-Dokument betont, dass die alte Struktur (`v_core`, `v_mesh`, `v_mini`, `runtime`) weiterhin aktiv bleibt und `mesh/` nur ein Spiegel ist [6]. Einige Worker/Developer implementieren irrtümlich im neuen Pfad.	Klare Dokumentation (siehe *zurechtfinden.md*) befolgen: Implementieren nur in `sauber_main/core/` und `runtime/` [1]. Code-Linters oder Import-Guards können Warnungen ausgeben, wenn `mesh/` genutzt wird.
4	*Unvollständige Offgrid-Integration*	Das Offgrid-Broker/Host-System ist teilweise kopiert, aber es existiert kein produktiver Host-Daemon; Heartbeat-Loops registrieren keine realen Worker.	Für die Phase-Abschluss reicht es, Offgrid zu deaktivieren. Längerfristig: Worker-Daemon implementieren oder LLM-Worker via WebRelay aktivieren.

#	Blocker	Root-Cause / Evidence	Minimaler Patch-Ort
5	*Fehlende Rate-Limit- und Retry-Konfiguration*	Der Refactoring-Plan listet Idempotenz, Retry, Timeout, Prioritäts-Queue, Rate-Limiting und Abhängigkeiten als „Production-Features“ [8]; in der aktiven Version sind diese nur rudimentär implementiert.	Die TODO-Dateien in `mesh/core/` listen fehlende Features; minimal kann man im Dispatcher definierte Limits („max_jobs_per_minute“, „max_concurrent_jobs“) in `runtime/config.json` einstellen und auf `RateLimiter.check_limit` anwenden.

4 Definition-of-Done (DoD) & Testabfolge

Die Phase „sauber_main Core v2“ gilt als abgeschlossen, wenn folgende Bedingungen erfüllt sind:

1. **Startup & API funktionieren ohne Fehler.**
2. Starte `python core/main.py` (Port 8001) und prüfe, dass der Lifespan alle Module initialisiert (Baselines, Diagnostics, Dispatcher, ChainRunner).
3. Rufe `GET /api/system/health` auf – das Ergebnis sollte „Core API: active“ und für nicht gestartete Dienste „down“ anzeigen.
4. **Mission/Task/Job Lifecycle funktioniert.**
5. Erstelle via API eine Mission (`POST /api/missions`), dann einen Task (`POST /api/missions/{id}/tasks`) und einen Job (`POST /api/tasks/{id}/jobs`).
6. Überprüfe mit `GET /api/jobs/{job_id}`, dass der Job `status=pending` wird.
7. Warte ab, bis der Dispatcher den Job zu `working` und `completed` ändert (kann via `list_jobs` oder Log verfolgt werden).
8. Rufe `POST /api/jobs/{job_id}/sync` auf, um das Worker-Result zu synchronisieren.
9. **Decision-Trace validiert.**
10. Prüfe, dass Trace-Logger-Einträge (z.B. in `logs/decision_trace.jsonl`) gültig gegen das Schema sind (kein „constraints missing“-Fehler).
11. Implementiere, falls nötig, `normalize_trace_state()` bei allen Trace-Loggings.
12. **Ledger optional oder fixiert.**
13. Für Tests ohne Ledger: Setze `SHERATAN_LEDGER_URL` leer und erstelle in `mesh/registry/ledger.json` einen Test-Benutzer mit genügend Balance.
14. Prüfe `GET /api/mesh/ledger/{user_id}` – der Rückgabewert sollte ein Balance-Objekt liefern.
15. **Smoke-Test besteht.**

16. Führe das vorhandene `tests/smoke_e2e_job.py` aus. Der Test erzeugt eine Mission, einen Task und einen `read_file`-Job und verifiziert, dass der Job vollständig abgeschlossen wird. Das Terminal sollte „PASS: Job completed successfully“ melden.

17. **Phase-Grenze gezogen.**

18. Keine neuen Features in `mesh/` implementieren. Fokus: Stabilisieren von Core v2.
19. Dokumentation aktualisieren (README, ARCHITECTURE.md) und die Migration-Pläne für die nächste Phase definieren.

5 Gateway-Layer (Policy / Boundary)

Das frühere `sheratan-gateway`-Docker-Container war ein Versuch, Sheratan über einen separaten Gateway-Service zu betreiben. Es war kein Seitenweg, sondern ein Architektur-Experiment. In der neuen Architektur wird das Gateway-Konzept in zwei Teile integriert:

1. **Mesh-Gates (G0–G4):** Im `mesh/core/gates`-Verzeichnis existieren Gate-Handler (Barrier, Payload, Resource, Capability, Policy) [7] . Diese definieren Authentifizierungs-, Ressourcen- und Policy-Checks für interne Jobs.

2. **Gatekeeper-Service:** In `external/gatekeeper` entsteht ein eigenständiges Service, das die Gate-Entscheidungen zentral durchsetzt (API-Boundary). Dieses Service ersetzt langfristig den alten Gateway-Container.

Empfehlung: Das alte Gateway-Image nicht reaktivieren. Die Gateway-Funktion soll als *Boundary-Layer* in den Mesh-Gates und den External-Services integriert werden. Erst wenn Core v2 stabil ist, sollte man das Gatekeeper-Service implementieren und zwischen Front-Ends (WebRelay, Dashboard) und dem Core schalten, um Authentifizierung, Rate-Limitierung und Sicherheitsrichtlinien durchzusetzen – ohne das aktuelle System zu destabilisieren.

Letzte Aktualisierung (Audit): 14 Jan 2026 – Zürich.

---

[1] [5] zurechtfinden.md
https://github.com/J3r3C0/sauber_main/blob/8a33fe2a2894625841e700336812ccbbdd4970e1/zurechtfinden.md

[2] decision_trace_v1.json
https://github.com/J3r3C0/sauber_main/blob/8a33fe2a2894625841e700336812ccbbdd4970e1/schemas/decision_trace_v1.json

[3] webrelay_bridge.py
https://github.com/J3r3C0/sauber_main/blob/8a33fe2a2894625841e700336812ccbbdd4970e1/core/webrelay_bridge.py

[4] [7] [8] SHERATAN_REFACTORING_PLAN.md
https://github.com/J3r3C0/sauber_main/blob/8a33fe2a2894625841e700336812ccbbdd4970e1/docs/SHERATAN_REFACTORING_PLAN.md

[6] MIGRATION_MAP.md
https://github.com/J3r3C0/sauber_main/blob/8a33fe2a2894625841e700336812ccbbdd4970e1/docs/MIGRATION_MAP.md