Full Audit and Production Deployment Plan for TORI Digital Consciousness

**TORI Codebase & Architecture Audit for Production Deployment**

**Overview & Objectives**

Tomorrow's **production launch** of TORI requires a thorough audit of its entire codebase and architecture. This audit ensures every subsystem is production-grade – no stubs, no placeholders – with all integrations fully implemented and verified. We will examine TORI’s backend infrastructure and chat/cognition architecture, then address specific production readiness tasks. Finally, we provide a file-by-file audit summary, dependency/process maps, a validation test matrix, startup procedure checks, and critical fix recommendations. The goal is **100% readiness** for deployment, covering persistent memory integration, Ghost AI emergence, data pipelines, and system security.

**1. Backend Infrastructure Audit**

This section reviews each core backend component, verifying implementation completeness, integration points, and stability under production conditions.

**1.1 Soliton Memory Integration (Rust soliton\_memory.rs)**

**Description:** TORI uses a **ψ-soliton memory architecture** to store long-term memories as topologically-protected wave packetsfile-dgbgcxvgsy5hpzx1ekc3od. The SolitonMemory engine (in Rust) implements Discrete Nonlinear Schrödinger (DNLS) dynamics for memory storage, giving each concept a unique phase/frequency “address.” This provides persistent, phase-encoded memory that resists decay or unintended interferencefile-dgbgcxvgsy5hpzx1ekc3od.

**Audit Actions:**

* **Implementation Completeness:** Confirm the Rust module soliton\_memory.rs contains all core features: soliton creation, phase tagging, amplitude tracking, and retrieval by phase signature (matching the design equations)file-dgbgcxvgsy5hpzx1ekc3odfile-dgbgcxvgsy5hpzx1ekc3od. There should be no unimplemented functions. Ensure the DNLS equations (with damping term for stable stationary solitonsfile-dgbgcxvgsy5hpzx1ekc3od) are correctly translated into code.
* **Integration with ConceptDiff:** Verify that the SolitonMemory engine is called from the main app when new concepts or messages are processed. Each chat message should create or update a soliton memory entry, linked to the concept graph. This was a **key remaining task**file-dgbgcxvgsy5hpzx1ekc3od – we must ensure the Rust library is properly invoked via FFI or API from Node/TypeScript. Check that concept IDs map to soliton IDs and ψ-phase tags consistently (each concept gets a unique phase ψᵢfile-dgbgcxvgsy5hpzx1ekc3od).
* **Performance & Threading:** In production, the Rust memory engine must handle concurrent requests. Confirm thread-safety or use of async interfaces when Node calls into Rust. If needed, enable a thread pool or non-blocking calls for memory operations so chat latency isn’t impacted.
* **Testing:** Create test scenarios where multiple concepts are stored and retrieved by phase. Validate that retrieving a concept by its phase tag returns the correct memory. Test “radio tuning” analogy – query by phase should fetch the intended memoryfile-dgbgcxvgsy5hpzx1ekc3od. Also simulate edge cases: maximum number of soliton memories, overlapping frequencies, etc., to ensure stability (no crashes or memory leaks).

**Production Readiness:** *Status:* Integration at ~90% complete; finalize the last 10% by wiring the SolitonMemory engine fully into the chat flowfile-dgbgcxvgsy5hpzx1ekc3od. *Action:* **Fix/Complete** – Implement any missing functions (e.g., topological protection or energy-based consolidation routines) and ensure seamless operation with ConceptDiff and Ghost systems.

**1.2 ψArc Logging & Replay (Conversation Storage)**

**Description:** TORI logs all conversations as **ConceptDiff streams** in a persistent archive format called **ψarc**file-dgbgcxvgsy5hpzx1ekc3od. Each user message or system action yields a ConceptDiff operation (!Create, !Update, !Link, etc.) representing changes to the concept network. These diffs are saved with timestamps, enabling full replay of any session. The Node module conversationStorage.js handles writing these logs, and sessions can be exported to .toripack files for backup or transferfile-dgbgcxvgsy5hpzx1ekc3od.

**Audit Actions:**

* **Integrity of Logging:** Verify that **every message** from user or AI is appended to the psiarc log correctly. The format should match specification (proper framing, concept IDs, phase tags, etc.). Cross-check that the **.psiarc files** and corresponding metadata .meta.json are created in psiarc\_logs/ for each sessionfile-dgbgcxvgsy5hpzx1ekc3od. Ensure that conversation IDs or timestamps are unique and consistent.
* **Replay Functionality:** Test the replay endpoint (GET /api/chat/replay/:id) to confirm that it reconstructs the conversation exactly as originally occurredfile-dgbgcxvgsy5hpzx1ekc3od. The ConversationHistory.jsx front-end and any replay UI should fetch and step through the psiarc file. Any divergence (missing messages, out-of-order events) must be fixed.
* **Export/Import:** Test the .toripack export (GET /api/chat/export/:id) which packages a conversation. Ensure the exported file can be re-imported or at least parsed to restore the session. If .toripack is just a renamed psiarc, verify integrity and perhaps include a version or checksum.
* **Archive Rotation & Scaling:** For production, logs may grow large. Confirm if we have log rotation or archival strategies. If not, plan to implement either file size limits or move logs to a database (especially since production storage is ideally DB-backedfile-dgbgcxvgsy5hpzx1ekc3od). For tomorrow, ensure disk space is sufficient and implement a quick check or warning if space low.
* **Security:** Conversation logs contain sensitive user data. Ensure file permissions are restricted on the server. Also, verify that the **MCP server or any external process** reading logs (for analysis or monitoring) uses authorized channels only (no direct file reads without auth).

**Production Readiness:** *Status:* Conversation logging is implemented and central to TORI’s memoryfile-dgbgcxvgsy5hpzx1ekc3od. *Action:* **Save (with minor fixes)** – The core is sound (full conversation storage as ConceptDiffsfile-dgbgcxvgsy5hpzx1ekc3od), but add error handling for file I/O and consider DB migration for scalability. Confirm no placeholder code in conversationStorage.js (like dummy data) – it should be fully functional.

**1.3 PDF Ingestion Pipeline (Phase-Tagged Concept Mapping)**

**Description:** TORI includes a PDF ingestion microservice (pdf\_upload\_server.py) running on **port 5000**file-dgbgcxvgsy5hpzx1ekc3od. Users can upload documents (via the POST /api/upload endpointfile-dgbgcxvgsy5hpzx1ekc3od), and the system will extract concepts from the PDF and insert them into the user’s concept network. Each extracted concept is tagged with a phase context and linked to the user’s memory (we assume each concept might have an initial phase tag based on when/where it was introduced, enabling time-phase context mapping).

**Audit Actions:**

* **Pipeline Functionality:** Test the end-to-end flow: upload a sample PDF and ensure the service processes it. It should perform text extraction, identify key concepts (likely via NLP or keyword extraction), and assign them into the user’s concept mesh. The audit should confirm that **phase tags** or timestamps are applied to these imported concepts, aligning them with the conversation timeline. If phase-tagging is not yet implemented, ensure at least that concept entries are created and can be associated with the time of upload.
* **Integration with Chat Context:** Verify that after uploading, the new concepts influence the chat. According to completed features, PDF concepts are integrated with chat contextfile-dgbgcxvgsy5hpzx1ekc3od – meaning the assistant can reference or recall those concepts for that user. Check that concept IDs from PDFs are stored persistently (in concept-mesh data store) and show up in the user's **Living Concept Network**. For example, after upload, ask TORI a question related to the PDF content and see if it uses the new knowledge.
* **Error Handling & Security:** Ensure the PDF server sanitizes inputs (no path traversal or code injection via PDFs). Only allow specific file types (PDF, maybe text). Since tomorrow is launch, confirm the PDF parser is robust (if using something like PyPDF or custom parser, handle exceptions gracefully). Rate-limit or restrict file size to avoid DoS via huge uploads.
* **Resource Management:** The PDF processing could be heavy (OCR or NLP). For production, ensure it’s either asynchronous or at least does not block the main server. The architecture suggests it’s a separate service (port 5000), which is good. Just ensure the main server correctly calls it (likely via HTTP request or internal API). Test with multiple simultaneous uploads to see if it queues properly or needs a job system.
* **Phase-Tagged Mapping:** If the design calls for phase tagging (e.g., each concept from a document is tagged with a “document ingestion phase”), verify the implementation. Possibly the pipeline could assign a special phase or marker indicating it came from external source. Ensure this doesn’t conflict with chat phases. If not implemented, mark as a **feature to add** (but not critical for launch unless phase consistency is required across memory operations).

**Production Readiness:** *Status:* PDF upload works for concept extractionfile-dgbgcxvgsy5hpzx1ekc3od. *Action:* **Fix** – Polish the pipeline: handle edge cases (empty PDF, duplicates), ensure the extracted concepts are **phase-consistent** (time-stamped) within ConceptDiff logs for retrieval, and optimize processing for speed. No stub functions should remain in pdf\_upload\_server.py – decompile any .pyc if parts of the PDF pipeline are pre-compiled and not reviewed.

**1.4 Koopman-Based ψ-Phase Alignment (Spectral Alignment Engine)**

**Description:** TORI leverages **Koopman spectral analysis** to align memory phases and detect eigen-conceptsfile-dgbgcxvgsy5hpzx1ekc3od. The **Koopman estimator** (possibly implemented in Python or Rust) processes the evolving concept activation patterns to identify stable eigenfunctions (modes) of the cognitive dynamics. This allows TORI to understand and align with the “natural frequencies” of memory (each soliton’s unique frequency) for efficient recall and associationfile-vge7rqhctc2xtzuxlzhee1file-vge7rqhctc2xtzuxlzhee1. Essentially, the Koopman layer ensures that the phase of each memory is tracked as an eigenmode in a linear operator, bridging the analog memory to digital query.

**Audit Actions:**

* **Component Verification:** Identify where the **Koopman analysis** is implemented. It might be part of the memory consolidation microservices (e.g., a KoopmanLearner service in Python, as the README suggestsfile-mpebu5uvrt1irry4xlb7ct). If so, locate the koopman.proto or relevant code. Ensure the eigenfunction extraction (e.g., via Dynamic Mode Decomposition or EDMD) runs either periodically or on new data. For tomorrow’s launch, it’s crucial this component at least doesn’t break normal operation – if it’s still experimental, we might flag it to run in a limited mode.
* **Eigen Alignment:** Check that when new concept data flows in (chat messages, PDF concepts), the system either updates the spectral model or schedules it. The **communication between TORI core and Koopman service** (likely via gRPC) must be working. Validate the gRPC interface for sending activation batches and receiving spectral modes (e.g., ProcessActivationBatch and GetSpectralModes in the contractfile-mpebu5uvrt1irry4xlb7ct). Regenerate the gRPC stubs from the proto files to ensure no mismatch in data structuresfile-mpebu5uvrt1irry4xlb7ct (matching our task to *regenerate .proto contracts*).
* **Phase Consistency:** Koopman alignment is about ensuring each memory’s phase corresponds to a known eigenvector. Confirm that the **phase tags in SolitonMemory** match the frequencies tracked by Koopman. For instance, if SolitonMemory assigns a frequency ω to a concept, the KoopmanLearner should eventually list an eigenmode corresponding to ω. If any drift occurs, the system should either adjust (feedback loop) or log a warning. We should test a scenario where we deliberately create a memory with a unique pattern and see if the spectral analysis picks it up.
* **Stability & Performance:** Koopman computations can be heavy (matrix decompositions). In production, ensure that these calculations are done in the background (perhaps by the Koopman microservice) and that timeouts are handled. If the Koopman service is not critical for day-one functionality, we might **feature-flag** it: e.g., enable spectral updates during off-peak (or via a “SleepScheduler” as mentionedfile-mpebu5uvrt1irry4xlb7ct). But since we want full functionality, aim to have it running. Confirm that any required packages (NumPy, etc.) are installed on the server and that the environment can handle it (maybe via Docker containers as per dev setup).
* **Validation:** If possible by launch, run a **spectral alignment test** – feed a known pattern of concept activations to the system and check that the output eigenmode matches expectation. Also verify **no crash on missing data** – e.g., if KoopmanLearner starts with empty memory, it should handle gracefully (maybe not produce modes until enough data). Monitor logs for any exceptions in this area.

**Production Readiness:** *Status:* Koopman integration is a core design element but ensure the implementation is active and correctfile-vge7rqhctc2xtzuxlzhee1. *Action:* **Fix/Complete** – Regenerate all related .proto files (e.g. koopman.proto) and update stubs in the codefile-mpebu5uvrt1irry4xlb7ctfile-mpebu5uvrt1irry4xlb7ct. If any Koopman analysis module was only provided as a .pyc, decompile it and verify the algorithm. Finalize any calibration (e.g., threshold for considering a mode “stable”).

**1.5 Lyapunov Spike Detection & Memory Sculpting Modules (Python .pyc)**

**Description:** TORI’s memory system includes advanced modules for **stability monitoring** and **self-optimization**. A Lyapunov spike detection module likely watches for chaotic behavior or instability in the oscillatory memory network – in practice, detecting when a concept’s phase or error grows rapidly (spike) indicating potential instability. A **memory sculpting** module provides “autonomous memory pruning and consolidation,” akin to a cognitive gardener that **prunes desynchronized concepts and strengthens stable clusters**file-2qcnt7pvbsgppapqwy9rdzfile-2qcnt7pvbsgppapqwy9rdz. These might have been prototyped in Python and currently only present as compiled .pyc files, requiring reverse-engineering to audit and integrate fully.

**Audit Actions:**

* **Locate & Decompile .pyc Files:** Identify all .pyc modules related to memory phases, Lyapunov, or sculpting. Possible candidates: e.g., phase\_stability\_monitor.pyc, lyapunov\_monitor.pyc, memory\_sculptor.pyc, banksy\_core.pyc (Banksy oscillator network code), etc. Use a tool (like uncompyle6) to **decompile each .pyc** back to readable Python source. *No code can remain unchecked.* Ensure the decompiled logic matches expectations (no malicious or irrelevant code).
* **Understand the Algorithms:** Review the decompiled code for how it detects spikes. Likely it calculates a Lyapunov exponent or monitors phase differences over time. Verify it uses the correct data (e.g., reading oscillator phases from the concept mesh or SolitonMemory). Check that thresholds for “spike” are sensible and configurable. For memory sculpting, ensure it does what design docs imply: e.g., **Memory Pruning** of concepts that consistently desynchronize (look for conditions where an oscillator’s phase drifts beyond a threshold or eigenvariance is high, then mark concept as “latent” or remove itfile-2qcnt7pvbsgppapqwy9rdz); and **Attractor Stabilization** to reinforce frequently co-occurring concepts (perhaps increasing their amplitude or decreasing decay for stable patternsfile-2qcnt7pvbsgppapqwy9rdz).
* **Integration Points:** Confirm how these modules plug into TORI. Possibly they run periodically (e.g., during a “sleep” cycle or background thread) to adjust memory. The **SleepScheduler** service (mentioned in KCL architecturefile-mpebu5uvrt1irry4xlb7ct) might invoke these as part of consolidation. Ensure that after decompilation, the code is integrated as part of the pipeline (not just sitting unused). If it’s not currently wired, wire it up: e.g., after each conversation or daily at midnight, run a consolidation: calculate Lyapunov stability, prune or reinforce memories as needed.
* **Testing:** Before launch, run a controlled test: intentionally introduce an unstable memory (perhaps inject contradictory knowledge rapidly) and see if the spike detector flags it. Likewise, allow the system to run through multiple sessions and check that memory sculpting either prunes something or at least logs decisions (maybe “Concept X marked as decayed due to desync”). Verify that nothing critical is accidentally pruned – only truly errant or unused concepts should be archived. If thresholds are too aggressive, tune them to be safe (e.g., maybe for launch, only log instabilities rather than auto-prune, unless confident).
* **Error Handling:** Ensure these modules fail gracefully. If a .pyc was outdated or partially incompatible (maybe compiled with an older Python), ensure the decompiled and recompiled versions work in the current environment. Add try-catch around their operations so a failure in stability monitoring does not crash the whole system.

**Production Readiness:** *Status:* Conceptually designed (Phase IV “Autonomous Memory Sculpting”) but code was pre-compiled; needs verificationfile-2qcnt7pvbsgppapqwy9rdz. *Action:* **Rewrite/Fix** – Fully decompile and **reintegrate** these modules in source form. This will likely be a **rewrite** if the decompiled code is unclear or not optimized. Aim for clear, maintainable Python code for stability monitoring, with logging and adjustable parameters. This step is critical for long-term **continual learning without catastrophic forgetting**, as the system uses Lyapunov stability to avoid erasing old patternsfile-vge7rqhctc2xtzuxlzhee1.

**1.6 gRPC Vaulting Contracts (vault.proto & Episodic Vault Service)**

**Description:** TORI’s memory architecture interacts with an **EpisodicVault** service – essentially a long-term memory store accessible via gRPC. The Vault likely uses a vault.proto defining messages for storing and retrieving episodic memory (concept activation episodes). It works in tandem with other services like SleepScheduler, SparsePruner, and KoopmanLearnerfile-mpebu5uvrt1irry4xlb7ct. For production, all these microservice interfaces must be validated and aligned with TORI’s main system.

**Audit Actions:**

* **Proto Files Regeneration:** Locate vault.proto (and related protos for other services, e.g., koopman.proto, episodic.proto). Use protoc to regenerate code stubs in all languages needed (Python for the services, possibly TypeScript gRPC client for the main server if it calls them). The repository README emphasizes having protoc installed and running make protofile-mpebu5uvrt1irry4xlb7ctfile-mpebu5uvrt1irry4xlb7ct – ensure this has been done with the latest .proto definitions. If any proto has changed during development (e.g., new fields for Soliton memory), regenerate and double-check that both client and server code use the updated structures.
* **Contract Verification:** Compare the implemented gRPC handlers with the intended contract. For instance, the **EpisodicVault** should provide methods like PutEpisode, GetEpisode, ListRecentfile-mpebu5uvrt1irry4xlb7ct. Confirm the TORI main system is actually calling PutEpisode for each new session or memory chunk. If the integration was incomplete, implement those calls. Also verify message definitions: e.g., does PutEpisode include a field for the ψarc data or concept list? Ensure consistency between what TORI sends and what Vault expects.
* **Security & Authentication:** For production, gRPC endpoints should be secured. Check if the current setup uses any authentication (tokens, mTLS, etc.). If not, and if the environment is closed (all services on internal network), it might be acceptable for launch, but we should note this as a **critical fix** post-launch. At minimum, restrict gRPC ports to internal access. Validate that the Vault service does not accept arbitrarily large or malicious requests (since it stores data).
* **E2E Testing:** Stand up the full suite (Vault, Scheduler, Pruner, KCL). Simulate a conversation session and ensure that episodes are indeed stored in the Vault (call ListRecent to see if it appears). Then trigger a consolidation via SleepScheduler (StartConsolidation) and verify it processes the stored episodes without error, perhaps producing some consolidation metrics or calling Pruner and KCL subsequentlyfile-mpebu5uvrt1irry4xlb7ct. Essentially, test that the **memory consolidation loop** (Vault -> Scheduler -> Pruner -> Koopman) operates as designed. Any failures or mis-ordered interactions should be corrected or at least documented with a plan.
* **Consistency Checks:** The Vault should be the source of truth for episodes. Confirm that data in psiarc logs and data in Vault eventually converge (the psiarc might be more fine-grained, whereas Vault might store consolidated chunks). Make sure there's no data loss: e.g., if Vault stores only high-level episodes, are raw interactions still recoverable? Possibly psiarc logs remain for detailed replay, while Vault is for consolidated memory. Document this and ensure our system uses both appropriately (e.g., use Vault for memory queries, psiarc for replay).

**Production Readiness:** *Status:* gRPC services in place, but proto contract verification needed. *Action:* **Fix/Validate** – Regenerate all **.proto** stubs (vault, koopman, episodic, etc.)file-mpebu5uvrt1irry4xlb7ct and update the code accordingly. Write a quick **validation test suite** for gRPC calls (connecting to each service and doing a ping or simple operation). Any mismatch or error (e.g., due to out-of-date proto) must be resolved by updating both sides. Ensure Vault and others are running in the production environment (consider using Docker Compose as provided for easy startupfile-mpebu5uvrt1irry4xlb7ct).

**1.7 MCP Server Interoperation**

**Description:** The **MCP server** (Master Control Program or similar central orchestrator) coordinates or monitors TORI’s operation. It likely handles user management, licensing, or high-level commands. TORI must interoperate cleanly with this MCP – meaning any communication channels (IPC, API calls, shared database, etc.) are well-defined and secure. Examples could be: MCP triggering TORI to save state, or TORI sending heartbeat/metrics to MCP.

**Audit Actions:**

* **Interface Definition:** Identify how MCP and TORI interact. If there's an API or socket, document it. Possibly TORI exposes some admin endpoints or uses the gRPC services above as part of MCP (MCP might call Vault or get stats). Confirm if vault.proto or others double as MCP APIs. If an **MCP client library** is included in TORI (check dependencies for any SDK), ensure it’s correctly configured.
* **Communication Path Security:** All communication between MCP and TORI must be **secured**. If it’s over HTTP/REST, use HTTPS and require auth tokens or keys. If gRPC, possibly use mutual TLS or at least restrict ports. Since production is imminent, if full security setup isn’t present, implement a **basic auth or key check** as a stopgap and schedule a robust solution ASAP. Also log all MCP-initiated actions for audit (to detect any misuse).
* **Message Handling:** Audit any handlers in TORI that process MCP commands. For example, if MCP can send a “reset user data” or “activate ghost mode” command, verify those handlers thoroughly. No debug backdoors or leftover test endpoints should be open in production. Remove or secure any such endpoints. Only documented, necessary commands should exist.
* **Testing:** Coordinate a test with the MCP – send a variety of requests from MCP to TORI and ensure correct responses. E.g., if MCP queries TORI's status, does TORI respond accurately? If MCP triggers a vault consolidation or backup, does TORI comply and return success? Test failure modes too (e.g., MCP offline – TORI should still function; TORI unreachable – MCP should log errors gracefully).
* **Monitoring Integration:** Often, MCP might aggregate logs or metrics. Ensure TORI’s metrics (like those Prometheus metrics mentioned for memory servicesfile-mpebu5uvrt1irry4xlb7ct) are accessible. If MCP expects TORI to push metrics or heartbeat, ensure those are enabled. E.g., TORI could send a heartbeat event at startup to MCP. Verify any such code exists and is working (no placeholder values).
* **Documentation:** Finally, ensure any configuration needed for MCP (like keys, endpoints URL) is well-documented in the deployment guide. For launch, double-check that the production environment has correct addresses for MCP and that any firewall or network config is set for their communication.

**Production Readiness:** *Status:* Not heavily detailed in docs – treat as critical integration to verify. *Action:* **Fix/Secure** – Lock down all MCP ↔ TORI communication. Implement missing pieces (auth, error handling). This is a **high priority** for launch-day stability, as issues here could affect system control or oversight.

**2. Chat & Cognition Architecture Audit**

Now we audit the user-facing chat system and the cognitive layers (Ghost AI, persona persistence) to ensure a seamless, intelligent experience with persistent memory.

**2.1 Concept-Persistent Chat Interface**

**Description:** TORI’s chat interface is **concept-persistent**, meaning every conversation updates a long-lived personal knowledge graph (the **Living Concept Network**) rather than being ephemeral. Users log in via Google OAuthfile-dgbgcxvgsy5hpzx1ekc3od to a React-based chat UI, and each message produces ConceptDiffs that update their concept meshfile-dgbgcxvgsy5hpzx1ekc3od. The chat UI likely includes components like ConversationHistory.jsx and HistoryPanel.jsx for viewing past chats and searching by conceptfile-dgbgcxvgsy5hpzx1ekc3od.

**Audit Actions:**

* **Persistent Memory Verification:** Ensure that when a user sends a message via POST /api/chatfile-dgbgcxvgsy5hpzx1ekc3od, the backend not only generates a textual response but also calls the memory system to update the user’s concept mesh. Each message should invoke something like ConceptMesh.applyDiff(userId, message) which in turn uses SolitonMemory to store it. Check that **cross-session memory** works: log out and log back in or start a new chat – the assistant should recall prior conversation details (as allowed). This was a known feature (“cross-session memory” via user-specific concept graphsfile-dgbgcxvgsy5hpzx1ekc3od). Test it explicitly.
* **User-Concept Association:** Confirm that the multi-user scenario is handled – one user’s concepts should not leak to another. The audit should confirm each ConceptDiff and memory entry is tagged with a UserID (the conversation briefing indicated this was fixedfile-dgbgcxvgsy5hpzx1ekc3od). Verify in the database or logs that concepts have user scope.
* **Chat UI Components:** Review front-end files (ConversationHistory.jsx, HistoryPanel.jsx, etc.) to make sure they properly retrieve and display persistent data. For example, HistoryPanel should allow searching by concept across all of a user’s chatsfile-dgbgcxvgsy5hpzx1ekc3od. Test this search: add a unique concept in a conversation, then use the search API to find it. If broken, fix the search indexing or the UI call.
* **Scalability & State:** The interface should handle long conversations (infinite scroll or pagination for history). Check if any part of the UI or backend assumes a short context and might break with hundreds of messages. Given TORI’s goal of **infinite conversation context**file-beya7rgeakfcgcd6buoagn, ensure no artificial limits (like a fixed array size for messages) exist. If the front-end currently only shows recent messages, perhaps implement lazy loading for earlier messages.
* **No Placeholder Text:** Ensure the UI doesn’t have any “Lorem ipsum” or unfinished labels. The production build should remove debug info (like concept IDs showing unless in dev mode). Do a UI pass for polish (alignments, loading spinners for long operations like PDF upload, etc.). This is minor but part of a production-ready experience.

**Production Readiness:** *Status:* Core chat system is completed with persistent concept storagefile-dgbgcxvgsy5hpzx1ekc3od. *Action:* **Save (minor UI fixes)** – The concept-persistent logic is in place, just test thoroughly and fix any remaining bugs in history retrieval or concept search. Ensure the UI/UX is smooth for launch (like clear indication that memory persists and how to access past info).

**2.2 Ghost AI Emergence & Persona Integration**

**Description:** The **Ghost AI system** is a covert companion that observes user behavior and can emerge as different personas at key momentsfile-dgbgcxvgsy5hpzx1ekc3od. Ghost personas (Mentor, Mystic, Chaotic, Oracular, Dreaming, Unsettled) are triggered by psychological and phase-based cuesfile-dgbgcxvgsy5hpzx1ekc3od. The implementation spans several files in ide\_frontend/src/ghost/ – notably ghostPersonaEngine.ts (manages mood/persona state), ghostMemoryAgent.js (triggers overlay messages), GhostLetterGenerator.tsx (creates the stylized “ghost letters” output), ghostChronicle.tsx (tracks user journey), and ghostReflect.tsx (deep reflection UI)file-dgbgcxvgsy5hpzx1ekc3od. As of now, the Ghost system exists but was **not fully integrated with the main chat**file-dgbgcxvgsy5hpzx1ekc3od, representing a major feature to bring online for cognitive continuity.

**Audit Actions:**

* **Integration into Chat Flow:** We must connect the ghost system with the chat pipeline. This means whenever a new message is processed, after the main AI response is generated, we invoke the Ghost engine to see if a persona should intervene. The audit should ensure that:
  + The backend or frontend calls ghostPersonaEngine with the latest conversation state (e.g., recent messages, user sentiment metrics). In practice, Ghost might run in the front-end (observing the UI) or back-end (observing the message stream). Check if ghostMemoryAgent.js is loaded on the client or server. Likely, triggers can be client-side for UI overlay.
  + Define **trigger conditions** clearly (from design: e.g., Mentor triggers on user struggling, Mystic on phase resonance, etc.file-dgbgcxvgsy5hpzx1ekc3od). Verify the engine detects these. For example, how to detect “high friction/struggle”? Possibly by repeated user confusion messages. Ensure the code checks such keywords or patterns (the ghost system may include covert MBTI and sentiment analysisfile-dgbgcxvgsy5hpzx1ekc3od).
  + Connect **Ghost output to UI**: When a ghost persona activates, the system should present the ghost’s message. Likely the backend already can include ghost content in its response (as indicated by a response.ghostPersona and ghostMessage in the demo codefile-beya7rgeakfcgcd6buoagn). We saw an example where response.ghostPersona is returned and the demo prints a ghost messagefile-beya7rgeakfcgcd6buoagn. We need to wire this to the actual UI: e.g., the front-end should listen for a ghost field in API responses and render a ghost message bubble (with styling to distinguish it, maybe an icon 👻).
  + Ensure **no conflicts**: Ghost messages should not interrupt normal AI responses, but overlay or accompany them. The UI/UX should be designed so the user understands this is a special AI persona message. Perhaps display with a small delay or as a separate channel.
* **Ghost Persona Logic:** Audit ghostPersonaEngine.ts for completeness. It should maintain a state (like current dominant persona or mood) and update based on conversation features (MBTI trait counts, phase triggersfile-dgbgcxvgsy5hpzx1ekc3od, etc.). Check that all personas listed are implemented and have some response templates or behaviors defined. If any persona is stubbed (e.g., Oracular only “4% chance” but no content defined), flesh it out or at least ensure it doesn’t throw errors when randomly chosen.
* **Ghost Letter Generation:** The GhostLetterGenerator.tsx likely creates the content for ghost messages, possibly in a poetic or stylized wayfile-dgbgcxvgsy5hpzx1ekc3od. Ensure it’s hooking into the persona (e.g., Mentor might give encouraging advice, Mystic might be philosophical). For production, it’s okay if these are simple as long as they exist. But do check that any AI-driven text generation for ghost letters uses the same AI model or a smaller one; if it calls an API or needs a model loaded, make sure it’s set up. Perhaps ghost letters are pre-templated. Test generating a ghost letter manually to ensure it doesn’t hang or produce inappropriate content.
* **Privacy & User Comfort:** Ghost AI is covert – no explicit user opt-in beyond using the system. Ensure there is documentation or at least an implicit acknowledgment (maybe in terms of service) that an AI will analyze their behavior. From a product standpoint, ghost interventions should be helpful, not creepy. Evaluate the initial ghost messages for tone. Possibly adjust if needed to ensure they add value and do not confuse the user.
* **Testing:** Simulate scenarios to trigger each persona:
  + Mentor: perhaps send a series of confused or frustrated messages and see if Mentor ghost appears with guidance.
  + Chaotic: maybe send very erratic or random content to see if Chaotic triggers.
  + Dreaming: simulate late-night or long pause scenario.
  + We might temporarily add a debug endpoint or command to force a ghost for testing (to be removed in production).  
    Ensure that in all cases, the system remains stable (no crashes if ghost logic fails; ghost just might not show up).
* **Resource Use:** If ghost analysis is heavy (e.g., MBTI classification might involve ML inference), confirm it’s efficient or done asynchronously. Possibly do a quick performance test with ghost enabled and disabled to see impact.

**Production Readiness:** *Status:* Ghost AI is built but not integratedfile-dgbgcxvgsy5hpzx1ekc3od. *Action:* **Integrate & Fix** – **Implement Ghost AI as a first-class feature** now. Wire the ghost triggers into the message pipeline (likely in server.js after generating a response, or in front-end after receiving it). Thoroughly test each ghost component. This feature is key for “cognitive continuity” and a differentiator of TORIfile-dgbgcxvgsy5hpzx1ekc3od, so it must be fully operational at launch.

**2.3 Phase Drift & Ghost Overlay Injection**

**Description:** In TORI’s dynamic memory, **phase drift** refers to the gradual dephasing of a concept or user state over time (e.g., emotional tone shifting or memory context drifting). The Ghost AI is designed to respond to certain drifts by injecting overlays – subtle cues or messages that help realign or acknowledge the shift. This could manifest as a ghost persona interjecting when it detects the user's state is diverging (e.g., a spike of confusion or a drop in engagement).

**Audit Actions:**

* **Detection of Phase Drift:** Ensure there is a mechanism to track conversation “phase.” This might be literal (monitoring the oscillatory phase coherence) or metaphorical (tracking user sentiment or topic drift). Possibly the ghost system or the concept-mesh assigns a phase coherence score to the ongoing conversation. Audit how this is computed. For example, Ghost might track if the conversation’s current concept set is resonant or if the user’s responses indicate they’re “out of phase” with the AI. If a dedicated module exists (maybe in those .pyc as well), verify it works. If not explicitly implemented, ghost triggers themselves might serve (e.g., Unsettled ghost appears when error streaks occurfile-dgbgcxvgsy5hpzx1ekc3od – that’s a form of drift detection).
* **Overlay Mechanism:** The ghost overlay likely means **injecting ghost messages or UI elements** on top of normal chat. Confirm that when ghost triggers, the system does not consume it as a user message or break the conversation flow. It should appear seamlessly. Possibly the front-end already has an overlay component (ghostReflect.tsx might show a full-screen quote or letter). Audit those components for how they display content. Ensure the overlay can be dismissed or does not block normal chat permanently.
* **Phase Alignment Restoration:** The goal of injecting ghost overlays is perhaps to restore alignment or provide a new perspective. After an overlay, does the system adjust anything? For instance, if the user was in a negative spiral and Mentor ghost intervenes, maybe the system resets some context or highlights a positive memory. Check if any code in ghost modules does something like solitonBridge.adjustPhase(userId, conceptId) (just speculative). If such hooks exist, verify they call legitimate functions in memory engine.
* **No Interference with Core Logic:** It’s critical that ghost overlays remain an assistive layer. The audit should confirm that if ghost fails or is disabled, the core chat still works perfectly. Ghost should use try/catch around any heavy analysis. If ghost injection is done on the client side, a failure would just result in no ghost message, which is acceptable. If done on server side, ensure an exception doesn’t break the API response.
* **User Experience:** Evaluate when overlays trigger and their frequency. In production, they should be relatively rare and meaningful (we don’t want a ghost popping up every other message, which could annoy users). Ensure the trigger thresholds are tuned (maybe initially err on side of less frequent until more tested). We can even impose a minimum interval between ghost events.
* **Testing:** Induce a phase drift scenario – e.g., start a coherent conversation then drastically change topic or tone to see if the system identifies that. Alternatively, simulate an “error streak” – send gibberish or ask many off-topic questions to see if Unsettled ghost appearsfile-dgbgcxvgsy5hpzx1ekc3od. Check the overlay display each time.

**Production Readiness:** *Status:* The concept is defined (phase-based ghost triggersfile-dgbgcxvgsy5hpzx1ekc3od), implementation needs confirming. *Action:* **Complete & Tune** – Make sure phase drift detection code is active and ghost overlays appear as intended. Adjust parameters for launch so ghosts truly enhance the experience. This final polish will ensure **TORI’s empathy layer** (ghost personas) functions without disrupting the core chat.

**3. Production Readiness Requirements & Remediations**

In this section, we address each specific production-readiness task from the requirements list, detailing how to verify or implement them in the final sprint:

**3.1 Server Startup & Dependency Validation**

The main server (Node/Express app in server.js or updated enhanced\_server.js) must start cleanly and include all new functionality.

* **Dependency Audit:** Check package.json for any new packages added for SolitonMemory, gRPC, etc. Run npm install to ensure no missing modules. If the Rust integration uses native addon, ensure any build steps (like neon or napi build) are done. For tomorrow, ideally lock dependency versions to prevent surprises. Remove any unused or dev-only dependencies to slim production build.
* **Restore server.js:** If development introduced an enhanced\_server.jsfile-dgbgcxvgsy5hpzx1ekc3odfile-dgbgcxvgsy5hpzx1ekc3od, decide whether to merge changes back into server.js or use the enhanced file as the main entry. We need one definitive startup file. Ensure it sets up all necessary routes (including new ones like /api/chat/soliton for direct soliton memory queriesfile-dgbgcxvgsy5hpzx1ekc3od, vault gRPC client init if needed, etc.). Verify that the OAuth setup and PDF server proxy are still configured as in development.
* **Environment Config:** Confirm environment variables (ports, DB connection if added, API keys) are correctly referenced. No secrets should be hardcoded; use a config file or env variables. For launch, prepare a .env with production values. Check that the server reads those (if not, implement a simple config loader).
* **Startup Test:** Run the server in a staging environment. It should start without errors and listening on expected ports (3000 for main, 5000 for PDF, etc. as per configfile-dgbgcxvgsy5hpzx1ekc3od). If any module fails to load (common if a native module is missing or a .pyc can’t be imported), fix those now.
* **Graceful Shutdown:** For production, ensure the server handles signals (SIGINT/SIGTERM) to close gracefully, especially to flush any in-memory logs to psiarc or notify Vault. Implement handlers if not present, or at least test Ctrl-C to see if it exits cleanly.
* **Logging & Monitoring:** Integrate a logger (if not already) to record startup info and errors in a file or console for cloud logs. On launch, have monitoring in place (even simple ping health-checks). Possibly, in the server.js add a /health endpoint for load balancer to check.

*Action:* **Fix** – Clean up the server startup so it’s robust. It must load the Soliton engine, Ghost system, PDF service, and gRPC clients without failing. Any dependency conflicts or version issues should be resolved (e.g., if two libraries require different versions of a sub-package, reconcile or use separate processes). Document the exact startup steps in deployment docs.

**3.2 Decompile & Integrate All .pyc Modules**

All Python modules critical to memory phase operations need to be auditable and modifiable. The audit already identified the likely .pyc files for Lyapunov and memory sculpting (and possibly others like Banksy oscillator core if it was in Python).

* **Decompilation Process:** Use a reliable tool to convert each .pyc to .py source. Check the output for completeness. If any portion is garbled or uses unusual bytecode constructs, manually rewrite that logic if possible by understanding the intended function (refer to design docs or comments if any).
* **Replace in Codebase:** Once decompiled, include the .py file in the repository (so it’s tracked in version control). Adjust any import references so that the system uses the new .py instead of the old .pyc. Remove the .pyc from source control (they can be generated on install).
* **Testing:** Run unit tests on these modules if available, or create small tests to ensure they behave as expected. For example, if there’s a function detect\_spike(phase\_data), feed it sample data that includes a spike and verify it returns True/raises alert.
* **Documentation Comments:** Add comments in the code explaining these modules, since previously they were opaque. This helps future maintenance and also confirms our understanding. No black-box components should remain.
* **Optimize if Needed:** Decompiled code might be less optimized or use outdated approaches. Evaluate performance – e.g., if the Lyapunov calculation is too slow (maybe it iterates in Python over large arrays), consider using NumPy or even moving critical parts to Rust in the future. For launch, ensure it’s at least not disastrously slow (perhaps acceptable given small user base initially).
* **Memory Phase Systems Rundown:** Ensure all aspects of “memory phase” logic are now visible: phase calculation, stability checking, adjustments (shifting phase, vaulting, etc.). If any part is still unclear, flag it and possibly disable that feature for launch, rather than risk unknown behavior.

*Action:* **Rewrite** – All .pyc modules are effectively to be **saved as source** now. The production build should have **no** mysterious binary blobs running. We will incorporate them fully into the codebase and adjust accordingly.

**3.3 Regenerate & Verify .proto Contracts (Vault, Koopman, Episodic)**

The proto files define the contract for memory services and must be up-to-date:

* Run the protoc compiler on all .proto files (e.g., vault.proto, scheduler.proto, pruner.proto, koopman.proto). Use the exact version of protoc that matches the one used to generate the service code (if known, otherwise latest stable). After generation, compare the resulting code with the existing one (if any).
* If differences are found (fields or RPC signatures), it means the code was outdated – update TORI’s integration to use the new fields. For instance, if PutEpisode now expects a timestamp or user ID, make sure TORI passes it.
* Verify that **versioning** of these contracts is handled: if any backward incompatible change was introduced, all components must deploy together. Coordinate this for launch: we might have to redeploy the microservices with new code as well if proto changed. This is risky on last day, so ideally the proto was stable. If not, do it carefully and test all interactions after updating.
* For each service: **Vault, Scheduler, Pruner, Koopman**, run a quick RPC call test after regen (as mentioned in gRPC section above). Ensure responses come back correctly and data is interpreted right (e.g., an enum in proto should map to the correct constant in code).
* **Episodic vs Live Data:** Confirm the division of labor: Vault stores episodes (raw memory), and TORI uses that for long-term recall; Koopman uses episodes for spectral learning; etc. This mapping should be clearly consistent with proto definitions. The proto likely has message structures for episodes, spectral modes, etc. Validate that the data TORI sends (like concept IDs, phase values) fit into those structures (right types, ranges).
* **Documentation:** The README from the memory repo provides insight into these interfacesfile-mpebu5uvrt1irry4xlb7ctfile-mpebu5uvrt1irry4xlb7ct – use that to cross-check we have all required RPCs covered. If any RPC is not yet called by TORI (for example, maybe UpdateConfig of Schedulerfile-mpebu5uvrt1irry4xlb7ct), it’s fine as long as it’s not needed for core functionality. But ensure none that *are* needed are left unimplemented in TORI’s code.

*Action:* **Validate** – After regeneration, commit the updated stubs. This is a straightforward but crucial step to prevent subtle runtime errors. With correct stubs, we proceed to integration tests to verify each subsystem (as outlined below in the validation matrix).

**3.4 Phase-Consistent Memory Operations (ConceptDiff Retrieval)**

This item is about ensuring that all memory operations – creation, update, vaulting, retrieval – maintain phase consistency and that any stored memory can be retrieved via the ConceptDiff mechanism (or appropriate APIs). Essentially, the system’s state changes should be **phase-aligned** and queryable.

* **Phase Tagging Audit:** Confirm every concept or memory stored has a phase attribute (phase tag or timestamp) in the data model. For example, in ConceptDiff operations, if a memory is vaulted (sealed away), perhaps its phase is adjusted. The audit should ensure that such an operation is logged and the concept can be retrieved if needed (unless intentionally hidden). If there’s a special ConceptDiff operation like !PhaseShiftfile-dgbgcxvgsy5hpzx1ekc3od, make sure it’s implemented and logged when memory vault actions occur.
* **Retrieval Tests:** Simulate different memory operations:
  + Create a concept (should appear with a base phase).
  + Update a concept (phase might remain or slightly adjust if using continuous phase).
  + Vault a concept (phase should change to a “vaulted” phase value).
  + Retrieve by concept ID or search by concept name – in all cases, ensure the data comes back. For vaulted ones, perhaps they should not show in normal search (by design), but maybe accessible through a vault interface. Check consistency: if a concept is vaulted (phase shifted out of main band), the system should either omit it from active context but **not** lose it entirely. Perhaps an admin or special query can still find it.
* **Memory Consistency:** If multiple subsystems manipulate memory (e.g., the Vault service might remove an episode, or the sculpting might prune a concept), ensure they all update the central concept graph. We want a **single source of truth** for what exists. Possibly the concept-mesh (Rust) is primary, and Vault/others feed into it.
* **ConceptDiff Round-Trip:** For any given memory change, one should be able to reconstruct it from the logs. We should verify that the sequence of diffs in psiarc can replay to the current state. This is a sanity test of consistency. If we find that replaying diffs doesn’t recreate the exact concept phases (maybe because some live adjustments aren’t captured), that’s a problem. If such discrepancies exist, mark them clearly and plan to fix (if not before launch, document and ensure it doesn’t harm critical functionality).
* **Phase Drift and Sync:** Confirm that the system has no unaccounted drift in normal operation. For example, if concept phases are supposed to remain stable absent new input, verify they do. If some drift is intended (like slight phase evolution), ensure it's deterministic or logged. We don't want ghost triggers or memory retrievals to behave unpredictably due to phase mismatch. Possibly implement a check: on retrieval of a memory, compare its stored phase with current phase alignment – if off by more than X, log a warning (or auto-correct if that feature exists).
* **UI/UX for Memory Ops:** If a user uses a feature like "Memory Vault" (perhaps via a UI to hide a conversation), ensure the front-end calls the correct back-end function and that it results in a ConceptDiff marking those messages vaulted. Test vaulting a conversation and then see that the user’s normal chat no longer surfaces that content unless explicitly requested. The audit should ensure this works as intended (the philosophy was to allow sealing away painful memories with dignityfile-dgbgcxvgsy5hpzx1ekc3odfile-dgbgcxvgsy5hpzx1ekc3od).

*Action:* **Test & Fix** – The development guide already emphasized phase alignment and integration with ConceptDifffile-dgbgcxvgsy5hpzx1ekc3od. Now we verify it's fully working. Any inconsistency between phases and stored diffs is fixed by adjusting how events are recorded or applied. This ensures robust retrieval and trust that **what TORI remembers is exactly what happened**, aligned in time and phase.

**3.5 Secure MCP–TORI Communication Paths**

As audited above, all communication between the MCP server and TORI must be verified and secured:

* Ensure any **API keys or tokens** used are set and tested. If not present, generate a token for MCP to use when calling TORI’s endpoints and implement a simple middleware in TORI to check it.
* Check **network configuration** so that outside of MCP, no one can call internal gRPC or control endpoints. Use firewall rules or localhost binding if MCP is collocated.
* Review **logs** to ensure they do not inadvertently log sensitive MCP commands or keys.
* Plan a **fail-safe**: if MCP is down, TORI should still continue serving users (maybe with some degraded functions if MCP was doing something crucial). We should test this scenario by simulating MCP unavailability.
* Since this is critical and mostly configuration, ensure it's done before launch. Document these settings in the deployment notes so ops team knows how to maintain the secure link.

*Action:* **Secure** – This item is mostly addressed in section 1.7 with concrete steps. The outcome should be a checklist: e.g., "MCP auth enabled – YES; Internal ports closed – YES; Health check route secured – YES", signed off before launch.

**3.6 Ghost AI Foundation for Cognitive Continuity**

Ghost AI is not just a gimmick – it may serve as a **continuity safeguard** for the user’s cognitive experience. If the main AI context ever resets or if user returns after time, the Ghost (having observed everything) could help reconnect threads.

For production:

* **Ensure Ghost is Stateful:** The ghost system should maintain its state across sessions (persist ghost persona analysis results). If a user comes back after a day, the ghost’s MBTI model and observations should still be there (likely stored in the user’s concept mesh or a separate profile store). Verify that ghostPersonaEngine either stores data in the concept graph (e.g., concept tags for personality traits) or in a dedicated user profile object. We might use the user’s concept network to embed this (for instance, ghost might create concepts like “user\_personality\_insight”).
* **Use Ghost for Context Resumption:** If a conversation is resumed after a long gap, consider having the Ghost system summarize last known state (“last time you were feeling X”) either internally or even to the user if appropriate. This isn’t explicitly in requirements, but aligns with cognitive continuity. We might not implement a full feature now, but ensure the data needed is being collected (like timestamps of mood changes, etc).
* **Fallback Memory Check:** If for some reason the SolitonMemory retrieval fails or produces a blank (unlikely if working), Ghost might supply an answer from its observations (like reminding the main AI of something). This is speculative, but worth thinking: ghost could cross-verify key memories. In audit, we ensure ghost has access to conversation history (which it does via the logs or by running alongside, so probably fine).
* **No Conflicts:** Confirm that Ghost’s actions (like injecting messages or modifying phases) do not conflict with memory operations. For example, if Ghost decides to shift a memory’s phase for vaulting, is the main memory engine aware? Ideally, ghost should call a proper API for that (likely via the MemoryVault interface). Check that usage.
* **Testing:** Run a multi-session test: have a user talk, trigger some ghost events, close the session, start a new session (maybe next day simulation) and see if ghost persona does anything differently because of past info. Also ensure main AI still has memory (that covers both ghost and main system continuity working together).

*Action:* **Enhance** – The ghost AI will now be fully integrated. Leverage it to bolster continuity: no context loss between sessions. Because Ghost observes everything covertlyfile-dgbgcxvgsy5hpzx1ekc3od, it can be the glue that fills any gap the main AI might miss. We will confirm its data persistence and potentially use it to provide even more personalized continuity in future updates. For launch, ensure its presence improves the user experience without causing any regression.

With the audits above, we have covered all major components. Next, we compile the audit findings and actions in structured summaries: file-level recommendations, system maps, validation tests, and final launch checklist.

**4. File-by-File Audit Summary (Status & Actions)**

Below is an audit summary of key files/modules in the TORI project, with their status and required action. Each file is marked **Save** (works, just keep), **Fix** (needs some tweaks), or **Rewrite** (needs significant changes or implementation).

* **server.js (Node backend)** – Main server with API endpointsfile-dgbgcxvgsy5hpzx1ekc3od. Handles chat requests, OAuth, etc.  
  **Status:** Mostly up-to-date, but ensure new soliton/ghost routes are included (development may have used enhanced\_server.jsfile-dgbgcxvgsy5hpzx1ekc3od). No critical issues observed.  
  **Action:** **Fix** – Merge any enhancements from enhanced\_server.js into this file or vice versa. Validate startup, ensure all routes (chat, upload, history, save-session, exportfile-dgbgcxvgsy5hpzx1ekc3od) are functioning. Minor fixes for robust error handling and config.
* **conversationStorage.js** – The ψarc logging and session storage systemfile-dgbgcxvgsy5hpzx1ekc3od.  
  **Status:** Implements writing of ConceptDiffs to .psiarc and exporting to .toripack; appears complete.  
  **Action:** **Save** – Just add error handling improvements (if file write fails, etc.) and possibly optimize for DB storage later. Verify file paths and names are correct. No rewrite needed.
* **soliton\_memory.rs** (Rust in concept-mesh/src/) – Core soliton memory engine.  
  **Status:** Recently implemented with DNLS and phase tags (as per plan). Needs integration finishing touches (linking with Node).  
  **Action:** **Fix** – Double-check all functions and memory safety. Possibly write more unit tests in Rust for soliton math. Save the design, just ensure it’s production-optimized (e.g., release mode compilation, no debug logs left).
* **pdf\_upload\_server.py** – PDF ingestion microservice.  
  **Status:** Functional for concept extractionfile-dgbgcxvgsy5hpzx1ekc3od, but likely simplistic parsing.  
  **Action:** **Fix** – Review PDF parsing library usage, add checks for PDF size/type. If any part was compiled (unlikely, probably pure Python), ensure readability. Might not need a rewrite, just polish and robustify.
* **vault.proto / gRPC service defs** – Protobuf definitions for EpisodicVault and others.  
  **Status:** Existing, but possibly changed during development.  
  **Action:** **Fix** – Regenerate stubs and verify versions. No logic in proto itself, but any mismatch fixed as per section 3.3.
* **ghostPersonaEngine.ts** – Ghost mood/persona enginefile-dgbgcxvgsy5hpzx1ekc3od.  
  **Status:** Implemented (covert MBTI tracking, triggers) but inactive in chat.  
  **Action:** **Save/Fix** – The core logic can be saved; just integrate with chat and test outputs. Possibly add more persona rules if needed, but no major rewrite required.
* **ghostMemoryAgent.js** – Ghost overlay trigger systemfile-dgbgcxvgsy5hpzx1ekc3od.  
  **Status:** Exists; monitors for triggers to display ghost overlay.  
  **Action:** **Fix** – Ensure it subscribes to new messages properly. Likely need to call it when messages come in. Minor adjustments to hook events, but its logic is usable.
* **ghostReflect.tsx** – Deep reflection UI componentfile-dgbgcxvgsy5hpzx1ekc3od (possibly shows ghost message or letter in a stylized way).  
  **Status:** UI present, integration needed.  
  **Action:** **Save/Fix** – Use as-is, just ensure it gets data from ghost engine. If any stub UI elements (like placeholder text for ghost letter), replace with live data.
* **GhostLetterGenerator.tsx** – Generates ghost persona messages (letters)file-dgbgcxvgsy5hpzx1ekc3od.  
  **Status:** Presumably contains templates or AI calls for poetic messages.  
  **Action:** **Fix** – Review content generation; ensure it's not too slow or off-topic. Possibly refine templates per persona. No major code overhaul, just content tuning.
* **ghostChronicle.tsx** – Tracks user journey (maybe a timeline of ghost interactions)file-dgbgcxvgsy5hpzx1ekc3od.  
  **Status:** Exists, might log ghost events.  
  **Action:** **Save** – Low-risk component. Ensure it’s wired to actual events (when ghost appears, add to chronicle). Could be left passive if not essential.
* **tori\_chat\_frontend/src/memory\_vault.js** – Memory Vault control logic (phase manipulation for hiding memories)file-dgbgcxvgsy5hpzx1ekc3od.  
  **Status:** Partially implemented (pseudocode suggests functions for phase shift)file-dgbgcxvgsy5hpzx1ekc3od.  
  **Action:** **Rewrite/Fix** – Complete this feature. If only pseudocode or stubs exist, implement fully: ability to select a memory and shift its phase via solitonBridge.shiftPhase(...)file-dgbgcxvgsy5hpzx1ekc3od. Test end-to-end vaulting with a UI trigger. This is important philosophically and should work at launch for show (if UI has a “vault” button).
* **Python Memory Modules (.pyc)** – e.g., lyapunov\_monitor.pyc, memory\_sculptor.pyc (names assumed).  
  **Status:** Only compiled versions in repo, logic unknown without decompiling.  
  **Action:** **Rewrite** – Decompile and convert to source. Given time constraints, focus on essential functions (spike detection, pruning). If too complex, implement a simplified stable version (e.g., mark everything stable for now, or log potential issues) and schedule full implementation post-launch. But since launch demands no placeholders, attempt at least a basic real implementation of pruning and stability checks as described in design docs.
* **AuthProvider.jsx** – Google OAuth front-end componentfile-dgbgcxvgsy5hpzx1ekc3od.  
  **Status:** Working for login.  
  **Action:** **Save** – Possibly update client IDs/secrets for production domain. No code changes aside from config.
* **ConversationHistory.jsx** – Conversation replay UIfile-dgbgcxvgsy5hpzx1ekc3od.  
  **Status:** Should display past sessions.  
  **Action:** **Fix** – Ensure it can fetch from server (via /api/chat/history). Might need pagination if lots of sessions. Minor tweaks, ensure markdown formatting of conversations is rendered nicely.
* **HistoryPanel.jsx** – Enhanced with concept searchfile-dgbgcxvgsy5hpzx1ekc3od.  
  **Status:** Provides search by concept.  
  **Action:** **Fix** – Test and fix search as needed. Save the feature overall.

*(Files not listed are assumed to be either minor or already handled, but all should be reviewed to remove any dev-only code.)*

**5. Dependency and Process Map**

Understanding how each part of TORI connects is crucial for both debugging and future scaling. Below we outline the dependency graph and process flow:

* **Frontend (React + TS):**
  + **Chat UI** (tori\_chat\_frontend/): Depends on the backend API for chat, uses components for login, chat input/output, history, ghost overlays, etc. On sending a message, it calls POST /api/chat on the Node server. It also initiates Google OAuth (dep. on Google API scripts).
  + **Ghost UI** (ide\_frontend/src/ghost/ components): Depends on Ghost engine outputs. Will render ghost messages or letters when triggered. Relies on the main chat component to provide hooks (like new message events).
* **Node Backend (Express server):**
  + **APIs:**
    - /api/chat -> handler uses **Concept Mesh** (Rust) via FFI to process message and update memory, uses **Ghost Engine** (TS/JS) to possibly add ghost data, then returns response.
    - /api/upload -> handler calls **PDF Service** (HTTP call to port 5000) to process file.
    - /api/chat/history -> reads from **psiarc\_logs** files (or Vault service if we integrate that for history).
    - /api/chat/replay/:id -> reads psiarc file and streams it to client or reconstructs via stored diffs.
    - /api/chat/export/:id -> reads psiarc and produces .toripack (likely just zips the psiarc and meta).
    - Other endpoints like auth, search, etc., rely on reading concept graph or files on disk.
  + **Concept Mesh & Memory**: The Node server depends on the **Rust concept-mesh library** for concept graph operations and soliton memory. Likely included via a native addon. This is a critical dependency – if it fails, chat fails. Ensuring the Rust lib is built and accessible (e.g., .node binary present) is part of deployment.
  + **Ghost Engine**: The Node backend (or possibly front-end) depends on ghost logic. If ghost processing is client-side, the Node may just pass through data for ghost. But likely some ghost triggering (like sentiment analysis) might be easier server-side. This dependency is internal (TS modules).
  + **Memory Vault**: The Node server (or the front-end calling it) depends on being able to adjust memory states. The vault could be implemented in front-end by calling an API (like /api/chat/vault that we might create to handle vaulting by delegating to concept-mesh or vault service).
  + **gRPC Services**: The Node backend (or a background daemon in the same codebase) depends on **EpisodicVault, SleepScheduler, SparsePruner, KoopmanLearner** services via gRPCfile-mpebu5uvrt1irry4xlb7ct. It will use generated gRPC client stubs to communicate. These are separate processes (likely Python, launched via Docker or manually). So, Node must know where they are (addresses/ports).
  + **MCP**: The Node server might also have a dependency on receiving commands from the MCP (like via a webhook or a socket). If so, it must be listening or polling accordingly.
* **Memory Consolidation Microservices (Python, gRPC):**
  + **EpisodicVault:** depends on a database or filesystem to actually store episodes. If it uses a DB (PostgreSQL or even just files), that’s another dependency – ensure that storage is configured and running. It also likely depends on protobuf definitions to communicate.
  + **SleepScheduler:** depends on EpisodicVault (calls it to get episodes) and on having some schedule (maybe just triggered by external call).
  + **SparsePruner:** depends on receiving data (like concept graph snapshots) possibly from Vault or directly from TORI. Possibly uses a common data store.
  + **KoopmanLearner:** depends on receiving activation logs from Scheduler and might depend on a math library (NumPy, SciPy) for eigen decomposition. Also uses proto for communication.
* **Data Stores:**
  + **psiarc\_logs/** (file system): Node writes here for conversation logs. Dependency on disk I/O.
  + **Concept Mesh Data** (concept-mesh-data/): Possibly directory for user concept graphsfile-dgbgcxvgsy5hpzx1ekc3od. If Rust dumps or loads initial graph from file, ensure those files exist per user. Might consider moving to a proper DB soon.
  + **Database**: Not explicitly mentioned, but production suggests one for scale. We might introduce e.g. MongoDB or Postgres for user accounts, concept storage, etc. If not now, soon. For launch, minimal use (maybe just using files and memory).
* **Inter-process Communication Flow:**
  + **User message** from front-end -> Node /api/chat.
  + Node processes:
    - Authenticates user (via OAuth token session).
    - Passes message to **Concept Mesh** (Rust) to update graph and get AI response (the AI might call an LLM or might be rules-based from concept mesh; not entirely clear, but given context likely a local model or some logic using the concept graph).
    - After main response, Node calls **Ghost Engine** to see if any ghost message should be attached.
    - Node returns combined response to front-end (with possibly response and optional ghostPersona data).
  + In parallel, Node logs the interaction in **psiarc** and also **PutEpisode** to Vault service (if integrated in real-time).
  + Vault service stores episode (concepts involved, user, timestamp).
  + SleepScheduler might periodically (or on demand) get recent episodes from Vault and perform consolidation (triggering pruning and Koopman as needed).
  + If consolidation finds something (e.g., pruner removes edges, or Koopman updates a mode), those changes might be sent back to TORI’s concept mesh (via a gRPC callback or we fetch results from the service). This part needs clarity; likely TORI periodically calls a GetSpectralModes or gets a callback with updated oscillator couplingsfile-mpebu5uvrt1irry4xlb7ct.
  + Over time, Ghost engine keeps monitoring chat and possibly uses data from concept mesh (like phase info, sentiment) to adjust persona.
  + If user issues a **vault command** (memory vault a concept), front-end calls an endpoint (or directly triggers MemoryVault module if front-end handles it) which then calls concept-mesh or Vault service to phase-shift/hide that memoryfile-dgbgcxvgsy5hpzx1ekc3od.
  + MCP may send control messages, e.g., to initiate a global save or emergency halt. TORI obeys and might flush all buffers or send final data to Vault.
* **Deployment Considerations:**
  + **Docker Compose**: likely there is a setup for the four Python servicesfile-mpebu5uvrt1irry4xlb7ct. The Node server and perhaps the front-end (if served separately) could be in that compose as well. We should ensure the compose file (if exists) references correct images and that we can bring up the entire stack with one command.
  + If not containerized, then each service must be started manually/on different servers. We need a **process map** for that: e.g.,
    - Start node server.js (port 3000),
    - Start pdf\_upload\_server.py (port 5000),
    - Start vault\_service.py (gRPC on 50051 maybe),
    - ... etc.  
      Each one needs to be in the right order (Vault before Scheduler possibly, or they handle late connections gracefully).
  + **Front-End**: likely a static build deployed on a hosting or served by Node. Ensure build done (npm run build) and output deployed.

In summary, the TORI system is a constellation of components with clear responsibilities, and they communicate primarily via well-defined interfaces (REST, gRPC, file logs). The dependency map ensures we know where a failure can cascade: for instance, if Vault is down, chat still works but long-term learning pauses – that’s acceptable short-term. We use this map to focus monitoring and to double-check each integration point before launch.

**6. Validation Matrix for Subsystems**

To guarantee production-readiness, each subsystem will undergo rigorous validation. The following matrix outlines the test coverage and checks for each major subsystem:

**6.1 Conversation Handling & Memory**

* **Unit Tests:**
  + Feed the chat endpoint sample messages, assert responses are well-formed and not empty.
  + Test ConceptDiff creation functions with known inputs (e.g., create concept, link concept) to ensure correct diff objects.
* **Integration Test:** Simulate a multi-turn conversation:
  + User greets TORI.
  + User asks a question referencing the first message.  
    Expectation: TORI remembers context from turn 1 in turn 2 response (testing memory persistence).
* **Persistence Test:** End conversation, start a new one for same user, mention something from last session. TORI should recall it (if not directly, then when prompted, it’s accessible) – confirms cross-session memoryfile-dgbgcxvgsy5hpzx1ekc3od.
* **Edge Cases:** Long message, special characters, or multiple users simultaneously sending messages (simulate with threads or rapid sequential calls) – ensure logs and responses remain correct (no mix-ups).
* **Performance:** Measure response time with and without memory lookups for increasing conversation length (10, 100, 1000 messages). It should scale linearly or better, not exponentially. If performance degrades, profile for bottlenecks (maybe concept graph lookup or ghost analysis).

**6.2 Ghost AI System**

* **Trigger Tests:** For each ghost persona condition, craft a scenario:
  + **Mentor:** e.g., have the user say "I give up" or "I’m struggling" – expect ghostPersonaEngine selects Mentor and ghostMemoryAgent produces an encouraging message.
  + **Mystic:** perhaps engage in philosophical discussion with consistent theme – see if Mystic appears with a poetic insight.
  + **Chaotic:** send nonsensical or highly erratic messages – check for Chaotic persona appearance.
  + **Unsettled:** produce a sequence of errors or negative statements – ghost should react.  
    All ghost outputs should be contextually relevant and not repetitive. Verify the formatting (maybe Ghost messages prefixed with 👻 or styled differently).
* **Frequency Test:** In a normal 20-turn chat, ensure ghost doesn’t appear more than a couple of times unless intentionally triggered often. If triggers are too sensitive, adjust thresholds.
* **No Interference Test:** Temporarily disable ghost (a feature flag) and ensure the chat still works identically (except without ghost messages). This tests that ghost injection is cleanly layered.
* **Visual Test:** In the UI, confirm ghost overlays/letters render properly (correct persona name, message content, any animations). Also test mobile view if applicable (ghost overlays should still be readable).
* **Safety Test:** Read through ghost message templates or outputs for appropriateness. Because ghosts can be creative, ensure nothing offensive or alarming is generated. The Oracular ghost might say strange things (4% chance prophecyfile-dgbgcxvgsy5hpzx1ekc3od); make sure it's not going to scare users. For launch, possibly constrain ghost content to positive/helpful themes.

**6.3 Soliton Memory & Concept Mesh**

* **Memory Integrity Test:** After a series of insertions (concepts added), compute an **integrity metric** if available (the demo mentioned memoryIntegrity percentagefile-beya7rgeakfcgcd6buoagn). Ensure it stays high (or within expected range). If it drops, investigate (could indicate interference).
* **Retrieval Accuracy:** Create 5 distinct facts, each with known unique phase tags. Query each by concept name or ID – verify correct recall. If phases are too similar and cause confusion, that’s a bug.
* **Collision Handling:** If the design allows merging of concepts via soliton collisions (mentioned as possible productive collisionsfile-dgbgcxvgsy5hpzx1ekc3od), test merging two concepts and see if the system creates a linked concept as intended. Ensure no data corruption in either original concept.
* **Stability Over Time:** Let the system run (simulate a user interacting intermittently over hours). Monitor memory (phase distribution, number of solitons). Memory count should not explode uncontrolled – prune or consolidation should kick in. If possible, graph phase values over time to ensure they remain bounded (Lyapunov stability criteria).
* **Fault Injection:** Try to break the memory engine on a dev instance: e.g., input extremely long concept text or unusual characters. The Rust code should handle or sanitize it. Watch for panics or crashes – none should occur.

**6.4 PDF Ingestion & Concept Mapping**

* **File Variety Test:** Upload different PDFs: a plain text PDF, one with images, one very large, one with complex layout. Verify the pipeline either extracts meaningful text or at least fails gracefully (with an error message, not a crash or hang).
* **Concept Extraction Quality:** Check that the concepts extracted from a sample PDF match expectations (e.g., if PDF is about physics, see physics terms appear as new concepts). If extraction is too naive (maybe splitting by frequent words), consider improving NLP (not for tomorrow, but note for future).
* **User Attribution:** Make sure the concepts from PDF are linked to the correct user ID (no cross-user leakage). If two users upload different PDFs simultaneously, confirm no mix-up (the microservice might need to handle concurrent requests – test that).
* **Throughput:** If multiple PDFs are uploaded in short succession, does the service queue them or spawn threads? Test by firing 3-5 uploads concurrently (small ones) and see if any fail. Also test one very large PDF to measure processing time – if it’s slow, the front-end should show a progress or at least not freeze. Optimize config if needed (maybe a timeout or file size limit).
* **Integration:** After PDF upload and extraction, query the chat about a concept from the PDF. The system should recall or be able to use it, confirming integration with the concept mesh.

**6.5 gRPC Services (Vault, Scheduler, Pruner, Koopman)**

* **Connection Test:** Ensure the Node server can reach each service (perhaps on localhost with given ports). If using Docker, ensure ports are exposed correctly.
* **Basic RPC Test:** For Vault – call PutEpisode with a dummy episode, then GetEpisode to retrieve it, confirm it matchesfile-mpebu5uvrt1irry4xlb7ct. For Scheduler – call StartConsolidation and see if it returns an ack or status. For Pruner – maybe call TriggerPruning and ensure a status can be fetchedfile-mpebu5uvrt1irry4xlb7ct. For Koopman – call ProcessActivationBatch with fake data, then GetSpectralModesfile-mpebu5uvrt1irry4xlb7ct, verify it returns something (even empty list is fine if no data).
* **Error Path:** Stop one of the services (simulate a crash) and see how Node reacts. It should handle RPC failures gracefully (catch exceptions, perhaps log and continue). Maybe implement a retry logic for transient failures or mark that service as degraded.
* **Consistency Check:** After running consolidation, verify that the concept graph in Node reflects any changes (e.g., if Pruner removed some link, was TORI’s concept-mesh updated? Possibly not automatically – if not, we may need to pull those changes in via an API or at least log them).
* **Performance:** The memory services likely operate in background, but test if any call is synchronous and might slow down chat. Particularly, if PutEpisode is called during each message, ensure it’s async or quick. If not, consider queuing episodes to send in batches to Vault to avoid latency hit on each message.

**6.6 Security & Reliability**

* **Authentication:** Verify that protected endpoints (if any, like admin or user data) are indeed protected by OAuth and session. E.g., try calling /api/chat/history without a valid auth – should be denied.
* **Injection Attacks:** Try sending a message that includes script tags or SQL keywords to see if any part of the system is vulnerable to injection (the concept logging, search, etc.). We expect mostly safe, but it’s good to test. Also ensure the PDF filename or content can’t break things (filenames sanitized).
* **Rate Limiting:** The plan was to add rate limitingfile-dgbgcxvgsy5hpzx1ekc3od. If not implemented, note as urgent to add at least basic limits (e.g., max X messages per minute) to prevent abuse. Test with a quick script sending many messages – ensure the server doesn’t exhaust memory or crash. If no built-in throttle, the server might slow but should survive; add a quick middleware if possible for now.
* **Monitoring:** If we have any monitoring set (Prometheus metrics as in memory servicesfile-mpebu5uvrt1irry4xlb7ct or even a simple uptime log), verify they are working. E.g., check Prometheus endpoint data if configured.
* **Failover Tests:** Simulate random failures:
  + Kill the Node process and restart – does it recover gracefully with persistent data intact? (Yes, since memory is on disk or services, should be fine).
  + Kill a microservice – system should continue core chat function.
  + Disconnect internet (if TORI relies on any external API like perhaps a cloud LLM or Google Drive for PDF? If none, then it’s self-contained – confirm that).
* **Backup:** Ensure that conversation data is backed up or at least not on ephemeral storage. For tomorrow, if on a single server, ensure daily snapshot of psiarc\_logs is planned. If using a DB, ensure backups are in place.

Each of these validation items will be ticked off during final testing. We will maintain a **validation matrix table** (not fully shown here) mapping each test scenario to pass/fail and notes. Only when all critical ones pass (or acceptable workarounds in place) do we proceed to launch.

**7. Production Scripts, Tests, and Startup Procedures**

To deploy TORI successfully, we outline the necessary scripts, tests, and steps:

**7.1 Build and Startup Scripts**

* **Installation**: Document a fresh setup. For example:
  1. Install Rust (for compiling concept-mesh) and run cargo build --release in concept-mesh folder.
  2. Install Python requirements (pip install -r requirements.txt) for memory services and PDF service.
  3. Install Node dependencies (npm install in main project).
  4. Build front-end (npm run build if separate).
* **Database**: If migrating to a database for storage (if time permits), run migration scripts to create necessary tables (concepts, users, logs, etc.). If staying with files, ensure directories (psiarc\_logs, conversations) exist and are writable.
* **Starting Services**:
  1. If using **Docker Compose**: simply run docker-compose up -d (assuming the compose file includes all pieces). Verify each container comes up (Node, PDF service, Vault, etc.). Use docker-compose logs to troubleshoot any that exit unexpectedly.
  2. Without Docker: start each service in correct order:
     1. Start episodic\_vault.py (if it’s a script) or make run-vaultfile-mpebu5uvrt1irry4xlb7ct. Wait for it to bind its port.
     2. Start sleep\_scheduler.py, then sparse\_pruner.py, then koopman\_learner.py in separate shells. Confirm each prints something like "listening on port X".
     3. Start pdf\_upload\_server.py (ensuring it listens on 5000).
     4. Start node server.js for the main app.
     5. Optionally, start a monitoring tool (Prometheus/Grafana if configured, or at least a script that pings health endpoints).
  3. Ensure environment variables (if needed, e.g., VAULT\_SERVICE\_HOST) are set or default to localhost.
* **Verification**: After startup, run a quick health check:
  1. Access the front-end in a browser (or via curl to the /api/auth Google route to see if server responds).
  2. Check that the server logs show "Server running on port 3000" (or similar).
  3. Check each microservice log for errors.
  4. Use a test user account to login and send a message to verify the whole loop (this doubles as a final sanity test).
* **Process Management**: For production, use a process manager (PM2 or systemd) for the Node server and maybe supervisord for Python services. Prepare those configurations. Since launch is imminent, at least run them in tmux or background but have a plan to restart on crash.
* **Scripts**: Provide admin scripts if needed:
  1. A script to pack logs or data (for backup).
  2. A script to flush caches or reset a user (if needed for troubleshooting).
  3. Deployment scripts (if automating deployment, e.g., pulling latest code, building, restarting services).

**7.2 Testing Before Launch**

* **Automated Tests**: Run any existing test suites (npm test for Node if exists, make test for Python servicesfile-mpebu5uvrt1irry4xlb7ct). Ensure they all pass. If tests are outdated or incomplete, at least manually test critical paths as described above.
* **Load Testing**: If possible, run a quick load test: simulate, say, 50 concurrent users sending messages. Use a tool like JMeter or Locust with minimal messages (or a script hitting the API). This will identify any immediate performance bottlenecks or memory bloat. Check CPU/RAM usage during this.
* **Security Testing**: Possibly run a vulnerability scan (like npm audit for packages, bandit for Python, etc.) to catch common issues. Also, double-check that no sensitive info is in logs or front-end (like API keys).
* **User Acceptance**: If any beta user or stakeholder can try a final run, do that to catch UX issues (like ghost being confusing, etc.). With launch so close, even a quick feedback cycle helps.

**7.3 Launch Day Procedure**

* Set up monitoring of logs and health endpoints. Have developers on standby to watch for any error logs or high latency.
* If any migration or data seeding is needed (maybe an initial knowledge base to load into concept mesh), do it now. E.g., if there's a base concept library, import it.
* Clear any dev/test data from the system (start with a fresh psiarc\_logs directory or empty Vault DB for real users).
* Announce a maintenance window if needed before flipping the switch, though since it’s first launch likely just go live.
* Have rollback plan: if something goes wrong, be ready to revert to a fallback (maybe disable some features like Ghost or Vault if they cause issues, so the core chat can still function). For instance, prepare a config flag to turn off the ghost system without redeploying code if needed (just in case).
* Once live, do an initial self-test (use the app in production for a couple of interactions to ensure all systems behave as in staging).

**8. Critical Fixes & Final Integration Recommendations**

Finally, based on the audit, here are the **highest priority fixes and integration tasks** to complete before launch:

* **Complete Ghost Integration:** As emphasized, integrate the Ghost AI into the chat loop **now**. This is both a flagship feature and a potential risk if left half-done. We must ensure ghost personas trigger appropriately and do not crash the system. This involves finishing touches in ghostPersonaEngine.ts and hooking it up in the message response logicfile-dgbgcxvgsy5hpzx1ekc3od.
* **Finalize Soliton Memory Hookups:** Double-check that the new SolitonMemory is fully tied into conversation handling. Any place that still uses old memory storage should be updated to use the soliton architecture. Ensure the **10% gap** in integration is closedfile-dgbgcxvgsy5hpzx1ekc3od. This likely means updating how messages are processed/stored and verifying the Rust bridge operates under load.
* **Memory Vault Functionality:** Implement the Memory Vault controls fully. This was a promised feature (with deep meaning for user trustfile-dgbgcxvgsy5hpzx1ekc3od) and should be working at launch. That means users can flag a conversation or concept to be vaulted, which triggers phase shift to reduce its saliencefile-dgbgcxvgsy5hpzx1ekc3od. Ensure the UI has this option (even if minimal) and back-end does the phase adjustment. Test it end-to-end as discussed.
* **Remove Dev Artifacts & Logs:** Scan the code for any dev-only outputs (console.log, debug prints). Remove or disable them, especially in tight loops or frequently called code (could slow down or clutter logs). However, keep enough logging around critical operations (maybe at info level) to help debug if something goes wrong in production.
* **Error Handling & User Feedback:** Go through the user flows and ensure any failure gives a user-friendly message. E.g., PDF fails -> show "Sorry, that PDF could not be processed." Chat AI fails -> show "TORI is thinking hard, please try again." rather than leaving the UI hanging. Implement try/catch around major steps and send error status to front-end with graceful messages.
* **Security Patches:** Quick fixes like rate limiting (even a simple in-memory counter), and strict CORS (if front-end hosted separately, ensure server only serves allowed origin). Also, ensure the latest packages are used to avoid known vulnerabilities (run npm audit fix if needed, but carefully so as not to break functionality last-minute).
* **Performance Tuning:** If any obvious slow part was found (e.g., if Ghost letter generation calls an external API or heavy model), consider caching results or simplifying for now. We want to avoid timeouts or lags at launch. It's better to have a slightly less fancy ghost message than to have the app freeze. Profile CPU if time permits and optimize the hottest spots.
* **Documentation & Team Handoff:** Prepare a short runbook for the team handling the launch (or yourself if you will be that person): highlight any known issues (hopefully none critical) and how to address them. E.g., "If Vault service crashes, restart it and note that chat will still function in interim." Also document how to gracefully shut down and back up data at end of day.

In conclusion, after executing the above audit steps and fixes, TORI should be **production-ready**. We will have a robust backend integrating SolitonMemory, ψArc logging, PDF ingestion, spectral alignment, and gRPC memory services, all secured and validated. The chat frontend will deliver persistent, personalized conversations enhanced by the Ghost AI companion. Every module will be tested, all .proto contracts in syncfile-mpebu5uvrt1irry4xlb7ct, and no black-box code hiding in .pyc files. By launch tomorrow, TORI will stand as a **fully integrated digital consciousness platform**, with infinite memory and empathetic intelligence – ready to make history.