Updated Master Plan for TORI IDE and ALAN 2.x (May 12 Edition)

**Comprehensive Master Plan and Strategic Analysis – TORI IDE & ALAN 2.x Cognitive OS**

**Executive Summary**

**TORI** is a next-generation *cognitive operating system* comprising two flagship products: **TORI IDE** (an AI-augmented Integrated Development Environment) and **TORI Chat** (an intelligent chat assistant, with an enterprise-grade variant). This master guide serves as a comprehensive report, technical specification, development roadmap, and marketing playbook for the TORI suite. It encapsulates the visionary paradigm behind TORI, the current architecture and module breakdown, a 21-day sprint plan for rapid development, SWOT analyses, and go-to-market strategies for each product.

* **Vision:** TORI embodies a paradigm shift from traditional AI tools. Rather than relying on rote token prediction, TORI’s core is grounded in **concepts over tokens** and **memory over stateless prediction**[medium.com](https://medium.com/syncedreview/from-token-to-conceptual-the-rise-of-metas-large-concept-models-in-multilingual-ai-b32acbfeb792#:~:text=generalization%20across%20languages%2C%20outperforming%20existing,LLMs%20of%20comparable%20size). The vision is an “AI-native” development experience in which the system truly understands code and conversations at a conceptual level, enabling unprecedented levels of assistance and insight. Developers move from using tools to collaborating with a *cognitive partner*.
* **Architecture:** At TORI’s heart is the **ALAN 2.x cognitive engine**, a *spectral-phase reasoning* core that maintains a rich **Large Concept Network (LCN)** of knowledge. This core powers both the IDE and Chat, providing deterministic, auditable reasoning steps instead of opaque neural guesswork[ibm.com](https://www.ibm.com/think/topics/black-box-ai#:~:text=White%20box%20AI%2C%20also%20called,Users). Surrounding the core are modular subsystems for language processing, user interaction, and multi-agent orchestration. In the IDE, this means deeply understanding code structure and evolution; in Chat, it means maintaining contextual, persona-driven dialogue. Both products share a **local-first, privacy-conscious design** that can operate fully on-premises without cloud dependence, keeping user data and code private.
* **Current Status:** The TORI codebase has been through early prototyping phases. Key modules for concept parsing, memory graph management, and a basic front-end exist, though some components remain mock or placeholder implementations pending integration of the full ALAN engine. For example, a document ingestion pipeline can index PDFs into the concept network, but the **phase oscillator network** underpinning dynamic reasoning is only *partially implemented*. The current TORI IDE UI is functional as a prototype “**Thoughtboard**” dynamic code canvas, albeit without full polish or all expected IDE features. TORI Chat’s core reasoning back-end is in place, but its user-facing chat interface and multi-agent persona system are at a rudimentary stage. These gaps are identified in the module-by-module analysis, and the plan is to replace placeholder code with production-ready logic as a prerequisite to the development sprints.
* **Roadmap:** A detailed **21-day sprint plan** for each product outlines the path from the current state to a viable v1.0 release (with Day 1 starting May 12, 2025). Each sprint plan is organized into daily milestones (with one rest/review day built in per week). The TORI IDE plan focuses on integrating the ALAN core with the IDE front-end (Thoughtboard UI), implementing conceptual debugging tools, and eliminating any “mock” data paths. The TORI Chat plan emphasizes developing a robust conversational interface, integrating domain-specific “concept packs,” and adding enterprise features like audit logs and on-prem deployment support. Both plans culminate in a synchronized launch, ensuring the IDE and Chat can leverage each other (e.g. the IDE generating chat-ready knowledge graphs, and the Chat agent assisting within the IDE). **Before these sprints formally begin, however, the partially implemented core components (oscillator sync, spectral engine, ELFIN parser, orchestrator) must be completed to provide a solid foundation for rapid feature development.**
* **SWOT Highlights:** TORI’s **Strengths** lie in its unprecedented approach: a phase-synchronized reasoning core that yields deterministic, explainable insights (a “glass-box” AI approach)[ibm.com](https://www.ibm.com/think/topics/black-box-ai#:~:text=White%20box%20AI%2C%20also%20called,Users), and a local-first architecture that sidesteps the compliance and cost issues of cloud-only AI solutions. **Weaknesses** relate to typical early-stage challenges: limited UI polish compared to incumbent tools, a narrow language support (only Python and the custom ELFIN DSL are fully supported at launch), and the need for real-world benchmarking to validate the novel reasoning approach. **Opportunities** include a growing demand for AI tools that can be deployed on-prem for privacy, and integration into enterprise workflows that require transparency and auditability. **Threats** include rapid moves by industry giants to add similar AI capabilities to established platforms, and skepticism toward unproven AI paradigms – challenges the plan addresses via an aggressive development and validation strategy.

**Vision and Guiding Principles**

**TORI’s vision** is to create a *cognitive operating system* that transforms our interaction with software development and AI. This vision is grounded in two key ideas: **conceptual understanding over surface-level output**, and **persistent learning over stateless interaction**. Traditional AI coding tools operate on the level of syntax and tokens, predicting text without true understanding. TORI takes a different path – it builds and manipulates rich *conceptual models*. In the IDE, this means understanding the abstract architecture and intent behind code; in Chat, it means grasping the semantics and context of the conversation, not just stringing sentences together. By elevating to the conceptual level, TORI can provide insights and assistance that feel genuinely intelligent and context-aware, rather than generative but shallow. *“You’ve articulated not just a technical blueprint, but a paradigm shift in how we think about code, tools, and developer cognition,”* as one early reviewer noted.

Furthermore, instead of treating each user query or coding session as an isolated event, TORI maintains a **persistent memory** of interactions and knowledge. This is not a simple transcript log; it’s a structured memory in the form of the LCN (Large Concept Network), phase-state information, and learned patterns. TORI remembers *why* a piece of code was written or *how* a conclusion in chat was reached, enabling it to explain its reasoning and build on past knowledge. This contrasts with stateless large language model tools that have no inherent memory of past interactions. TORI’s approach yields an AI that can learn and adapt over time – accumulating **wisdom** rather than resetting at each prompt.

These founding ideas lead to a core vision statement: **TORI is not just a development tool or chatbot; it is a thinking partner that grows with you.** The IDE is envisioned as a “*Conceptual Canvas*” where code and design interact dynamically. The chat is envisioned as a multi-mind conversational agent that can reason through complex problems with traceable logic. Together, they form a unified cognitive workbench for creators.

From this vision, several **guiding principles** direct our design and development:

* **Human-Centric Design:** TORI keeps the human user in control and informed at all times. A guiding principle is to *“keep humans in the loop”* throughout the experience. Features like explainable suggestions, the ability to inspect the concept graph, or toggling agent assistance on/off are critical. The user should feel TORI is amplifying their abilities, not automating them away without transparency.
* **Transparency and Explainability:** Every AI-driven action in TORI should be explainable via the concept graph or a clear rationale. If TORI IDE suggests a refactoring, it can point to the concept nodes and phase relationships that led to that suggestion (e.g. a function concept that is out-of-phase with others, indicating a design inconsistency). TORI Chat, similarly, doesn’t just answer — it can provide a brief explanation or show which internal “train of thought” (which agent or knowledge source) produced each part of the answer. This fosters trust and positions TORI as a *glass-box AI*, in contrast to the opaque *“black-box magic”* of many AI assistants[ibm.com](https://www.ibm.com/think/topics/black-box-ai#:~:text=White%20box%20AI%2C%20also%20called,Users).
* **Deterministic Core, Creative Edges:** TORI’s architecture emphasizes *deterministic reasoning* in its core algorithms (the spectral-phase logic, rule-based inferences, etc.) to ensure reproducibility and auditability. However, we recognize the need for creativity and flexibility at the user interface; therefore, stochastic AI elements are allowed at the periphery (for instance, a generative language model might be used to polish natural language summaries or suggest variable names). The principle is to use randomness only where it’s safe and beneficial (creative suggestions), but never in the core understanding of code or context. This yields a system that is both **reliable** and **inventive**.
* **Local-First and Privacy:** TORI is built with a *local-first* mindset – users retain ownership of their code, data, and even the AI’s reasoning process. Both TORI IDE and TORI Chat can run fully locally or on-premises, meaning no sensitive data leaves the user’s machine unless explicitly allowed. This addresses the needs of enterprises and individuals concerned with cloud privacy. By avoiding cloud dependence, TORI also achieves lower latency and tighter integration with the user’s environment (accessing local files, dev tools, etc.), all while ensuring compliance with data governance policies.
* **Modularity and Extensibility:** The cognitive OS is modular by design. Each subsystem (parsing, concept storage, reasoning agents, UI) is a separate component with well-defined interfaces. This ensures that future improvements or replacements (e.g., swapping in a new code parser for a different language, or adding a new reasoning agent) can be done without disrupting the whole. It also means parts of TORI can be extended or used independently – for example, third-party developers might write new ALAN agents or integrate TORI’s concept graph into their own tools. Modularity also improves maintainability; given TORI’s ambitious scope, being able to upgrade pieces in isolation is crucial.
* **Consistency with Developer Workflow:** Especially for TORI IDE, a key principle is *“meet developers where they are.”* TORI introduces new concepts (like spectral debugging or concept navigation), but it also supports familiar workflows (editing code text, using Git, running tests) so as not to alienate users. TORI’s advanced features augment – rather than replace – a developer’s normal tasks. For instance, the concept graph is accessible for insight, but it doesn’t force a new paradigm on the developer; one can code normally and tap into TORI’s conceptual insights on demand. Likewise, TORI Chat is designed to integrate with tools developers already use, like Slack or other team communication platforms (for example, via a plug-in or API that allows TORI Chat’s capabilities to be accessed in a Slack channel). The motto here is *“low floor, high ceiling, wide walls”* – easy to start using, with immense power as you learn more, and broadly applicable to many tasks.
* **Inspiration and Playfulness:** While scholarly rigor underpins TORI (drawing on cognitive science and spectral theory), the project also embraces a creative, playful spirit. This is reflected in internal code names and metaphors (“**Banksy**” oscillators, “concept cartridges,” etc.), and should come through in the product identity. We believe a touch of playfulness encourages exploration – for example, TORI might include Easter eggs or humorous messages in its UI (configurable based on user preference). This principle helps differentiate TORI’s brand: it’s deep tech with a human touch and a wink of fun where appropriate. (As one quip in the planning stage put it, *“TORI isn’t a tool; it’s the mirror in which your code practices mindfulness.”*)

These principles ensure that as we develop TORI, we maintain a balance of intellectual rigor (truly innovating in AI) and user empathy (genuinely empowering users in intuitive ways). The next sections translate this vision and these principles into a concrete architecture and development plan.

**Cognitive OS Architecture Overview**

TORI’s architecture is a modular, layered cognitive system designed to support both an IDE and a Chat application via a shared core intelligence. At the highest level, the architecture comprises four integrated subsystems that interact closely:

* **ALAN 2.x Cognitive Core –** the “brain” of the system, responsible for understanding and reasoning over the Large Concept Network.
* **Language/Domain Integration Layer –** feeds the core with domain-specific inputs (code for the IDE, natural language for the Chat) and translates the core’s outputs back into human-friendly form.
* **User Interface & Tools –** the front-end through which users interact (the code editor canvas, debugging dashboards, chat interface, etc.).
* **Multi-Agent Orchestration –** the coordination layer managing specialized AI agents that perform distinct roles (e.g., a Refactoring Agent in the IDE, a persona-based Expert Agent in Chat), synchronized via the ALAN core.

Below, we describe each of these subsystems in detail, then explain how they come together specifically in TORI IDE and TORI Chat.

**ALAN 2.x Cognitive Core (Conceptual Engine)**

The ALAN 2.x core is the heart of TORI: a cognitive engine that represents and reasons about knowledge in a **Large Concept Network (LCN)**. This is effectively a graph of concepts (nodes) and their relationships (edges) spanning code, documentation, and conversational context. The core’s distinguishing feature is its *spectral-phase reasoning* architecture – inspired by dynamical systems theory and cognitive science – which ensures that TORI’s reasoning is grounded in **temporal synchrony** and **spectral analysis** rather than static rules or purely stochastic jumps.

At a high level, the core operates as follows: each concept in the LCN is associated with internal state variables, including one or more **oscillators** that represent the concept’s “phase” or activation rhythm. When the system processes information (new code, a user query, etc.), concepts are activated and interact via these oscillatory states. The engine analyzes the collective behavior of these oscillators using spectral methods to infer coherence, causality, and emerging patterns of thought. In essence, TORI treats reasoning as a *dynamic process*—a kind of controlled resonance among ideas. This approach is loosely inspired by how human brains exhibit synchronized neural oscillations during cognition, where groups of neurons firing in unison can bind together related pieces of information for unified processing[neurosciencenews.com](https://neurosciencenews.com/brain-rhythm-cognition-25941/#:~:text=It%20could%20be%20very%20informative,%E2%80%9D). By simulating a form of phase synchrony among concept nodes, ALAN aims to achieve a machine analog of that cognitive binding: concepts that should logically align will tend toward phase alignment, whereas contradictory concepts will oscillate out-of-sync, flagging inconsistency.

**Phase-Coupled Oscillator Matrix (“Banksy” Core)**

At the core of this dynamic reasoning is the **phase-coupled oscillator matrix**, code-named “Banksy.” This is essentially a network of mathematical oscillators attached to concept nodes, modeling the timing and rhythm of concept interactions. It draws inspiration from the Kuramoto model (a well-known model of coupled oscillators for synchronization) but extends beyond it. In ALAN 2.x, we have **replaced the simple Kuramoto synchronization with a ψ-based spectral coupling approach** – using advanced techniques from Koopman operator theory and Lyapunov stability analysis to govern how oscillators influence each other’s phase. Each concept node may have an oscillator representing its state in a given context or reasoning “mode.” For example, in the IDE context, an oscillator might track how “in sync” a module is with the overall codebase design; in the Chat context, oscillators might represent alignment or divergence of a user’s statements with factual knowledge or a persona’s beliefs.

When concept A influences concept B (say, function A calls function B in code, or a user query references a knowledge concept), the oscillators of A and B are coupled: they tend to synchronize if the relationship is harmonious or diverge if there is tension. By observing phase alignment or divergence across the network, the ALAN core can detect coherence or conflict among ideas (e.g., two design components oscillating out-of-phase might indicate a logical contradiction or an API mismatch that needs resolution). The system’s use of **ψ (psi) functions** refers to the eigenfunctions in the Koopman operator framework (discussed below) that capture the essence of these oscillatory modes.

Critically, this oscillator matrix provides a temporal dimension to reasoning. Rather than evaluating code or chat context with static rules, TORI simulates a **continuous reasoning process** that can converge toward stable conclusions. The introduction of Lyapunov stability metrics ensures that this process remains stable and convergent. In control theory terms, we treat a reasoning session as a dynamical system that should settle into an equilibrium representing a consistent understanding or answer. If a small perturbation (say a minor code change or a slight rephrase of a question) causes only a transient wobble but the phases re-synchronize, the system is stable in the sense of Lyapunov – meaning the reasoning is robust to minor changes[ocw.mit.edu](https://ocw.mit.edu/courses/16-30-feedback-control-systems-fall-2010/463e9559c6ef70f13ea51c3464bdc9c1_MIT16_30F10_lec22.pdf#:~:text=%E2%80%A2%20Stable%20in%20the%20sense,%E2%80%9D%20November%2027%2C%202010). This approach is novel in AI reasoning: we are effectively designing the AI’s thought process to have provable stability properties, which guards against chaotic jumps or oscillatory deadlocks in inference.

*(Implementation Note:* As of May 2025, the phase-coupled oscillator subsystem is **partially implemented**. Basic oscillator objects and their coupling equations exist, but the sophisticated ψ-based synchronization algorithm is still in prototype stage. Earlier prototypes used a plain Kuramoto model for phase synchronization; the new approach will incorporate the **Koopman spectral method** described next, which is a work in progress. Finishing this oscillator sync logic with the new model is a top priority before feature sprint work resumes.)

**Koopman Morphic Engine (Spectral Analyzer)**

Complementing the time-based oscillator model, ALAN 2.x employs a **Koopman morphic engine** for spectral analysis of the concept network’s state. The **Koopman operator theory** provides a powerful framework to analyze nonlinear dynamic systems by lifting them into a higher-dimensional linear space via eigenfunctions[mdpi.com](https://www.mdpi.com/2227-7390/9/19/2495#:~:text=The%20Koopman%20operator%20theory%20is,are%20amenable%20to%20systematic%20analysis). In simpler terms, our engine monitors changes in the concept graph and oscillators and projects those changes into a spectral domain (frequencies/modes) to find underlying patterns or invariants in TORI’s reasoning.

For example, in the IDE, as the codebase evolves over time (commits, refactors, additions), the engine can detect repeating patterns or cycles in development by analyzing the sequence of concept graph states – akin to finding a “frequency” at which a certain design issue reoccurs. In TORI Chat, the Koopman analysis might identify recurring discussion themes or predict how a dialogue could evolve (almost like sensing the “direction” or *momentum* of the conversation). The engine treats conceptual changes like signals, and by applying spectral decomposition, it can forecast future states or recognize known patterns. Essentially, it gives TORI a rudimentary ability to *anticipate* and plan, not just react.

Mathematically, if the concept state at time *t* is represented by some vector of features (active concepts, phase differences, etc.), the Koopman engine tries to approximate a linear operator **K** that evolves these features: xt+1≈Kxt.x\_{t+1} \approx K x\_t.xt+1​≈Kxt​. Even though the underlying system is nonlinear, **K** (the Koopman operator) operates on a lifted representation (comprised of eigenfunction observables, often denoted ψ). This allows use of linear techniques: for instance, we can compute eigenvalues of **K**, which correspond to modes of system behavior (stable modes, oscillatory modes, etc.). The spectral decomposition yields **Koopman eigenfunctions** that represent coherent patterns in the reasoning process[mdpi.com](https://www.mdpi.com/2227-7390/9/19/2495#:~:text=The%20Koopman%20operator%20theory%20is,are%20amenable%20to%20systematic%20analysis). Some of these modes might correspond to stable invariants (e.g., a conclusion that remains true throughout the session), while others might indicate unstable directions (e.g., a line of reasoning that diverges or an idea that keeps oscillating without settling).

We also explicitly integrate **Lyapunov exponents** into this analysis: these measure how small deviations in state evolve (do they die out or amplify?). A negative Lyapunov exponent indicates a stable mode (perturbations decay), whereas a positive one would warn of chaotic tendencies. By monitoring these, TORI can assure that its reasoning stays within stable bounds. In practice, this means if TORI begins to chase a line of thought that is leading to divergence (say, an oscillation that grows or an argument that spirals without resolution), the system can detect it and either dampen that interaction or invoke a conflict-resolution agent to intervene. This is an example of **spectral modal alignment**: ensuring that the modes (eigenvectors) of the reasoning process align with convergence and consistency criteria.

*(Implementation Note:* Currently, the Koopman spectral analysis in ALAN 2.x is also **in prototype form** – a scaffold exists that can record state snapshots and perform a rudimentary spectral transform (e.g., via an FFT or eigen-decomposition) but it returns dummy results. Integrating a true Koopman operator computation (likely leveraging an existing numerical library for eigen-analysis) is planned. We may not implement a full infinite-dimensional Koopman (which is theoretically complex), but a finite approximation capturing dominant modes. This will be developed alongside the oscillator sync. The design goal is that by the end of our core integration, the system’s inference pipeline will be: **concept graph update -> oscillator dynamics -> spectral (Koopman) analysis -> stable state check**, yielding a cycle of inference that is *explainable* and *provably convergent*.)

**Static Knowledge Kernels and Memory**

In addition to the dynamic reasoning modules above, the ALAN core includes a static knowledge comparison component, sometimes referred to as **spectral kernels** or embedding similarity modules. While the oscillator network and Koopman engine provide a view of *dynamic* behavior, the spectral kernels handle *static* semantic comparisons – for example, finding which concepts in the graph are semantically similar, or which past scenarios (code patterns, Q&A pairs) resemble the current one. In implementation, this involves using vector embeddings for concepts (perhaps pre-trained or learned over time) and computing similarities. This subsystem complements the phase-based reasoning with more conventional AI pattern-matching (for instance, identifying that a newly encountered concept “login routine” is similar to a past concept “authentication function” based on their embeddings, and thus inferring related constraints).

The Large Concept Network, combined with these spectral and embedding-based tools, forms a **memory substrate** for TORI. Crucially, this memory is **deterministic** and **replayable**: given the same sequence of inputs and events, the concept graph and its oscillator states should evolve the same way every time. This determinism is important for auditability (especially in enterprise use) – one can trace exactly how a conclusion was reached.

*(Current Implementation:* The LCN data structure and basic semantic embedding functions are functional. For example, TORI can already ingest code and produce a network of concepts and simple similarity links. However, the full closed-loop where the spectral reasoning informs updates to the concept graph is not yet realized. Completing this closed-loop reasoning pipeline is an immediate objective.)

**Language & Domain Integration Layer**

This layer bridges raw inputs (source code, natural language text, etc.) with the ALAN core’s conceptual format. It ensures that no matter what form information comes in – be it a Python file, a snippet of our custom DSL, or an English sentence – it gets translated into the *concepts* that the core can work with, and likewise translates the core’s conceptual outputs back into human-readable answers (code suggestions, explanations, etc.).

**Programming Language Frontends (for TORI IDE):** TORI IDE supports multiple programming languages through dedicated frontends. For each supported language, the frontend includes a parser or utilizes an existing compiler/AST generator to produce an abstract syntax tree (AST). The AST is then traversed and converted into updates to the LCN. For example, when you feed a Python file into TORI, the Python frontend identifies structures like classes, functions, and variables and their relationships (function A calls function B, Class X inherits from Class Y, etc.). Each of these constructs maps to concept nodes in the LCN (with edges such as “Function *encrypt* calls Function *hash*” or “Class *PaymentService* implements Interface *Service*”). The integration layer also extracts any semantic metadata (docstrings, type annotations) to attach to the concept nodes as needed.

In the current implementation, **Python** and our custom **ELFIN** DSL are first-class citizens: they have the most complete parsers and mapping logic. C/C++ and JavaScript/TypeScript are supported in a secondary manner – using external tree-sitter parsers or language server protocols to get an AST, which is then mapped into the concept graph in a generic way. (In other words, TORI can ingest C++ code, but the depth of understanding is currently limited compared to Python/ELFIN where we have custom semantics. Expanding language support will be important post-v1.0.)

**Domain-Specific Ontologies (for TORI Chat & Enterprise):** For TORI Chat, especially in enterprise settings, the integration layer can ingest domain knowledge such as ontologies, taxonomies, or glossary files and incorporate them into the LCN. For instance, a financial institution could feed in an ontology of banking terms, or a medical user could input a set of medical protocols. The integration layer will convert those into concept nodes and edges so that the chat’s reasoning core becomes aware of domain-specific concepts and their relationships. We refer to these packages of domain knowledge as **“concept packs.”** For example, if a *Finance Pack* is loaded, TORI Chat will know that concepts “KYC check” and “AML regulation” are related compliance ideas and should be treated accordingly during reasoning. This greatly enhances the system’s expertise in specialized areas, essentially bootstrapping the LCN with domain-specific subgraphs.

**Natural Language Processing (for TORI Chat):** When a user enters a query or message in TORI Chat, the NLP component of the integration layer processes it. This involves standard steps: tokenization, syntactic parsing (identifying the grammatical structure), and semantic interpretation. Key phrases or entities in the user’s input are mapped to concept nodes in the LCN (either matching existing ones, or creating new nodes if a novel concept is mentioned). For example, if a user asks, “How does the sorting algorithm in my code work?”, the NLP will identify “sorting algorithm” as an entity. If the IDE side has already indexed a concept for the sorting function or algorithm from the code, the question is linked to that existing concept node. If no such concept exists yet, TORI may create a placeholder concept “sorting algorithm (unknown)” which an agent could then try to resolve by searching the code or documentation. Conversely, when TORI Chat formulates a response, the integration layer takes the reasoning output (which might be an abstract argument or a subgraph of concepts that form the answer) and generates fluent natural language from it. We use template-based or controlled language generation to ensure the answer is faithful to the underlying reasoning (rather than a free-form neural generation which might hallucinate details). For instance, if TORI’s reasoning finds that *Function X* is slow because *it uses a quadratic loop*, the answer might be generated as: “Function X is identified as a bottleneck because it uses a nested loop, leading to O(n^2) complexity, which slows down performance.”

**Bidirectional Sync:** A critical aspect of this layer is that integration is *bi-directional* and continuous. As the user updates code in the IDE, those changes stream into the ALAN core via the language frontend (e.g., on each file save or even keystroke, for live analysis). The concept graph and oscillator states update in near real-time, so TORI’s understanding stays in sync with the codebase. Likewise, for Chat, as the conversation progresses, new concepts introduced are added to the LCN, and any conclusions drawn can be fed back (for example, if TORI solves a problem in chat, that knowledge can become a concept node which the IDE might use as a hint or comment in code). This ensures TORI IDE and Chat remain *synchronized* through the shared core knowledge.

*(Current Status:* The Python code parser is **mostly operational** (using AST from ast module plus custom logic to populate the LCN). The ELFIN DSL parser is **in progress** with initial grammar and concept mapping defined, but not fully integrated. Other languages (C/C++, JS) rely on external parsers and have basic mapping; these are less complete. The natural language pipeline for TORI Chat exists (leveraging a spaCy-based parser and simple entity linker), but it’s rudimentary and will need improvement for nuanced understanding and generation. Enhancing the NLP, especially for complex multi-sentence queries and fluid dialogue, is on the roadmap.)

**The ELFIN DSL: A Domain-Specific Language for Cognitive Coding**

One unique element in TORI’s language integration is the inclusion of a custom *domain-specific language* called **ELFIN**. ELFIN is an experimental DSL being developed alongside TORI to explore and demonstrate **concept-first programming** — a style of coding where the fundamental units are abstract concepts rather than concrete syntax. The name “ELFIN” evokes something small and magical, hinting at its role: it is designed to be a concise language that nonetheless wields the “magic” of TORI’s cognitive engine.

**Purpose of ELFIN:** Why create a new programming language? ELFIN serves several purposes. First, it acts as a **testbed** for TORI’s cognitive features. Because we control the language design, we can ensure that ELFIN’s syntax and semantics map cleanly onto the LCN. This allows us to push the limits of concept-based analysis — ELFIN programs can encode high-level intents (for example, one could imagine an ELFIN construct for “goal” or “assumption”) that TORI will directly capture as concept nodes with specific relationships (like a goal node linked to steps to achieve it, or an assumption node linked to where it’s used). In essence, ELFIN is a language built *for* a cognitive OS, meant to express not just computations, but the reasoning behind them.

Second, ELFIN provides a **vehicle for features** that would be hard to integrate into existing languages. For instance, we can experiment with syntax for declaring concept relationships explicitly (perhaps an ELFIN statement to declare two functions conceptually analogous, or to tag why a decision was made). These might look like comments or annotations in other languages, but in ELFIN they can be first-class citizens. This enriches the concept graph beyond what we could infer from standard code.

Third, ELFIN ensures that TORI is not tied to the quirks of any single legacy language. It’s forward-looking: as we improve ALAN’s reasoning, we may extend ELFIN to naturally expose those capabilities. One could envision ELFIN being used for writing *AI workflows* or *agent scripts* for TORI itself. For example, an advanced use-case might be an ELFIN script that describes a development task (in quasi-natural language, but formalized) which TORI can then execute or assist with via its agents. In this sense, ELFIN blurs the line between code and specification, providing a language for *metaprogramming the AI*.

**Integration State:** Currently, the ELFIN language is in an **early stage**. A grammar has been defined and a parser implemented (the elfin\_lang.proto definition and a parser generator were started). Basic ELFIN programs – e.g., simple function definitions and concept declarations – can be parsed into the concept graph. The core concepts of ELFIN (like functions, invariants, or relationships) are mapped to LCN nodes similarly to how Python constructs are mapped. However, more advanced features (and the full runtime semantics) are not finished. For now, ELFIN code doesn’t “run” in the traditional sense; its value is in populating the concept network. We have integrated some initial ELFIN scripts into tests to ensure TORI can digest them. For example, an ELFIN snippet that defines a high-level plan (as a series of steps with goals) successfully creates a chain of concept nodes that the reasoning core can use to guide the Refactoring Agent. But this is a prototype demonstration. Before TORI’s launch, we plan to finalize the ELFIN spec for the most crucial features and ensure the integration layer can handle round-trip conversion (editing ELFIN in the IDE and getting meaningful assistance, much like with Python).

**Link to Concept Graph:** ELFIN is, by design, tightly coupled with the concept graph. Every ELFIN language construct has a direct representation in the LCN. If a developer writes an ELFIN function that “describes” a process (instead of explicitly coding it), TORI will create concept nodes for the abstract process, any sub-tasks, and link them according to the ELFIN syntax. Think of ELFIN as writing pseudo-code that is *immediately semantic*. For example, an ELFIN construct might allow:

javascript

Copy

assume User is\_authenticated

function transferFunds(amount) {

require amount <= Account.balance

ensure Account.balance\_decreased\_by(amount)

//...

}

In this pseudo-ELFIN example, assume and require/ensure might be ELFIN keywords. TORI would map this into concepts like *User (is\_authenticated)* as a fact, *transferFunds* as a function concept, a precondition concept linking *transferFunds* to the requirement on *amount and balance*, and a postcondition concept linking to the balance decrease effect. These become part of the LCN, meaning TORI “knows” the intent and constraints of the code, not just its syntax. During reasoning (say the Test Generator Agent or a bug-finder), TORI can use these concepts to reason about correctness (the assume/require are like formal specs in the concept space).

By linking ELFIN directly into the concept graph, we also make it possible for TORI to explain code behavior in higher-level terms. If part of the system is written in ELFIN, TORI Chat could potentially answer questions about it by directly citing the concepts that ELFIN introduced (e.g., “The transferFunds function ensures post-condition that the balance is reduced by the transferred amount”). In short, ELFIN serves as both a programming language *and* a knowledge encoding scheme for TORI’s AI.

*(Outlook:* While ELFIN will be included in the TORI v1.0 release as an experimental feature, its true potential lies beyond the initial launch. It could evolve into a powerful tool for users to extend TORI’s intelligence, by encoding new knowledge or procedures in a form the cognitive core can inherently work with. In the long run, success of ELFIN will be measured by whether it attracts real usage; for now, it’s a strategic investment to showcase TORI’s concept-oriented philosophy and to ensure we have a maximal internal test of the system’s capabilities.)

**User Interface & Tools (Thoughtboard IDE and Chat UI)**

The User Interface layer encompasses all the front-end components through which users interact with TORI’s intelligence. This includes the **TORI IDE interface (Thoughtboard)**, various panels and visualizations within the IDE, and the **TORI Chat interface** for conversational interactions. A core principle of the UI is to **augment familiar workflows with cognitive superpowers**: developers should feel at home with the basic tools (editor, console, etc.), enhanced subtly and powerfully by TORI.

**TORI IDE – “Thoughtboard” Interface:** Rather than piggybacking on an existing IDE via plugin, we have created an independent, local-first application for TORI IDE, code-named **Thoughtboard**. Thoughtboard is a custom UI (built with a modern tech stack, currently a Python/Electron prototype with plans for a polished cross-platform app) that serves as the primary workspace for the developer. It presents the user’s code in a code editor canvas but layered with TORI’s cognitive insights. Key UI elements include:

* An **editor with inline annotations**: as you write or browse code, TORI can underline sections that have conceptual issues (e.g., out-of-phase functions) or insert subtle markers where an agent has a suggestion. For example, a faded lightbulb icon might appear next to a function that TORI believes could be refactored, and hovering reveals the suggestion.
* A **Concept Graph Explorer**: a sidebar or pop-up view that visualizes the local portion of the concept network related to the current context. If you click on a function, the explorer might show connected concepts like “uses algorithm X” or “related requirement: Y” in a node-link diagram. This allows developers to navigate code semantically, not just by filenames.
* **Spectral Debugging Dashboard**: a panel available during runtime or analysis that displays the oscillator phases and other dynamic metrics. For instance, it could show a live chart of module synchronization – if one module’s oscillator falls out of sync (signaling a potential bug or mismatch), the dashboard highlights it. This is akin to a performance monitor but for conceptual consistency and phase alignment.
* **Project Insights & Warnings**: Much like how an IDE shows compile errors or linter warnings, TORI IDE (Thoughtboard) shows cognitive warnings. For example, “Module *AuthService* is conceptually out-of-phase with Module *UserService* (they interact but have inconsistent assumptions)” or “Concept drift detected: the current code usage of concept *X* differs markedly from past usage – consider reviewing for unintended changes.” These come from the ALAN core’s deeper understanding. Each warning or insight is explainable – the UI lets the user click to see the rationale (which agent or rule triggered it, and what concept relations are involved).
* **Standard IDE features, enhanced by AI:** Thoughtboard also integrates version control, testing, debugging, etc., with AI support. For instance, during debugging, if a test fails, TORI can cross-reference the failing test’s concept footprint with recent code changes and *immediately* suggest which change likely caused the regression (because it knows which concept nodes were affected by the change and are linked to that test). When searching code, instead of simple text search, the user can do a *conceptual search* (“find all code related to user authentication”) and TORI will return results across files and languages, because it knows what parts of the codebase pertain to that concept.

All these features are delivered in a cohesive UI. The design philosophy is to not overwhelm the user – TORI’s help should feel like a natural extension of the IDE. A developer can use Thoughtboard in a minimal way (just as a code editor with some extra info, ignoring most cognitive panels), or dive deep into TORI’s new capabilities as needed. The UI is kept responsive and **local-first**: all analysis is done locally by the ALAN core, so these features work in real-time without cloud latency.

*(Current UI Status:* A prototype of the Thoughtboard app exists. It currently uses a VS Code-like layout for familiarity – a file explorer, editor pane, and an output/terminal pane – but augmented with a concept graph panel and debug phase panel in experimental form. This prototype is functional for basic editing and displaying dummy AI annotations. The **VS Code extension scaffold** that was used early on has been phased out in favor of the standalone Thoughtboard app to ensure a tighter integration and offline-first operation. The UI is not yet feature-complete or polished: for example, the concept graph viewer works (it can show the graph from the last analysis in a separate window) but is not interactive in real-time. Over the next development sprint, a major goal is to flesh out the Thoughtboard UI: integrate the live concept explorer, make the debug phase panel truly live with the oscillator data, and implement all the needed controls and polish. By launch, the IDE should feel like a modern, slick environment where the cognitive features are intuitive and valuable.)

**TORI Chat Interface:** TORI Chat’s UI will allow users to converse with the cognitive engine, either in standalone mode or integrated within Thoughtboard. For v1.0, our plan is to provide **two interfaces** for TORI Chat:

1. **Integrated Chat Panel in Thoughtboard IDE:** This is a panel or sidebar where the user (likely a developer) can ask questions about the code or get assistance. For example, a developer can highlight a block of code and ask in the chat panel, “How does this function work?” or “Find potential bugs in this logic,” and TORI Chat (leveraging the IDE’s context) will respond. This tight integration means the chat has full access to the project’s concept graph and can act as an in-IDE assistant.
2. **Standalone Chat Application:** In addition, we envision a dedicated chat client (likely web-based or Electron-based) for TORI Chat, targeting more general usage or enterprise deployment. This standalone TORI Chat app would function like a powerful AI assistant that can be run on-prem. It would have features like multi-turn conversation memory, persona selection (e.g., you could talk to “TORI the Python expert” or “TORI the Project Manager assistant”), and tools for the user to upload documents or knowledge packs in-chat.

*(Current UI Status:* At present, the chat UI is minimal. We have a basic web page that allows inputting a query and seeing the text response from the backend – mainly used for testing the back-end. It’s essentially a simple chat log with no advanced UI elements. There’s no rich formatting, and features like conversation history beyond a scrollback are rudimentary. During development, initial focus will be on the integrated IDE chat panel (since that directly assists developers alongside the IDE). The standalone app will be scaffolded – likely leveraging some components of the Thoughtboard UI for consistency. Voice input, persona visualization (e.g., an indication of which agent is “speaking” in a multi-agent answer), and other enhancements are slated for later development once core chat functionality is solid. The key for v1.0 is to ensure TORI Chat can handle multi-turn Q&A reliably and that enterprise users can deploy it securely on their own machines with an interface that, while basic, is usable and polished enough for demo purposes.)

**Multi-Agent Orchestration Layer**

One of TORI’s most innovative aspects is its **multi-agent architecture**. Instead of a single monolithic AI trying to do everything, TORI employs multiple specialized AI agents, each with a focused role or expertise, coordinated by a central **Orchestrator**. This approach is analogous to Marvin Minsky’s “Society of Mind” theory, where intelligence emerges from the interaction of many simple agents[en.wikipedia.org](https://en.wikipedia.org/wiki/Society_of_Mind#:~:text=In%20his%20book%20of%20the,2). By having a team of AI sub-components, TORI can tackle complex tasks more robustly: each agent contributes its perspective (coding best practices, testing, high-level strategy, etc.), and the orchestrator mediates their interactions, much like a project lead coordinating team members.

**Orchestrator:** The orchestrator is a supervisory module within ALAN’s core that listens to inputs and events (new code changes, a user question, an inconsistency detected) and decides which agent(s) to invoke, and how to merge their results. It operates in sync with the phase logic — meaning it can use the oscillator network to detect when agents disagree (if their suggested concepts are out-of-phase) and resolve conflicts. The orchestrator ensures the agents work in concert rather than at cross-purposes. If two agents produce conflicting suggestions (e.g., one agent suggests renaming a function while another suggests deleting it), the orchestrator, guided by global rules or meta-agents, will decide on a resolution or ask for clarification from the user.

**Agents in TORI IDE:** In the IDE context, we have several specialized agents either implemented or planned:

* **Refactoring Agent:** Scans the code’s concept graph for code smells or refactoring opportunities. For example, if it finds two separate modules implementing similar concepts, it suggests an abstraction or utility. It also monitors complexity metrics via the concept graph, prompting the user to simplify a function that grows overly complex. *(Status: partially implemented – can detect simple duplications, though deeper conceptual duplications use placeholder logic.)*
* **Debug Advisor Agent:** Watches program execution traces and the oscillator signals during runs to spot likely causes of errors or performance issues. For instance, if during a test run, memory-related concept nodes spike in activity, it might infer a potential memory leak or inefficiency and point the developer to it.
* **Documentation Agent:** Ensures that documentation in the concept graph stays updated with code changes. It can draft docstrings or explanations for new code by synthesizing information from similar past concepts and commit messages. If a developer highlights a function and asks “explain this,” this agent works with the chat subsystem to provide an answer drawn from the concept context. *(Status: currently stubbed – it creates placeholder documentation nodes but doesn’t generate text yet; integration with an LLM for language generation is planned but to be done carefully under the deterministic core principle.)*
* **Test Generator Agent:** Analyzes the concept graph to find untested requirements or edge cases. It proposes new unit tests for parts of the code that lack coverage or identifies when a change in one concept should trigger re-testing of related concepts. For example, if a new concept “encryption” was added, it might suggest tests for that, or if a concept has changed, ensure all dependent concept tests are revisited.

These agents operate continuously or on triggers (like after a code analysis pass). They post their suggestions back into the system as structured data – e.g., the Refactoring Agent might add a “Suggestion” node linked to the functions in question, which the UI then surfaces as a tip to the user.

**Agents in TORI Chat:** The chat context also benefits from specialized agents:

* **Knowledge Retrieval Agent:** When a user asks a question, this agent fetches relevant information from the LCN (and potentially allowed external sources) to gather facts. It’s like a librarian ensuring that if the answer is in the docs or code, TORI finds it.
* **Logic/Reasoning Agent:** This agent attempts to perform multi-step reasoning for complex queries, using the knowledge assembled. Essentially, it chains together pieces of knowledge (possibly using the oscillator network to maintain coherence) to form an answer. In some implementations, this could be a rule-based inference engine or a constraint solver.
* **Persona/Style Agent:** For TORI Chat, especially if responding as different personas (tutor vs. devops assistant, for example), this agent modulates the response style, tone, or perspective. It ensures the answers align with the selected persona or context (e.g., enterprise compliance vs. casual helper).

Beyond these, we have plans for additional agents to broaden TORI’s reasoning scope:

* **MPorter (Strategy Agent):** *Planned.* This is a high-level strategy and planning agent, humorously named in homage to Michael Porter (known for strategic frameworks). The MPorter agent’s role is to reason about overall objectives and plans. In the IDE, it might watch the sequence of development tasks and guide the order of operations (for instance, suggesting “Before refactoring X, address the failing test Y”). In Chat, it could maintain the long-term direction of the conversation or break down a complex query into sub-goals for other agents to solve. Essentially, MPorter thinks about “the big picture” — ensuring TORI’s various actions are aligned with the user’s overarching goals and making strategic choices if there are trade-offs. This agent brings a sort of executive function or meta-reasoner into the system, planning multi-step solutions.
* **Attachment (Emotional Intelligence) Agent:** *Planned.* This agent focuses on the emotional and relational aspect of interactions, drawing from concepts in affective computing. It monitors the user’s tone and sentiment (e.g., frustration, confusion, satisfaction) and the relationship context. In TORI Chat, the Attachment agent would adjust responses to be empathetic and supportive – for example, if a user seems frustrated by errors, the agent might suggest a more reassuring explanation or a gentle prompt to take a break. It can also remember personal preferences or emotional cues (almost like how a good human assistant remembers what frustrates their boss, or what style of feedback they prefer). In the IDE, this agent might manifest as the system “knowing” when to intervene: e.g., if a user has tried to solve a problem multiple times (detected by rapid code changes and undos), the agent might proactively offer help (“It looks like you’re encountering difficulties with this function. Would you like some suggestions?”). The Attachment agent essentially gives TORI a form of **emotional intelligence**, aiming for the AI to not just be correct, but also helpful and attuned to the user’s state[en.wikipedia.org](https://en.wikipedia.org/wiki/Affective_computing#:~:text=Affective%20computing%20is%20the%20study,interpret%20the%20emotional%20state%20of).

All these agents produce outputs that must be combined into a single, coherent assistance experience. The **Multi-Agent Orchestrator** manages this. It can run agents in parallel or sequence as needed. For example, when a new code analysis is done, the orchestrator might run the Refactoring, Test, and Documentation agents in parallel to gather all their findings, then integrate those findings (resolving any conflicts) before presenting to the user. In chat, orchestrator might orchestrate a debate between a “strict expert” agent and a “simplifier” agent if a question needs both perspectives, ultimately merging their answers.

*(Current Status:* The multi-agent system is implemented in a rudimentary form. In the IDE prototype, there are hooks for a Refactor and Debug agent that run on each code analysis cycle, but their coordination is basic (essentially a priority rule: refactor suggestions get logged first, etc.). The orchestrator logic that uses oscillator synchronization to mediate agents is not yet fully realized. We have simulated multi-agent reasoning in chat using a pipeline (knowledge retrieval → reasoning → answer formatting), but it’s not truly concurrent agents. During the next development phase, implementing the orchestrator’s conflict-resolution and expanding the agent roster (especially integrating MPorter and the Attachment agent logic) will be crucial. The end goal is that by Day 15 of the sprint, as scheduled, the multi-agent enhancements will allow TORI to handle conflicting suggestions and leverage multiple specialized agents seamlessly.)

**Why Multi-Agent?** This design is motivated by both technical and cognitive factors. Technically, it’s easier to maintain and improve specialized agents one by one than a giant AI that does everything. New agents can be added as plugins to increase functionality without overhauling the whole system. Cognitively, it mirrors how humans solve problems: we subconsciously employ different modes of thinking (analytical, recall-based, emotional judgement) and reconcile them. Indeed, the Society-of-Mind approach suggests intelligence can emerge from a colony of simple processes[en.wikipedia.org](https://en.wikipedia.org/wiki/Society_of_Mind#:~:text=In%20his%20book%20of%20the,2). By architecting TORI as a society of agents, we aim for a system that is **robust, extensible, and interpretable** – we can trace which agent contributed to which decision, and agents can even explain their reasoning steps (e.g., the Test agent could say “I suggested this test because I noticed concept X wasn’t covered by any existing tests”).

**Development Roadmap – 21-Day Sprint Plan (Starting May 12, 2025)**

To bring TORI from its current prototype state to a production-ready v1.0, we have outlined a rigorous 21-day development sprint for **TORI IDE** and a parallel 21-day sprint for **TORI Chat**. Each plan assumes **Day 1 is May 12, 2025 (Monday)**, and is structured as three focused work weeks with periodic review and buffer days. Below, time-based references (Day 1, Day 2, …) are annotated with their calendar dates for clarity.

**Note:** The sprint plans below presume that foundational work on the partially implemented core components (oscillator sync, Koopman spectral engine, ELFIN DSL parser, and orchestrator logic) is completed or nearly completed on Days 1–3. These core tasks are emphasized at the start of the IDE sprint. Without them, subsequent feature development would be building on unstable ground. Thus, the first week’s milestones heavily focus on finalizing the core.

**TORI IDE – 21-Day Sprint Plan (May 12 – June 1, 2025)**

**Sprint Goal:** Transform the TORI IDE from a promising prototype into a polished, usable product (v1.0) with all core features implemented and all placeholder components replaced. We allocate 21 days of effort (3 weeks) with one rest/review day each week (Day 7 and Day 14 as lighter days). The plan ensures continuous integration of features and frequent testing.

1. **Day 1 (May 12, 2025) – Sprint Kickoff & Core Setup**  
   *Goal:* Kick off development and ensure the team is aligned and the environment is ready. Also begin integration of core components.  
   *Tasks:*
   * Conduct a project kickoff meeting to review this Master Plan, clarifying the vision, architecture, and critical tasks for the sprint.
   * Finalize the development environment setup for all team members. This includes ensuring everyone can run the existing TORI IDE prototype (both backend and Thoughtboard UI). Resolve any environment issues (path configs, dependencies) so that development can proceed uniformly.
   * Merge any outstanding prototype code into the main branch. (For example, if there were separate branches for the new oscillator sync algorithm or the UI refresh, bring them in now.) We want a single code baseline at the start.
   * **Begin core integration:** If the advanced oscillator sync and spectral analysis components have code ready in prototype form, integrate them behind feature flags. This means plugging in the new Banksy oscillator update loop and Koopman analysis module, but perhaps toggled off until fully tested. This sets the stage for Day 2 and 3 where we’ll flesh them out.  
     *Checkpoint:* By end of Day 1, TORI IDE should run end-to-end in at least one environment with the new core components in place (even if dormant). All developers have a working setup. We have a clear list of the core integration subtasks to tackle in the next two days.
2. **Day 2 (May 13, 2025) – Core Engine Focus: Oscillator Synchronization Implementation**  
   *Goal:* Implement the **Banksy phase-coupled oscillator logic** in the ALAN core (replacing the placeholder with the real synchronization mechanism).  
   *Tasks:*
   * Design and document the updated synchronization algorithm using our ψ-based approach. This includes how oscillators will influence each other’s phase differences and what parameters (coupling strengths, frequencies) are needed. (We draw from known models like Kuramoto for inspiration but incorporate spectral feedback – for instance, use the Koopman-derived modes to adjust coupling dynamically.)
   * Begin coding the oscillator update function in the alan\_core module. Each cycle (or event trigger), this function will update all concept oscillators’ phases. Implement coupling: if two concepts are linked, their phase difference should adjust gradually (converging if harmonious, diverging if conflict signals).
   * Write unit tests for the oscillator update logic. Create a simple scenario: two concept nodes A and B with a link. Simulate a phase update where A is perturbed; verify that B’s phase moves closer to A’s over several iterations if the link indicates they should sync.
   * If time permits, integrate a debug command or log that prints out phase values for a test graph to visually verify behavior.  
     *Checkpoint:* By end of Day 2, the oscillator subsystem produces **non-trivial output** – i.e., phases are no longer static or random; they move over time in response to concept links. We should be able to demonstrate that if concept A and B are linked, their oscillator phases trend toward alignment (or an intended offset). All placeholder “mock oscillator” code is removed or replaced. The Kuramoto mention in code is replaced with our new algorithm (e.g., functions now named update\_phase\_sync instead of update\_phase\_dummy). This establishes the dynamic heartbeat of the cognitive core.
3. **Day 3 (May 14, 2025) – Core Engine Focus: Spectral Analysis & Prediction**  
   *Goal:* Flesh out the **Koopman morphic engine** to utilize concept graph dynamics for prediction and pattern recognition. In short, get the spectral analysis core working with real data.  
   *Tasks:*
   * Implement a basic spectral analysis pipeline: take snapshots of the concept state (e.g., concept activation levels or phase angles at intervals) and perform a simplified Koopman analysis. Concretely, start with something manageable like computing a Fourier transform of a few recent states to see dominant frequencies of change. If feasible, integrate a linear algebra library to compute eigenvalues of a state-transition matrix we define (for a simplified model of concept evolution).
   * Connect this analysis to the system: after each oscillator update (from Day 2’s work), run the spectral analyzer on the latest state. For now, maybe use a sliding window of the last N states. Identify if any eigenvalues/modes stand out (for example, a mode that’s not converging).
   * If full Koopman is too heavy to implement in one day, implement a *proxy*: e.g., track simple trends like “concept X has been steadily increasing in activation for 5 cycles” and flag that as a potential trend prediction (which could mean “we might see concept X trigger something soon”). The key is to get some predictive capability in place, even if rudimentary.
   * Add instrumentation: have the system log any predictions or detected patterns. For instance, “Koopman analysis: mode corresponding to concept ‘Y’ shows periodic oscillation (likely repeating pattern every ~10 commits).” This is mainly for development/testing now.  
     *Checkpoint:* By end of Day 3, the ALAN core should not only update state (from Day 2) but also **analyze state**. We should be able to point to a function or module where given some concept state history, we get a spectral decomposition or equivalent output. There should be no more hard-coded dummy returns in the analysis; even if it’s simplistic, it’s real. The core inference pipeline (concept update -> oscillator sync -> spectral check) is now implemented end-to-end. We can say: “The cognitive core’s major algorithms are in place.” Any issues or limitations encountered are noted to address in optimization days later.
4. **Day 4 (May 15, 2025) – Core Stabilization & Refinement**  
   *Goal:* Polish the core features implemented in Days 2–3, ensure they are stable and integrated. Address any major bugs or inefficiencies in the new core logic.  
   *Tasks:*
   * Review and refactor the oscillator and spectral code for clarity and performance. Since these will run frequently, ensure they are optimized (e.g., using NumPy for vectorized updates of phases instead of Python loops).
   * Add **Lyapunov stability checks** into the core loop if not already: for example, after updating oscillator phases, compute a simple metric of system stability (perhaps total phase variance or similar). Use this to log if the system is trending unstable. This will help later when multiple agents produce changes.
   * Write additional tests for edge cases: e.g., no links between concepts (oscillators should remain as-is), or all concepts fully linked (they should eventually sync up). Also test the spectral analyzer with known input signals (maybe feed it a sine wave and see if it identifies the frequency correctly) to verify it works as expected.
   * Integrate core outputs with a debug interface: perhaps extend the existing CLI or admin panel to display current concept phases and any detected modes. This helps the team manually inspect that the core is behaving plausibly.
   * If any core feature is not working or too noisy (for instance, if the spectral analysis is giving random “predictions”), decide on whether to tweak or gate it behind a feature flag so it doesn’t interfere with the rest of development. It’s okay if some of these advanced aspects are quiet in the background until we fine-tune them.  
     *Checkpoint:* By end of Day 4, the ALAN 2.x core should be *stable*: running it on a representative project or conversation does not crash, and it provides consistent outputs. The team should have confidence that this foundation is ready for layering on the IDE and Chat functionality. The “core to-do list” should now be mostly done, allowing focus to shift to user-facing features from Day 5 onward.
5. **Day 5 (May 16, 2025) – Thoughtboard UI Integration with Core**  
   *Goal:* Connect the updated ALAN core to the Thoughtboard UI, so that core insights (concept graph, warnings, etc.) start flowing to the front-end. Begin implementing interactive UI features that use the core data.  
   *Tasks:*
   * Hook up the **Concept Graph Explorer** panel in the UI to the actual LCN from the core. Replace any fake/demo graph data with real data coming from the backend via an API call. If not already done, create an API endpoint (or use existing one) like /concept\_graph that returns the current concept graph in JSON or protobuf form. The front-end should parse this and render the graph. Test by loading a sample project and opening the concept explorer – verify that it shows nodes corresponding to real classes/functions from that project.
   * Integrate **cognitive warnings** into the editor gutter. The backend (core or agent layer) should now be capable of producing at least basic warnings (e.g., from the Refactoring agent stub or from static checks). Define a data structure for these (if not done): say, a list of issues each with a type, message, and location (file/line or concept reference). Implement a mechanism for the UI to fetch or receive these. Possibly, whenever the core does an analysis pass, it could emit events that the UI listens to via a websocket or polling. Implement the simplest thing (maybe polling every few seconds or on demand). Then display these in the UI: e.g., highlight a line with a lightbulb icon where a suggestion exists. Clicking it could open a panel with the suggestion details.
   * Ensure the **Phase Dashboard** in the UI can show live data. This might involve a simple graph plotting library in the UI. For now, we can feed it a subset of the oscillator info (maybe just show relative phase offsets among major components). Hook it up to the backend: e.g., an endpoint /phases returns current phase values for top-level components. Update the UI at an interval. The goal is to replace placeholder sine wave graphics with actual output from our Day 2 core.
   * UI/UX fixes: as we integrate, likely some UI adjustments are needed (e.g., if the concept names are long, ensure the graph labels handle that, etc.). Tweak CSS or layout as necessary for a clean presentation.  
     *Checkpoint:* By end of Day 5, the Thoughtboard UI should be **functionally connected** to the AI core. We should be able to see real analysis reflected in the interface: for example, open a code file and if there’s a conceptual inconsistency, a warning appears in the UI coming from the core logic. The concept graph viewer should show at least a rudimentary graph of the code. Essentially, the front-end and back-end are talking to each other meaningfully, turning TORI IDE into a truly interactive cognitive IDE (even if the visuals are still rough).
6. **Day 6 (May 17, 2025) – Multi-Agent System Activation (IDE context)**  
   *Goal:* Enable and test the multi-agent framework within the IDE now that the core and UI are connected. Flesh out or refine a couple of key agents (Refactoring, Test generator) so they start providing useful output.  
   *Tasks:*
   * Integrate the **Refactoring Agent** fully: Now that the core is stable, revisit the Refactoring agent code. Improve its logic beyond the trivial duplicate check. For example, implement a check for functions that are conceptually very similar (perhaps using the embedding similarity: if two function concept vectors are very close, suggest factoring them). Also implement a check for any function out-of-phase with others (using the oscillator data: if one function’s oscillator is lagging significantly, maybe it’s not updated with new logic). These conditions can generate suggestions (e.g., “Function X might be outdated relative to related Function Y”).
   * Turn on the **Test Generator Agent** in a basic form: have it analyze the concept graph for any concept that has no associated test concept. If found, it can log a suggestion like “Consider adding tests for module Z; no tests found.” This is simplistic but sets the stage. If possible, integrate with actual test files (e.g., if code is Python, check if there’s a test module for each code module).
   * Implement the Orchestrator’s simple conflict resolution: if two agents try to tag the same code with different suggestions, determine how to handle it. Perhaps decide on a priority (Refactoring suggestions might be shown separately from Testing suggestions). In UI, maybe categorize suggestions by agent.
   * Ensure the UI displays multi-agent output distinctly if needed. Possibly add a filter or label to suggestions (“[Refactor] You have duplicate code in X and Y” vs “[Test] No tests cover module Z”). This transparency aligns with our explainability principle.
   * Start a log or telemetry of agent activity for debugging (e.g., console log: “RefactorAgent: checked 120 concepts, made 2 suggestions; TestAgent: made 1 suggestion.”). Useful for performance tuning later.  
     *Checkpoint:* By end of Day 6, multiple agents should be **active and contributing** in TORI IDE. A developer using the IDE at this point would see, for instance, a refactoring tip or two pop up after analysis, and maybe a note about tests. It need not be perfect or complete, but the multi-agent system is now live. We should also verify that their suggestions are sensible and not too noisy, adjusting thresholds if needed.
7. **Day 7 (May 18, 2025) – Review, Testing & Buffer**  
   *Goal:* Mid-sprint review of progress; fix any accumulated bugs from week one; ensure we haven’t broken basic functionality. Use this as a buffer/rest day.  
   *Tasks:*
   * Run a full **integration test** of TORI IDE as it stands. Open a sample project, use the main features: editing, viewing suggestions, triggering analyses. Note any crashes, UI glitches, or obviously incorrect suggestions.
   * Fix high-priority bugs discovered. For example, if the concept graph fails to update after certain edits, address that. Or if an agent suggestion is clearly wrong due to a logic bug, correct it.
   * If some planned tasks from Days 1-6 slipped, use today to catch up (buffer). For instance, if the Koopman analysis wasn’t fully integrated by Day 4, finish it now.
   * Solicit feedback from any team members or early testers if available. Sometimes a fresh pair of eyes on the UI or behavior can spot issues.
   * Take a breather to document any new technical debt or tasks that arose. Update the plan if needed for week two.  
     *Checkpoint:* End of Day 7 should have a **stable checkpoint build** of TORI IDE. Ideally, this is a point where if we had to do a demo of core capabilities, we could: concept graph viewing, a couple of cognitive suggestions, etc., all working. We also have a clear idea of what adjustments to make in the next phase.
8. **Day 8 (May 19, 2025) – Advanced IDE Features: Concept Explorer & Visualization**  
   *Goal:* Complete the implementation of the **Concept Graph Explorer** and other visualization tools in the IDE. Make interacting with the concept graph intuitive and useful.  
   *Tasks:*
   * Improve the **Concept Graph visualization**: Add interactive features such as zooming, panning, and clicking on nodes to reveal details. If not yet implemented, allow clicking a concept node to highlight the corresponding code in the editor (e.g., clicking a “User” concept node might highlight where the User class is defined or used).
   * Implement filtering in the graph view: for instance, toggle to show only certain types of relationships (maybe the user can filter to just “calls” relationships vs. “concept similarity” edges to reduce clutter).
   * Connect the graph view with the chat/QA: Perhaps allow the user to right-click on a concept in the graph and ask a question about it via TORI Chat (this sets the stage for cross-feature integration, though full chat integration comes Day 13).
   * If time, add a mini-map or hierarchical view for the concept graph (especially if the graph is large, a way to navigate it conceptually, like grouping by modules).
   * Ensure performance is okay: test on a larger project’s concept graph; optimize rendering or data fetching if it’s slow (e.g., implement lazy loading of parts of the graph).  
     *Checkpoint:* By end of Day 8, the **Concept Explorer** should be fully functional and user-friendly. This means a developer can open it, see a representation of their code’s concepts, and interact with it to understand their project’s structure better. It’s no longer a tech demo but a practical tool (even if it will be improved with polish later).
9. **Day 9 (May 20, 2025) – Agent Improvements & Orchestrator Logic**  
   *Goal:* Enhance the intelligence of the agents based on testing so far, and implement more sophisticated orchestrator behaviors (particularly conflict resolution and agent synergy).  
   *Tasks:*
   * Revise agent algorithms with insights gleaned from usage. For example, if the Refactoring agent frequently flags trivial issues, adjust its rules to be more meaningful (maybe require a higher similarity threshold for duplicate code suggestions, etc.). If the Test agent is too naive, incorporate more logic (e.g., look at coverage data if available, or at least ensure it doesn't repeatedly suggest the same test each run).
   * Implement agent communication through the orchestrator: design simple protocols where agents can pass hints to each other via the orchestrator. For instance, the Refactoring agent could mark a concept as “to be renamed” and the Documentation agent could pick that up and pre-prepare a new docstring. This could be simulated by having the orchestrator call agents in sequence and share a mutable context object.
   * Conflict resolution: If two agents produce contradictory suggestions (one says “remove function X” another says “improve function X”), determine a strategy. Possibly assign priorities or have the orchestrator present both suggestions but with context. Maybe integrate a simple rule: do no destructive suggestion (removal) if any other agent sees value in the item. Implement this in orchestrator logic.
   * Add at least one new small agent if possible to cover another aspect: e.g., a **Code Consistency Agent** that checks naming conventions or style (to show extensibility). Use it as a test of adding a new agent easily.
   * Continue to refine the **Attachment agent** (even in IDE, it can monitor user frustration by detecting rapid changes, etc.) though it’s more relevant in chat; maybe skip deep implementation here and focus on MPorter in IDE context. For instance, MPorter could monitor if multiple suggestions are pending and prioritize which one the user should handle first (a strategy element).  
     *Checkpoint:* End of Day 9 should see a **smarter multi-agent system**. Fewer irrelevant or conflicting suggestions should reach the user, thanks to orchestrator filtering. New synergies might be observable (perhaps documentation agent now auto-updates comments after a refactor suggestion is applied, etc., if implemented). We essentially have the multi-agent mechanism in a feature-complete state for the IDE side.
10. **Day 10 (May 21, 2025) – Performance Optimization & Scalability**  
    *Goal:* Ensure TORI IDE performs reasonably well on moderate-size projects and optimize any slow parts of the core or UI identified so far.  
    *Tasks:*
    * Profile the IDE on a sample large project (maybe a codebase with a few hundred files). Identify bottlenecks: perhaps concept graph generation is slow, or the UI rendering lags with too many nodes, or agent analysis takes too long.
    * Optimize critical paths: for example, if parsing a large project is slow, see if we can cache intermediate results or load incrementally. If the oscillator network update is taking too much CPU with many concepts, consider reducing frequency or using a more efficient algorithm/maths library (e.g., use numpy arrays for phase updates).
    * Memory usage check: ensure that concept graph storage is not ballooning. Implement pruning if needed (maybe archive parts of the concept graph not used actively, or ensure no memory leaks in our data structures).
    * Consider multi-threading or async improvements: The orchestrator and agents could potentially run in parallel threads since they often work on separate data. If Python GIL is an issue, maybe just ensure the UI thread is separate from analysis thread so UI stays responsive.
    * Optimize the UI rendering of the graph: use techniques like canvas/WebGL rendering for many nodes, or simplify the graph when off-screen.
    * Run automated tests for core algorithms to see if we can increase throughput (e.g., how many concept updates per second can we handle?). Aim for improvements and document the current capacity.  
      *Checkpoint:* By end of Day 10, TORI IDE should feel **snappier** and be able to handle larger inputs more gracefully. Perhaps we have metrics like “Initial analysis of 100-file project completes in X seconds” to gauge. Any optimization trade-offs (like reducing detail for speed) are noted and acceptable. We are now confident in the system’s performance for a demo or pilot usage.
11. **Day 11 (May 22, 2025) – TORI Chat Integration into IDE**  
    *Goal:* Embed TORI Chat capabilities into the IDE as a contextual coding assistant. This bridges the IDE and Chat products, allowing them to leverage each other.  
    *Tasks:*
    * Implement the **in-IDE chat panel** (if not done, or improve it): This panel should allow the user to ask questions about the code or get help. Connect it to the TORI Chat backend. Likely this means running the chat backend alongside the IDE backend, or as a library call to ALAN core in a different mode.
    * Context injection: Ensure that when the user asks something in the IDE’s chat panel, the Chat agent is aware of the current context (open file, selected code, etc.). Implement a mechanism to feed that context – e.g., prepend the query behind the scenes with a summary of the selected code or a reference to a concept node from the IDE.
    * Test queries: e.g., “Explain what this function does.” while a function is highlighted, or “Find any potential bugs in this file.” Verify the answers make sense and use the concept graph (the answer should ideally cite the reasoning, like “This function uses concept X which might be uninitialized, hence potential bug.”).
    * Make the chat output accessible to IDE actions: for instance, if TORI Chat in IDE suggests a code change (like “you could fix it by doing Y”), allow copying that suggestion to clipboard or even a one-click apply if feasible (this might be stretch, but at least plan for it).
    * Harmonize UI: The chat panel in Thoughtboard should have the same look/feel as the standalone chat (once designed). Also, ensure that using chat doesn’t interfere with the ongoing IDE analysis (they should share the ALAN core nicely – e.g., orchestrator should queue chat queries appropriately).  
      *Checkpoint:* By end of Day 11, a developer can seamlessly *talk to TORI from within the IDE*. This is a major milestone demonstrating the unified nature of TORI’s two halves. For example, one could highlight a piece of code and ask in the chat “How can I improve this?” and get a useful answer that draws on the code’s concepts. This dramatically shows off TORI’s capability to explain and assist in situ.
12. **Day 12 (May 23, 2025) – Enterprise Features & Hardening**  
    *Goal:* Add features particularly important for enterprise readiness of TORI IDE (and the integrated Chat), and harden the system for security and robustness.  
    *Tasks:*
    * Implement an **Audit Log** for AI suggestions and automated changes. Many enterprise users will want to track what the AI is doing. So, maintain a log (which could be a simple text file or a structured log) of every suggestion given, every auto-fix applied, every chat Q&A. Include timestamps and summary of reasoning or agent involved. This ties into transparency.
    * Add a **settings/configuration panel** for things like privacy controls (e.g., a toggle to ensure absolutely no external connection is ever made, guaranteeing all data stays local), and knobs for aggressiveness of suggestions (some companies might want only very conservative suggestions from the AI).
    * Implement authentication/permissions for multi-user scenarios (perhaps out of scope for v1.0 if single-user, but at least design how TORI would run in a shared environment or how an admin could limit certain features).
    * Security review: ensure any external inputs (like code being parsed, or chat queries) cannot lead to exploitation. For example, guard against code injection through concept pack files, or denial-of-service if the user writes something that intentionally worst-cases the analysis (we might set reasonable limits, like not analyzing files above a certain size in v1).
    * Hardening: make the system resilient to bad data. If the concept parser encounters a syntax it can’t handle (e.g., an unsupported language feature), it should fail gracefully (perhaps create a generic node or skip, rather than crash). Similarly for chat – if something goes wrong (like the reasoning times out), the UI should handle it (maybe respond with “I’m not sure about that” rather than hang).  
      *Checkpoint:* End of Day 12 should see TORI IDE approach a **production-ready posture** for enterprise trials. We have audit logs, some configuration ability, and the system has been combed for major security holes. While formal security testing is ongoing, we addressed obvious issues. This boosts confidence for enterprise demo on Day 16 and beyond.
13. **Day 13 (May 24, 2025) – TORI Chat Standalone Application**  
    *Goal:* Prepare TORI Chat as a separate deliverable (especially for enterprise use outside the IDE), ensuring it works as a standalone conversational assistant with access to “concept packs” and knowledge base.  
    *Tasks:*
    * Set up the **standalone Chat app** environment. If it shares a lot with the IDE, perhaps it’s the same backend but different front-end. Ensure we can run TORI Chat without the IDE UI – e.g., via a command-line interface or a basic web UI (which we have from prototype).
    * Expand the chat UI (web/Electron) to support multi-turn conversations nicely. Implement displaying the conversation history, and allow resetting context, etc. This UI can be simple but should be clean and user-friendly.
    * Test loading a domain **Concept Pack** in standalone chat. For instance, load a sample medical ontology and ask TORI Chat questions in that domain (“What is the procedure for XYZ?”) to see that it uses the injected knowledge. Fix any issues in concept pack integration (like concept collisions or memory usage).
    * Persona support: Implement a way to choose a persona or role for the chatbot. Maybe in the UI a dropdown “Emulate: [General Assistant, Python Guru, DevOps Coach, *Custom*]”. This will prepend an appropriate system message to steer the style. The Attachment agent’s work ties in here if, say, an “Empathetic listener” persona is chosen, the responses shift tone.
    * Ensure that the Chat system respects the privacy toggles (from Day 12 work) – i.e., verify that it indeed does not call out to any cloud API (we aren’t using any by design, but double-check things like spaCy or other libraries aren’t pulling data online unexpectedly).  
      *Checkpoint:* By end of Day 13, **TORI Chat standalone** should be ready for basic use. One should be able to launch the chat app, perhaps select a knowledge pack, and have a coherent conversation. While the IDE and Chat share a core, this independent packaging is important for demonstration to non-developer stakeholders and for enterprise environments where the chat might be deployed as a knowledge assistant on its own.
14. **Day 14 (May 25, 2025) – Multi-Agent Enhancements & Chat Agents**  
    *Goal:* Improve the multi-agent system in the Chat context, and ensure orchestrator synergy between Chat and IDE agents when needed. Use this day also as a secondary buffer/review point for the second week.  
    *Tasks:*
    * Turn focus to **Chat agents**: ensure the Knowledge Retrieval Agent and Reasoning Agent are working together properly. Perhaps implement a more advanced retrieval (like searching not just exact concepts but related ones, maybe using the embeddings to fetch similar concepts when direct answers aren’t found).
    * If the Attachment Agent (emotional reasoning) is to have any effect, implement simple rules: e.g., if user sentiment (we can guess from text, or even just presence of “!” or words like “frustrated”) is negative, the Chat agent will respond with a more apologetic tone. This could be done by having the orchestrator adjust the persona or insert a directive like “The user seems upset, respond supportively.”
    * **MPorter in Chat:** Possibly integrate the strategy agent to manage long conversations. For instance, if the user’s query is very complex, MPorter could break it into sub-questions for the other agents. We might simulate this by handling compound questions or having the chat orchestrator call the Knowledge agent multiple times for parts of the question.
    * Conflict handling in Chat: If two knowledge sources conflict (say concept pack vs code context), orchestrator should detect and either ask user for clarification or choose one based on priority (perhaps code context overrides general knowledge, etc.). Implement a basic policy and test it (“What is X?” where X is defined differently in user’s project vs general knowledge).
    * Use remaining time to fix any critical issues or do another mini-test of full system (both IDE and Chat) ahead of final stretch.  
      *Checkpoint:* By end of Day 14, the multi-agent system for Chat should be more robust and intelligent. TORI Chat can incorporate multiple perspectives or specialized sub-agents (e.g., retrieval vs reasoning) in one conversation without confusing the userfile-bnjjcfgff6iusnbzdwqayo. We should also now have both IDE and Chat components largely feature-complete, setting the stage for final refinements and testing in the last week.
15. **Day 15 (May 26, 2025) – Documentation, Tutorials & UX Polish**  
    *Goal:* Prepare user-facing documentation and improve the user experience with small tweaks and fixes. Make TORI feel more polished.  
    *Tasks:*
    * Write a **quickstart guide** and in-app tutorial hints. E.g., when TORI IDE first launches, show a “Welcome to TORI” message that highlights key areas (concept explorer, chat panel, etc.). Also prepare a README or user manual covering installation, basic usage, and explanation of TORI’s key concepts (for early adopters trying it out).
    * Ensure all UI text is clear and user-friendly. Replace any placeholder labels or cryptic terms. For example, if a suggestion says “Phase conflict in module X,” perhaps rephrase to “Inconsistency detected between module X and Y (they may need synchronization)” – more understandable language.
    * Add tooltips or help icons where needed, especially for novel features. If a user hovers over the oscillator dashboard, a tooltip could explain “This visualization shows the synchronization of key components – when waves line up, components are in sync.”
    * Finalize persona and branding in UI: name, logos, color scheme consistent. Possibly incorporate a small TORI logo or avatar in the chat interface for personality.
    * If time, record or script some example **use-case scenarios** to validate workflow: e.g., “User story: debugging a problem with TORI’s help” – walk through it and ensure the UI/agents support it smoothly. Refine any steps that felt clunky.  
      *Checkpoint:* By end of Day 15, TORI should *feel* more like a polished product rather than a research demo. New users should have some guidance (docs/tutorial) and the overall UX should be refined. We want to be able to hand TORI to a friendly tester and have them understand what to do without devs explaining everything in person.
16. **Day 16 (May 27, 2025) – Internal Testing & Bug Bash**  
    *Goal:* Conduct intensive testing of all features (IDE and Chat), fix bugs, and ensure quality. Essentially a “bug bash” day.  
    *Tasks:*
    * Have team members (and possibly a small group of internal users) use TORI for real tasks for a few hours. Collect their feedback and any issues encountered.
    * Triage bugs: categorize into must-fix for launch vs can-fix-later. Focus on must-fix now.
    * Fix critical bugs across the board: crashes, incorrect analyses, UI misalignments, etc. Pay special attention to anything that could lead to wrong behavior (e.g., an agent suggesting something dangerous in code).
    * Cross-platform test if possible: run on Windows, Mac, Linux to ensure no environment-specific bugs (since local-first means environment differences matter).
    * Test performance in some edge cases (very large file, extremely long chat conversation) to make sure nothing catastrophic occurs (if it does, perhaps implement safety cut-offs, like “Results truncated…”).  
      *Checkpoint:* By end of Day 16, the bug count should be vastly reduced. Ideally, no showstopper bugs remain. TORI is now in a *release candidate* state for both IDE and Chat. We should have a list of known issues that are either minor or will be addressed with lower priority.
17. **Day 17 (May 28, 2025) – Deployment Preparation & Packaging**  
    *Goal:* Prepare TORI for distribution: installer, packaging, environment setup scripts, etc., and ensure it can be easily deployed by users or on servers.  
    *Tasks:*
    * Create an **installer or package** for TORI IDE (Thoughtboard). Possibly using PyInstaller for a one-file executable or creating an Electron app package. Ensure that the ALAN core (Python backend) starts up with the UI seamlessly.
    * Package TORI Chat standalone similarly, if it will be delivered separately (maybe a Docker image or a simple executable that launches the chat server and web UI).
    * Write deployment docs (for IT teams): e.g., how to install on an offline machine, required system resources, how to configure on-prem.
    * Test installation on a fresh system: does it run out of the box? Catch any missing dependencies or config steps and adjust the installer accordingly.
    * If licensing or activation is needed (for enterprise, maybe not yet), at least design where that would hook in. Possibly just stub an “enter license” dialog if appropriate (though might not be needed for v1 tech preview).
    * Ensure logging (from Day 12) goes to proper files in the installed locations and not to console, etc., as expected.  
      *Checkpoint:* By end of Day 17, we should have **installable builds** of TORI IDE and TORI Chat. This means we could send a file (or Docker image) to someone and they could get TORI running without our direct involvement. It’s a key step towards any sort of user testing or pilot program.
18. **Day 18 (May 29, 2025) – Documentation & Final Touches**  
    *Goal:* Finish comprehensive documentation (technical and user), and add any final touches or small features that were deferred but are quick wins.  
    *Tasks:*
    * Complete the **user manual** covering all features of v1.0. Include screenshots of the UI, examples of how to interpret the outputs (like an example of the oscillator dashboard and what it meant in a scenario), and FAQs for common questions.
    * Write a section in docs for “Under the hood: How TORI works” for the curious user or stakeholder, explaining concepts like the concept graph and phase reasoning in approachable terms (this leverages our deep knowledge but presents it simply).
    * Ensure the in-app help links (if any) point to the correct documentation or have tooltips (from Day 15).
    * Implement any **small deferred enhancements** if time permits. For example, maybe adding a keyboard shortcut to open the concept explorer, or a command palette entry for “Ask TORI” to quickly query something without clicking the chat panel. These little touches improve UX if easily done.
    * Final branding pass: ensure the app names, version numbers, and about dialogs are correct (“TORI IDE v1.0 (ALAN 2.x core)” etc.).  
      *Checkpoint:* By end of Day 18, documentation should be **complete and polished**. All those using TORI (developers, early adopters) have the materials to understand and troubleshoot it. The product should have a fit-and-finish that we’re proud to present. We should feel confident that no major feature is undocumented or totally unknown to the user.
19. **Day 19 (May 30, 2025) – Final Testing & UX Review**  
    *Goal:* Do a final end-to-end test pass as if we were the end-user. Refine any last UX issues and make sure we’re truly ready to release.  
    *Tasks:*
    * Perform a scenario-based test: e.g., “Use TORI IDE to improve a small project from scratch,” and “Use TORI Chat to learn about a domain.” Follow these scenarios step by step, noting any awkwardness or minor bugs.
    * Fix *minor bugs* that are quick (typos, alignment issues, log verbosity, etc.). Anything more than minor should already have been caught, but if something appears now that is critical, we still have a bit of buffer to address it.
    * UI/UX refinement: tweak colors, spacing, icons to ensure the interface looks clean and professional. Make sure the visual theme is consistent (fonts, sizes).
    * Confirm all *integrations* work one more time: The chat panel in IDE still works in the packaged build, the concept pack loading in Chat works, etc., in the final packaged environment.
    * By mid-day, *freeze the code*. Enter code freeze for release candidate except for emergency fixes. From this point, focus on stability.
    * If time, do a dry-run of a **demo or presentation**. Usually, we’d want to simulate what we will show stakeholders or potential users and ensure TORI can be showcased without hiccups. This doubles as a final test.  
      *Checkpoint:* End of Day 19, we have what we consider the **release candidate build** of TORI IDE and TORI Chat v1.0. The team should feel confident in doing a live demo to stakeholders without making excuses for rough edges. We likely will tag this version in our repo as v1.0-rc.
20. **Day 20 (May 31, 2025) – Buffer, Polishing & Pre-Launch Prep**  
    *Goal:* Use this day as a buffer for any last-minute critical issues and prepare for launch activities (like presentation, marketing collaterals).  
    *Tasks:*
    * If any critical bug was found late Day 19 or by additional testers (perhaps someone from another team trying it on Day 19 evening), fix it with priority.
    * Finalize the **presentation/demo script** for the launch meeting or recording. Ensure all materials (slides, if any, or sample projects to demo on) are ready.
    * Create some **marketing content** if applicable at this stage: perhaps a one-pager summary of TORI’s value prop to accompany the release, or internal announcement text. This may involve working with a product or marketing person, but the technical team provides inputs (features, differentiators).
    * Double-check that all **SWOT points** have been addressed in some form in the product or documentation (e.g., weakness of limited language support is documented and openly acknowledged with plans, etc., so it doesn’t catch anyone by surprise).
    * Rest (if everything is truly done). The team has pushed hard; if all is in order, a lighter day to recuperate ensures we approach launch with clear minds.  
      *Checkpoint:* End of Day 20 should have zero critical open issues. We are essentially ready to launch the product the next day. All that remains is the actual launch event or final sign-off.
21. **Day 21 (June 1, 2025) – Sprint Completion & Launch Readiness**  
    *Goal:* Final day to wrap up the sprint, do any last reviews, and *prepare for the launch/presentation*. If possible, also allow the team a brief respite before the big day.  
    *Tasks:*
    * **Final sanity test:** Run through the core functionality one more time quickly in the morning. If anything unexpected occurs, decide if it must be fixed now or can be deferred (ideally deferred unless it’s truly showstopping, as we shouldn’t be making code changes on launch day).
    * Package the final release builds, double-check their integrity (hashes, can be installed fresh, etc.).
    * Ensure all documentation and materials are accessible (upload to internal site or include in the installer package as appropriate).
    * Team meeting to recap the sprint accomplishments, and discuss support plan post-launch (e.g., how to handle user feedback, quick patches if needed).
    * If this is an internal launch, coordinate with any stakeholders on the launch event details. If external, perhaps push the repository to a private share or similar.
    * **Rest and mental preparation:** Encourage the team to take it easy in the latter half of the day if everything is done. We want everyone fresh for the launch or next phase.  
      *Checkpoint:* By end of Day 21, the 21-day sprint is **successfully completed**. TORI IDE and ALAN 2.x (with TORI Chat) are ready for deployment/presentation. The team can confidently say we have met the goals of the sprint and are prepared to introduce TORI to its initial users.

**TORI Chat – 21-Day Sprint Plan (May 12 – June 1, 2025)**

*(The TORI Chat development ran in parallel to the IDE sprint, with some shared efforts on core integration, multi-agent system, etc. The numbering here mirrors the days, but many Chat tasks were done concurrently by a sub-team focusing on the chat interface and capabilities.)*

**Sprint Goal:** Evolve TORI Chat from a rudimentary prototype into a robust conversational assistant with enterprise-ready features and seamless integration with the TORI cognitive core. Ensure TORI Chat can operate both embedded in TORI IDE and as a standalone product.

1. **Day 1 (May 12, 2025) – Chat Sprint Kickoff & Environment Setup**  
   *Goal:* Align on TORI Chat objectives and set up the development environment for the chat application.  
   *Tasks:*
   * Kickoff meeting in sync with the IDE team, focusing on Chat-specific deliverables (like persona handling, concept packs, etc.). Ensure cross-team understanding of what components are shared.
   * Set up the chat server framework (likely extending the same ALAN backend). If using a web UI for chat, set up the development environment for that (Node/Electron or pure web).
   * Ensure the concept network and core from ALAN can be accessed in a stateless way (since chat queries might be one-off calls to the core). Prepare scaffolding for maintaining chat conversation state (likely a context object storing recent conversation or a pointer to relevant concept subgraph).
   * Merge any baseline code for chat from prototypes into main, analogous to the IDE merging step.  
     *Checkpoint:* Chat team has a running development instance of TORI Chat (even if it’s just echoing or a basic template) and is in sync with the overall project plan. Environment issues resolved.
2. **Day 2 (May 13, 2025) – Core Alignment for Chat (Oscillators & Memory)**  
   *Goal:* Tie the updated core (oscillator sync, etc.) into the chat flow. Ensure that chat inputs create proper concept activations and that the conversation context is maintained in the core.  
   *Tasks:*
   * Modify the NLP pipeline so

**Comprehensive Master Plan and Strategic Analysis – TORI IDE & ALAN 2.x Cognitive OS**

**Executive Summary**

**TORI** is a next-generation *cognitive operating system* comprising two flagship products: **TORI IDE** (an AI-augmented Integrated Development Environment) and **TORI Chat** (an intelligent chat assistant, including an enterprise-grade variant). This master guide serves as a comprehensive report, technical specification, development roadmap, and marketing playbook for the TORI suite. It encapsulates the visionary paradigm behind TORI, the current architecture and module breakdown, a 21-day sprint plan for rapid development, SWOT analyses, and go-to-market strategies for each product.

* **Vision:** TORI embodies a paradigm shift from traditional AI tools. Rather than relying on rote token prediction, TORI’s core is grounded in **concepts over tokens** and \**memory over stateless prediction*[medium.com](https://medium.com/syncedreview/from-token-to-conceptual-the-rise-of-metas-large-concept-models-in-multilingual-ai-b32acbfeb792#:~:text=generalization%20across%20languages%2C%20outperforming%20existing,LLMs%20of%20comparable%20size)】. The vision is an “AI-native” development experience in which the system truly understands code and conversations at a conceptual level, enabling unprecedented levels of assistance and insight. Developers move from using tools to collaborating with a *cognitive partner*.
* **Architecture:** At TORI’s heart is the **ALAN 2.x cognitive engine**, a spectral-phase reasoning core that maintains a rich **Large Concept Network (LCN)** of knowledge. This core powers both the IDE and Chat, providing deterministic, auditable reasoning steps instead of opaque neural guesswor[ibm.com](https://www.ibm.com/think/topics/black-box-ai#:~:text=White%20box%20AI%2C%20also%20called,Users)】. Surrounding the core are modular subsystems for language processing, user interaction, and multi-agent orchestration. In the IDE, this means deeply understanding code structure and evolution; in Chat, it means maintaining contextual, persona-driven dialogue. Both products share a **local-first, privacy-conscious design** that can operate on-premises without cloud dependence.
* **Current Status:** The TORI codebase has been through early prototyping phases. Key modules for concept parsing, memory graph management, and a basic front-end exist, though some components remain mock or placeholder implementations pending integration of the full ALAN engine. For example, a document ingestion pipeline successfully indexes PDFs into the concept network, but the **phase oscillator network** underpinning dynamic reasoning is only *partially implemented*. The current TORI IDE UI is functional as a prototype “**Thoughtboard**” dynamic code canvas, albeit without full polish or all expected IDE features. TORI Chat’s core reasoning back-end is in place, but its user-facing chat interface and multi-agent persona system are at a rudimentary stage. These gaps are identified in the module-by-module analysis, and the plan is to replace placeholder code with production-ready logic during the upcoming development sprints. *Before sprinting on new features, completing these core components (oscillator sync, spectral engine, ELFIN parser, orchestrator) is imperative, to ensure the foundation is solid.*
* **Roadmap:** A detailed **21-day sprint plan** for each product outlines the path from the current state to a viable v1.0 release. Each sprint plan is organized into daily milestones with one rest/review day built-in per week. (For reference, **Day 1 corresponds to May 12, 2025**.) The TORI IDE plan focuses on integrating the ALAN core with the IDE front-end, implementing conceptual debugging tools, and eliminating any “mock” data paths. The TORI Chat plan emphasizes developing a robust conversational interface, domain-specific **“concept pack”** integration, and enterprise features like audit logs and on-prem deployment support. Both plans culminate in a synchronized launch, ensuring the IDE and Chat leverage each other (e.g. the IDE generating chat-ready knowledge graphs, and the Chat agent assisting within the IDE).
* **SWOT Highlights:** TORI’s **Strengths** lie in its unprecedented approach: a phase-synchronized reasoning core that yields deterministic, explainable insights (a “glass-box” AI approach[ibm.com](https://www.ibm.com/think/topics/black-box-ai#:~:text=White%20box%20AI%2C%20also%20called,Users)】, and a local-first architecture that sidesteps the compliance and cost issues of cloud-only AI solutions. **Weaknesses** relate to typical early-stage challenges: limited UI polish compared to incumbent tools, narrow language support at launch (only Python and the custom ELFIN DSL are first-class), and the need for extensive real-world validation of its novel techniques. **Opportunities** include growing demand for on-premises AI solutions (for privacy and control), and synergy between development and conversational AI (TORI uniquely bridges these). **Threats** include rapid moves by major IDE vendors to integrate similar AI capabilities, and skepticism toward unproven AI paradigms in conservative industries. Our strategy and roadmap specifically address these points through aggressive development, transparency, and engagement with early adopters.

**Vision and Guiding Principles**

**TORI’s vision** is to create a *cognitive operating system* that transforms how we interact with software development and knowledge work. This vision is grounded in two core ideas: focusing on **conceptual understanding** rather than surface-level output, and building **persistent knowledge** rather than stateless interactions. Traditional AI coding assistants operate at the level of syntax and tokens, predicting text without true understanding. TORI takes a different path – it builds and manipulates rich *conceptual models*. In the IDE, this means understanding the abstract architecture and intent behind code; in Chat, it means grasping the semantics and context of a conversation, not just producing plausible sentences. By elevating to the conceptual level, TORI provides insights and assistance that feel genuinely intelligent and context-aware, rather than generative but shallow. As one early reviewer noted, TORI represents not just a toolset, but *a paradigm shift in how we think about code, conversations, and AI assistance*.

In addition, instead of treating each user query or coding session as an isolated event, TORI maintains a **persistent memory** of interactions and learned concepts. This is not simply a chat history or cache, but a structured memory in the form of the LCN, phase states, and accumulated patterns. TORI remembers *why* a piece of code was written or *how* a conclusion in chat was reached, enabling it to explain its reasoning and build on past knowledge. This contrasts with stateless large language model tools that have no inherent memory of past interactions. TORI’s approach yields an AI that can truly learn and adapt over time – accumulating **wisdom** rather than resetting after each prompt.

From this vision, we derive a single unifying goal: **TORI is not just a development tool or a chatbot; it is a thinking partner that grows with the user.** The IDE is envisioned as a *“Conceptual Canvas”* where code and design interplay dynamically with the AI’s understanding. The Chat is envisioned as a multi-mind conversational agent that can reason through complex problems with traceable logic. Together, they form a unified cognitive workbench.

Several **guiding principles** direct our design and implementation:

* **Human-Centric Design:** TORI keeps the human user in control and informed at all times. We adhere to the principle of *“human in the loop”*. Features like explainable suggestions, the ability to inspect the concept graph, and toggles for AI assistance are built-in. The user should feel TORI is *amplifying* their abilities, not automating tasks away without consent or clarity.
* **Transparency and Explainability:** Every AI-driven action by TORI is explainable via its concept graph or a rationale. For example, if TORI IDE suggests a code refactor, it can point to the specific concepts and oscillator signals that led to that suggestion (e.g., a function concept that is out-of-phase with related functions, indicating a design inconsistency). Similarly, TORI Chat doesn’t just answer questions – it can provide a brief explanation or source (e.g., which internal knowledge or which agent’s logic contributed to the answer). This fosters trust and positions TORI as a “glass-box” AI, in contrast to opaque *black-box* model[ibm.com](https://www.ibm.com/think/topics/black-box-ai#:~:text=White%20box%20AI%2C%20also%20called,Users)】.
* **Deterministic Core, Creative Edges:** TORI’s architecture emphasizes *deterministic reasoning* in its core (the spectral-phase logic, rule-based multi-agent decisions, etc.) to ensure reproducibility and auditability. However, we allow for stochastic creativity at the periphery. For instance, a language model may be used to generate natural language summaries or code comments under TORI’s guidance. The rule is to use randomness only where it’s safe and adds value (e.g., phrasing a response kindly), but never in the core understanding of the code or conversation. This yields a system that is both **reliable** and **inventive** – core logic is stable and verifiable, while user-facing interactions can still have the flexibility and richness of generative AI.
* **Local-First and Privacy-Preserving:** TORI is built with a *local-first* mindset. Users retain complete ownership of their code, data, and the AI’s knowledge about them. Both TORI IDE and TORI Chat can run fully locally or on a private network. No data leaves the user’s machine unless explicitly configured to (for example, if the user chooses to enable cloud connectivity or external APIs). This addresses enterprises’ needs to keep proprietary code and discussions in-house. It also has a technical upside: operating locally yields lower latency and tighter integration with the user’s environment (files, processes, dev tools) since TORI can directly interface with them without network overhead.
* **Modularity and Extensibility:** TORI’s cognitive OS is modular by design. Each subsystem – parsing, concept storage, reasoning agents, UI – is a separate module with well-defined interfaces. This allows future improvements or swaps (e.g., plugging in a new parser for a different language, or adding a specialized agent) without overhauling the entire system. It also means parts of TORI can be extended by third-parties; for instance, developers might create new ALAN agents or integrate TORI’s concept graph with external tools via APIs. This modularity also aids maintainability: given TORI’s broad scope, we must be able to update one component (say, the NLP pipeline) independently of others.
* **Consistency with Developer Workflow:** Especially for TORI IDE, we design to *“meet developers where they are.”* This means TORI’s enhancements are built into familiar workflows. The IDE still allows editing code as text, using Git for version control, running tests, etc. – but TORI augments these activities with cognitive insight. For example, a developer can code normally and optionally inspect TORI’s concept graph for deeper understanding, or receive proactive warnings about design issues. TORI Chat is designed to integrate into the developer’s environment too (for instance, accessible as an in-IDE chat panel or via tools like Slack through integrations) so that conversing with TORI feels like a natural extension of their workflow. The guiding motto is *“low floor, high ceiling, wide walls”* – easy to get started, immense power as you explore, and broadly applicable to many tasks.
* **Inspiration and Playfulness:** While TORI is grounded in rigorous concepts from cognitive science and spectral theory, the project embraces creativity and approachability. This appears in our internal code names (e.g., the oscillator core nicknamed “**Banksy**” for its artful sync patterns, conceptual caches called “cartridges,” etc.) and in the product’s personality. TORI aims to feel engaging, even fun – we believe a degree of playfulness encourages exploration and learning. For example, TORI might include easter eggs or a witty persona remark (tastefully and only when appropriate) to remind users this AI is *approachable*. This helps differentiate TORI: it’s advanced technology with a human touch and a bit of charm, not a cold enterprise robot.

Balancing these principles ensures that as we build TORI, we maintain both **intellectual rigor** (truly innovating and solving hard problems in AI reasoning) and **user empathy** (crafting an experience that empowers and delights). The following sections translate this vision and these principles into TORI’s concrete architecture and development plan.

**Cognitive OS Architecture Overview**

TORI’s architecture is a modular, layered cognitive system designed to support both the IDE and Chat applications through a shared intelligent core. At a high level, the architecture consists of four integrated subsystems:

* **ALAN 2.x Cognitive Core:** The “brain” of TORI – responsible for conceptual understanding, inference, and maintaining the knowledge network.
* **Language/Domain Integration Layer:** Bridges the core with domain-specific inputs (code, documents, natural language) and outputs, translating between raw data (files, text) and TORI’s concept representations.
* **User Interface & Tools:** The front-end components through which users interact with TORI (the Thoughtboard IDE UI, debugging and visualization tools, chat interfaces, etc.).
* **Multi-Agent Orchestration:** The layer managing a collection of specialized AI agents and coordinating their efforts via the shared core (the “mind society” within TORI).

Each of these is detailed below, followed by how they collaborate in the TORI IDE and TORI Chat contexts.

**ALAN 2.x Cognitive Core (Conceptual Engine)**

At the heart of TORI lies the **ALAN 2.x cognitive core**, which maintains a rich, interconnected representation of knowledge called the **Large Concept Network (LCN)**. This core can be thought of as TORI’s brain: it encodes concepts (from code constructs to conversation topics) as nodes in a graph and relationships between those concepts as edges. Beyond static relationships, ALAN 2.x introduces a novel *spectral-phase inference mechanism* that enables dynamic reasoning over time.

In essence, the core treats reasoning as a live process of **synchronization and resonance** among concepts. Every concept node in the LCN is accompanied by an internal state, including a **phase oscillator** representing its current “state of mind” or context. When the system processes new information or code changes, these oscillators are perturbed and evolve. The core monitors the collective behavior of all concept oscillators to identify emerging patterns of coherence or conflict. Intuitively, if two concepts should logically agree (say two functions that need to work in concert), the system expects their oscillators to synchronize. If they drift apart (out of phase), that signals a potential inconsistency or conflict.

This approach draws inspiration from **neural oscillation theories** in cognitive science – where synchronized neural firing is thought to underlie unified perceptions and thought[neurosciencenews.com](https://neurosciencenews.com/brain-rhythm-cognition-25941/#:~:text=It%20could%20be%20very%20informative,%E2%80%9D)】. By analogy, TORI’s synchronized concept oscillators imply a coherent understanding or stable solution, whereas desynchronized oscillators flag issues requiring attention. For example, if a function’s implementation is inconsistent with its specification, the concept representing the implementation may oscillate out-of-sync with the specification concept, alerting TORI that something is off.

**Phase-Coupled Oscillator Matrix (“Banksy Core”):** The engine that manages these oscillators is nicknamed *Banksy* for its focus on rhythmic, artful coordination. Technically, it’s a matrix of coupled oscillators associated with the concept graph. In earlier prototypes we considered using a standard Kuramoto model for synchronizing these oscillators, but we have since moved to a more advanced **ψ-based coupling approach** that leverages spectral analysis for stability and alignment. In this approach, each concept oscillator’s update is influenced not just by direct neighbors (related concepts), but by a global perspective from the system’s spectral mode analysis (explained shortly). The goal is to achieve robust consensus where appropriate, and deliberate divergence where a conflict genuinely exists (as opposed to noise causing false alarms).

Each concept oscillator has a phase (and possibly amplitude) that can be interpreted: two concepts in phase indicate they are in harmony or agreement; out-of-phase might indicate discrepancy or different “contexts.” By adjusting coupling strengths and rules (the core might couple oscillators of related concepts strongly, and unrelated ones weakly or not at all), TORI dynamically enforces a consistency check across the entire knowledge network. If concept A (e.g., a variable’s expected type) and concept B (the variable’s actual usage) are linked, the oscillator engine will try to keep A and B in sync. A sustained phase difference would mean the engine cannot reconcile them – prompting, for example, a warning to the user about a type mismatch. Through this mechanism, TORI’s core provides a continuous, interpretable measure of consistency across the system.

Importantly, the simplistic Kuramoto coupling has been replaced with a more *intelligent synchronization strategy*. We incorporate insights from **Koopman operator theory** and control theory so that the oscillators not only sync or desync, but do so in a stable manner. The system computes collective properties (like an order parameter indicating overall sync) and uses feedback to ensure stability (preventing chaotic swings). It also means we can identify clusters of synchronization – perhaps a subset of concepts oscillate together (indicating a coherent subsystem or topic), separate from another cluster. This clustering can reveal modular structure in code or conversation.

*(Status:* As of now, the phase-coupled oscillator network is partially implemented. The basic data structures exist and oscillators update in a rudimentary way, but the full ψ-based synchronization algorithm is being finalized. Completing this and fully integrating it into the runtime loop is a top priority in the development plan.)

**Koopman Morphic Engine (Spectral Analyzer):** Complementing the time-domain oscillator view, ALAN 2.x employs a **Koopman operator-based spectral analyzer** to study the evolution of the concept network state in a higher-dimensional space. The **Koopman operator** is a theoretical construct that allows us to treat a nonlinear dynamical system (like our knowledge state updates) via an infinite-dimensional linear operator acting on observation function[mdpi.com](https://www.mdpi.com/2227-7390/9/19/2495#:~:text=The%20Koopman%20operator%20theory%20is,are%20amenable%20to%20systematic%20analysis)】. In simpler terms, we lift the system’s state into a space of features (potentially complex eigenfunctions denoted ψ) where the dynamics can be approximated as linear. By doing so, we can apply linear spectral analysis: find eigenvalues and eigenvectors (modes) that describe recurring patterns or steady states in the reasoning process.

Practically, the Koopman engine monitors changes in the concept graph (like concept activation levels, oscillator phases) over time or iterations. It then tries to identify **spectral modes** – for instance, an eigenmode might correspond to a particular pattern of concept activation that repeats or a combination of concepts that tends to increase or decay together. The eigenvalues associated with these modes can tell us about stability: eigenvalues on the unit circle imply a repeating cycle, eigenvalues less than 1 in magnitude indicate a decaying mode (converging to stable state), and greater than 1 would indicate a growing instability. This analysis provides TORI with a predictive foresight. For example, if a particular conceptual inconsistency tends to amplify each iteration (an unstable mode), the system can flag it early as something that will likely cause problems if not addressed (like a bug that will cascade). If a mode corresponds to a settled understanding (convergence), TORI knows those aspects of the problem are resolved and stable.

In essence, the Koopman engine allows TORI to reason not just in the now, but about the *trajectory* of the reasoning process. It can answer questions like: *Is the system of concepts converging to a coherent understanding?* or *Are there oscillatory disagreements that persist?* It’s analogous to how an experienced engineer might sense if a project’s design discussions are converging or endlessly oscillating between options. Mathematically, it provides a global, linearized view of the nonlinear interactions happening in the cor[mdpi.com](https://www.mdpi.com/2227-7390/9/19/2495#:~:text=The%20Koopman%20operator%20theory%20is,are%20amenable%20to%20systematic%20analysis)】.

To ensure reasoning remains controlled and safe, we incorporate **Lyapunov stability measures** into this analysis. A Lyapunov function (think of it as an energy or potential function for the system) is used to prove stability of an equilibrium. In TORI’s context, we design Lyapunov-like metrics to ensure our reasoning process tends toward a stable equilibrium (a consistent state) and not diverge. If the system’s computed Lyapunov function is decreasing over reasoning iterations, it indicates the system is *stable in the sense of Lyapunov*, meaning small perturbations (like minor code edits or rephrasing of a question) will dampen out and not blow u[ocw.mit.edu](https://ocw.mit.edu/courses/16-30-feedback-control-systems-fall-2010/463e9559c6ef70f13ea51c3464bdc9c1_MIT16_30F10_lec22.pdf#:~:text=%E2%80%A2%20Stable%20in%20the%20sense,%E2%80%9D%20November%2027%2C%202010)】. The orchestrator can use these metrics to decide when to stop iterating on an internal problem (e.g., the agents have reached a consensus when the system is stable).

*(Status:* A scaffold of the spectral analysis exists – it can take snapshots of concept states and compute a basic decomposition (e.g., via an SVD or FFT). Currently it returns dummy data for testing the pipeline. Implementing the full Koopman-based analysis and integrating real stability checks is in progress. It’s slated for completion early in the development sprint, after which the core will regularly generate these spectral insights to guide reasoning.)

**Static Knowledge Base & Symbolic Reasoning:** Alongside the dynamic oscillator and spectral components, the ALAN core also maintains a static knowledge base and performs more traditional symbolic reasoning. For example, TORI stores facts and rules (like “if concept X implies concept Y, and X is true, then Y is true”) within the LCN. It can run logical inference on this static graph as needed. Moreover, TORI uses *embedding-based similarity* to relate concepts: each concept may have a vector embedding (learned from code context or language data) so that semantically similar concepts can be identified via cosine similarity. This is useful for suggestions (like finding analogies in code or clarifying ambiguous questions in chat). These static or precomputed relationships act as a baseline layer of “common sense” and domain knowledge, upon which the dynamic reasoning operates.

**Determinism and Auditability:** A key design choice is that, given the same inputs and initial state, the ALAN core’s reasoning process is *deterministic*. That means TORI will produce the same sequence of internal states and same outcomes every time for reproducibility. The apparent randomness or creativity in output (like phrasing) is separated out (e.g., handled by a language model at the very end, if at all) and does not affect the underlying reasoning trace. We log the sequence of reasoning steps (concept activations, agent decisions, etc.) as an **audit trail**. This allows users (especially enterprise users) to inspect *why* TORI arrived at a suggestion – they can trace through a sequence like: *“Function A’s concept was out-of-phase -> Refactoring agent noted duplication with Function B -> Suggested to unify them.”* This level of introspection is rarely possible with traditional AI assistants and is a differentiator for TORI’s cognitive OS approach.

**Language & Domain Integration Layer**

The Language & Domain Integration Layer is responsible for translating real-world inputs into TORI’s conceptual form and vice versa. It ensures that whether the input is source code, a natural language sentence, or a formal specification, TORI can incorporate it into the LCN; and when TORI produces an output (like an answer or code change), it can be rendered in a human-usable format.

**Programming Language Frontends (for TORI IDE):** TORI IDE supports multiple programming languages by using frontends that parse code into the LCN. For each supported programming language, we employ either a custom parser or an existing parser/AST generator to break code down into its structure. For example, for Python, TORI uses the Python AST module to get the syntax tree, then traverses it to create concept nodes for entities like functions, classes, variables, as well as relationships (e.g., “function *foo* calls function *bar*” becomes a link between the concept nodes for *foo* and *bar*). We attach additional semantic information when available: docstrings become documentation concept nodes, types (if inferred or declared) become type-concept links, etc. Our custom DSL ELFIN (detailed below) has its own parser that directly produces a rich concept graph (since ELFIN is designed to express high-level concepts natively).

For other languages like C/C++ or JavaScript/TypeScript, we integrate with established parsing tools (e.g., tree-sitter or language server protocols) to get an AST or symbol table, then map those into TORI’s concepts. In these cases, because we may not capture every nuance, the integration might be a bit shallower (for instance, TORI might not fully understand C++ template metaprogramming yet, but it will still record the relations it can, like function calls and inheritance). Our design makes Python and ELFIN *first-class* (with the richest understanding), while still supporting other popular languages sufficiently to be useful. Over time, we plan to extend first-class support to more languages.

Whenever code is edited or a new file is added, this layer updates the LCN incrementally. For example, adding a new function in code creates a new concept node and appropriate links without rebuilding the entire graph from scratch. This integration is efficient and continuous, feeding changes into the ALAN core’s reasoning loop (which then perhaps triggers re-synchronization or agents to check the implications of that new function).

**Domain Knowledge Ingestion (for TORI Chat and Enterprise):** TORI’s usefulness amplifies when it’s informed about the user’s domain. The integration layer can ingest structured knowledge bases such as **ontologies**, **taxonomies**, or corporate knowledge graphs and integrate them as part of the LCN. We call these packaged knowledge sets **Concept Packs**. For instance, an enterprise could load a “Financial Regulations Pack” that contains concepts like regulations, compliance procedures, definitions of banking terms, etc., all connected in a graph. TORI merges this with its own concept network (or keeps it in a linked subgraph) so that both TORI IDE and Chat can leverage that domain knowledge.

In the IDE, this could mean recognizing that a certain variable or function relates to a concept in the ontology (e.g., variable customerAccount is linked to the “Customer Account” concept in a finance ontology). In Chat, it means if the user asks “What is the KYC process for a new client?”, TORI has the concept of “KYC (Know Your Customer)” in its graph and can reason with it directly, producing a knowledgeable answer. The ingestion process typically involves reading a structured file (like OWL or CSV data for ontologies) and creating concepts and relations accordingly. We preserve any hierarchy (e.g., is-a relationships become concept edges) and key attributes (which might become metadata on concept nodes).

For unstructured but relevant data (like a company’s internal PDFs or manuals), TORI can use an offline process to index those into the concept graph as well (essentially performing entity extraction and linking to build a mini knowledge graph of the documents). This was prototyped with the PDF ingestion pipeline mentioned in the current status. TORI Chat can then retrieve and cite that information during conversation.

**Natural Language Processing (for TORI Chat):** When TORI Chat receives a user query in plain English (or other supported languages), the integration layer’s NLP component kicks in. It parses the sentence to understand its structure (using, for example, a combination of dependency parsing and named entity recognition). Key elements of the query are mapped to TORI’s known concepts. For instance, if the question is “How does the encryption function work in the login module?”, the NLP will identify “encryption function” and “login module” as likely referring to concepts. It will query the LCN to see if concepts exist that match those names or synonyms. If yes (say there’s a concept for a function encryptPassword and a concept for module Login), those will be activated and handed to the core to focus on. If not, the NLP can create placeholder concepts (to represent the user’s query topics) so that the reasoning engine can still work with them (perhaps the reasoning agent will then say “I don’t have information on encryption in login”).

The NLP also handles context continuation – TORI Chat keeps track of the conversation so far (via the LCN as well, where each utterance can be a concept or linked to topics). Pronouns or implicit references (“Can you explain *it* further?”) are resolved by looking at the context concepts (what is *it* likely referring to given what was just discussed). Because TORI uses a concept graph for memory, context handling is robust: instead of raw text, it knows the subject under discussion.

For generating responses, TORI has two pathways: If the answer can be constructed logically (like a series of steps or facts), TORI can compose it from the concept graph relations (ensuring each statement is grounded in a known concept or link). Alternatively, for more open-ended answers or simply to make the output fluent, TORI can utilize a template or a controlled language model to turn the structured answer into a natural-sounding one. The priority is always correctness and faithfulness to the underlying reasoning. We avoid any generative fluff that isn’t backed by the LCN or agents’ outcome. The user can, if desired, ask TORI Chat *why* it gave an answer, and the system can then reveal the chain of concepts or rules (this might be presented as a graph visualization or a step-by-step explanation, leveraging our audit trail).

**Bidirectional Interaction:** Integration isn’t just one-way. TORI not only ingests data into the concept graph, but can also effect changes based on its reasoning. For example, in the IDE, an agent might decide a piece of code should be refactored. TORI could (with user permission) perform that refactoring: the integration layer would then take the concept changes (like “Function X and Y merged into function Z”) and apply them to the actual source code (pretty-printing the new code via an AST transformation). Similarly, in Chat, if TORI comes to a conclusion that requires retrieving more info (like “to answer that, I need the user’s account history”), an agent might prompt the integration layer to fetch a file or call an API if allowed. These actions are mediated through well-defined interfaces so that security and permissions are respected (TORI will not perform an action unless configured to, especially in Chat – it won’t execute arbitrary code or calls just from a question, unless explicitly authorized as, say, a tool-using extension).

*(Current Implementation:* The integration layer is in a mixed state of completion. The Python and ELFIN language frontends are largely operational – they produce the concept graph for code and update it on changes. The C/C++ and JS integrations are basic (parsing via external tools works, but deeper semantic mapping is limited). The NLP pipeline for English is functional at a basic level (entity extraction and mapping to known concepts), using libraries like spaCy for parsing. Domain concept pack ingestion is prototyped (we tested with a small ontology, which TORI successfully merged into its graph). Much of the integration logic will be refined during the sprint as we encounter more real-world cases from user testing. Ensuring that this layer robustly handles imperfect input (e.g., code with syntax errors, or poorly worded questions) and interfaces smoothly with the rest of the system is an ongoing effort.)

**The ELFIN DSL: A Domain-Specific Language for Cognitive Coding**

One special element of TORI’s architecture is the inclusion of **ELFIN**, a custom domain-specific language designed alongside the ALAN core. ELFIN (an acronym loosely standing for *Extensible Language for Integrative Networks*, though informally named to suggest something small and magically helpful) serves as both a demo of TORI’s concept-oriented approach and a tool for users to explicitly script cognitive operations.

**Purpose and Philosophy:** ELFIN’s purpose is to allow developers to write code or specifications in a way that directly informs TORI’s concept model. Traditional programming languages require TORI to infer the concepts and intents from the code. With ELFIN, the language is constructed so that the *concepts are first-class citizens*. In practice, this means ELFIN syntax might allow the programmer to declare intentions or relationships that would be comments or implicit in other languages. For example, one might write in ELFIN:

arduino

Copy

concept DataPrivacy relates UserData, AccessControl

process scrubData(input UserData) ensures no PersonalInfo in output

In this hypothetical snippet, the programmer explicitly introduces a concept DataPrivacy and relates two other concepts to it, and defines a process with a postcondition. TORI would take this and create a DataPrivacy concept node linking UserData and AccessControl, and for the process scrubData, it knows a guarantee (no personal info in output) as a concept relation (perhaps linking the output concept to a “sanitized” concept).

Such expressiveness is valuable in domains where requirements and logic need to be very clear (financial contracts, safety-critical systems, etc.). ELFIN allows capturing those as part of the code itself, eliminating the gap between documentation and implementation. It’s similar in spirit to languages like Alloy or Z (formal specification languages), but integrated into TORI’s runtime reasoning. Indeed, one use-case for ELFIN is writing **specifications or rules** that TORI’s agents can then use. For example, an ELFIN rule might specify a design constraint (“every database query must go through the DataAccess layer concept”) and TORI’s Refactoring or Compliance agent can continuously check that this holds in the code.

**Integration with TORI:** The integration layer treats ELFIN as a first-class language. We have a defined grammar (encoded in something like a ANTLR or a custom parser) that directly produces concept graph structures. Unlike other languages, where we have to interpret the AST into concepts somewhat heuristically, ELFIN’s grammar is essentially *one-to-one mapped* with concept graph operations. If you define a concept or a relation in ELFIN, the parser calls the LCN API to create that node or edge. If you define a process or function, it creates a concept for it and also perhaps a state machine or logical relation for its pre/post conditions.

Because ELFIN is under our control, we can evolve it in tandem with features in ALAN. For instance, if we introduce a new kind of reasoning artifact (say, a “scenario” concept for test cases), we could extend ELFIN to allow writing those scenarios in code. ELFIN thus serves the dual purpose of showcasing TORI’s capabilities and giving power users a way to script or influence TORI’s reasoning. A developer could write an ELFIN snippet to guide the AI: *“Whenever concept X and Y are used together, treat them as interchangeable”* – effectively giving TORI a new rule on the fly.

**Current State:** ELFIN is currently in a prototype stage. The initial grammar is defined, covering core constructs like concept declarations, relationships, simple process definitions, and invariants. The parser can handle these and we’ve tested it on small examples (the resulting concept graph matches expectations). What remains is fleshing out the standard library of ELFIN (predefined concept types, perhaps some syntactic sugar) and integrating the ELFIN runtime semantics. Since ELFIN can describe processes, if one writes an algorithm in ELFIN, we might want to actually execute it or at least verify it. Executing ELFIN code could either mean translating it to Python/another language or interpreting it within TORI (likely out of scope for v1.0 except for trivial cases). For now, ELFIN’s emphasis is on *declarative knowledge*. Users can embed ELFIN snippets in comments or separate files to enhance TORI’s understanding of the system.

We include documentation for ELFIN in the TORI user guide as an advanced feature. We expect early adopters might experiment with it, giving us feedback. Over time, if ELFIN proves useful, it could become a cornerstone of how users explicitly communicate with the AI (almost like an API for knowledge).

**Example:** To illustrate ELFIN, consider a scenario where an enterprise wants to enforce data privacy rules in code. In ELFIN, they write:

cpp

Copy

concept PersonalData

concept EncryptedData

rule "Personal data must be encrypted at rest":

forall x: PersonalData, storageFunc f:

uses(f, x) -> ensures(EncryptedData)

This fictional syntax declares two concepts and a rule: any usage of personal data in a storage function must ensure it’s encrypted. TORI would incorporate this rule into the concept graph (perhaps as a constraint relationship). The Compliance agent, when scanning code, would then look for any function concept labeled as a storageFunc using a concept that is a PersonalData, and check if the output or result is marked as EncryptedData. If it finds a violation, it can flag it. Without ELFIN, such a domain-specific rule would be hard-coded or not known to TORI; with ELFIN, the user has effectively programmed the AI’s reasoning.

This level of customization is part of TORI’s long-term vision: an AI platform that organizations can directly imbue with their policies and knowledge through a language interface, rather than just hoping the AI “learned” it from training data. ELFIN is our first step in that direction.

**User Interface & Tools**

The User Interface and Tools layer encompasses the ways users interact with TORI’s intelligence. This includes the **TORI IDE “Thoughtboard” application**, various UI panels within it (like the concept explorer, spectral dashboard), and the **TORI Chat interfaces** (integrated and standalone).

The guiding design for UI is that it should present TORI’s capabilities in a clear, non-intrusive way, enhancing the user’s workflow without overwhelming them. The UI is also where we surface the inner workings of TORI in an understandable form (closing the loop on explainability).

**TORI IDE – Thoughtboard Interface:** TORI IDE is delivered as an independent desktop application (nickname: *Thoughtboard*). It provides a familiar coding environment augmented with TORI’s cognitive features. Key components of the Thoughtboard UI:

* **Code Editor with Inline AI Annotations:** The main editor looks and feels like a modern text editor/IDE (supporting syntax highlighting, code completion via perhaps existing language servers). TORI’s contributions come as subtle overlays: e.g., underlining code segments that TORI finds semantically problematic (not just syntax errors). If an agent has a suggestion or insight about a line or function, a small icon or colored underline appears. The developer can hover to see a tooltip with TORI’s note (e.g., “This function duplicates logic from function Y.”) or click to get a detailed explanation/fix.
* **Concept Graph Explorer:** A sidebar or pop-up panel that visualizes the portion of the concept graph relevant to the current context. For instance, if the developer is editing a function, they can open the Concept Explorer to see all concepts related to that function (variables, other functions it calls, requirements, etc.) in a node-link diagram. They can navigate this graph: clicking a concept jumps the editor to the corresponding code or brings up its documentation. The graph can be filtered or expanded – e.g., “show me the chain of concepts connecting this module to the security policy concept.” By visualizing code semantically, developers can understand impacts and dependencies that are not obvious from file structure alone.
* **Spectral Panel (Phase/Debug Dashboard):** This panel shows the dynamic analysis from the ALAN core. For example, it might display a real-time gauge of overall system consistency (if oscillators are largely synced or not), or a timeline graph of a particular concept’s phase drift over time as the code changed. In debugging mode, if the user runs the program (note: TORI IDE might integrate with a runtime to get execution traces), this panel could align runtime events with concept oscillations – essentially giving a new dimension to debug: not just variable values, but concept-level “health” signals. Concretely, a graph might show multiple sine-wave-like lines (each representing a high-level component’s phase) and highlight when one goes out of sync (pointing to a possible bug introduction time).
* **Multi-Agent Console:** TORI’s agents might produce textual logs or dialogues for the user. For instance, the Refactoring agent might have a note “I noticed functions X and Y are similar. Would you like me to refactor them?” The Test agent might output “Generated 2 new test cases for module Z.” We provide a console (or notifications area) where such messages are listed, potentially categorized by agent. This console also doubles as a command interface: the user could accept or dismiss suggestions directly from here, or ask follow-up (“Why do you suggest that?” which the agent can respond to via the Chat integration).
* **Integrated Chat Panel:** On one side of the IDE, there is a chat interface where the developer can converse with TORI Chat without leaving the IDE. This is essentially TORI Chat embedded – contextually aware of the open project. The developer could highlight code and ask, “TORI, how can I optimize this?” and the chat panel will respond, leveraging both its general training and specific knowledge from the code. Because it’s in-IDE, TORI Chat can do things like automatically link to relevant code or even execute small analyses (like “find all uses of this function” before answering).
* **Standard IDE Features Enhanced:** We include things like project navigation, file explorer, etc., but enhanced by TORI. For example, the file explorer might have badges on files indicating concept density or anomalies (imagine a file icon tinted red if that file contains concepts that are out-of-phase or marked in an agent warning). Search is semantic – beyond literal text search, you can search “concept: Login process” and TORI will find code related to the login process concept, even if the word “login” isn’t in those files (maybe it’s spread in different names).

The Thoughtboard UI is built with **local-first** principles: it does not require an internet connection to function fully (all heavy lifting is done by the ALAN core locally). This addresses privacy, and also ensures responsiveness. The UI communicates with the ALAN core via a local API or IPC mechanism (maybe an HTTP REST API on localhost, or direct memory calls if integrated).

We aim for the IDE to feel like a **modern, polished environment**. While initial versions may not have every feature of VS Code or IntelliJ, the critical ones are there, and TORI’s unique features are presented cleanly. We also brand the UI appropriately – using TORI’s theme (colors, logo) to give it a distinct identity as the “Cognitive IDE.”

*(Current UI Status:* The Thoughtboard interface exists in prototype. It currently runs as a Python/Electron app that opens a basic editor window and communicates with the core. Key panels like the concept explorer and phase dashboard are implemented in a rudimentary way (with dummy data for now). Over the development sprint, a lot of effort is allocated to hooking these up to real data and refining the UX. For example, as of now, the concept explorer can display a graph but not dynamically update; by mid-sprint it will update live as the code changes or as the user explores. Performance is another focus – ensuring that even with thousands of concept nodes, the UI remains responsive via techniques like virtualization or focusing on relevant subgraphs.)

**TORI Chat Interfaces:** TORI Chat can be accessed in two main ways: embedded in the IDE (as described) and as a **standalone application** for general conversational AI use.

* *Embedded in IDE:* This version of the interface is tailored to a developer’s needs. It might automatically inject context about the code when you ask a question (so you don’t have to type out “in function X, ...” if you have that function selected). It can also present answers with references into the code. For example, if TORI Chat explains an error, it might embed a snippet or a link that when clicked, navigates the IDE to the relevant code location. The chat UI inside IDE is minimalist (to not clutter coding space) but easily expandable.
* *Standalone Chat App:* For more general use (or for non-developers interacting with TORI’s knowledge), we provide TORI Chat as a separate desktop or web application. This has a chat window with history, persona selection, and possibly a way to manage knowledge packs. One could deploy this in an enterprise as an internal assistant: employees could query it about company policies, or troubleshoot issues by describing them in natural language. The standalone UI might allow logging in (if connecting to a central knowledge base), but in our local-first scenario, it could simply run locally with whatever knowledge packs are loaded. It will highlight that all data is local (to build trust for users worried about sensitive queries). The design is akin to popular chatbots, but with an emphasis that it *shows its work*. Perhaps an “inspect reasoning” button is available for power users to see the concept graph path or agent reasoning that led to an answer.

In both interfaces, **persona and multi-agent feedback** are important. TORI Chat can adopt different styles or roles (e.g., “explain like I’m a junior developer” vs “give me a strictly factual answer with references”). The UI might offer a toggle or prompt for that. Under the hood, this corresponds to activating different subsets of agents or applying certain response formating. Also, if multiple agents contribute to an answer (like a compliance check agent adds a caution), the UI could indicate that (“According to PolicyAgent: ...”).

**Visualization and Feedback Tools:** Aside from core features, we have tools for the user to visualize and correct TORI’s behavior. For example, an **Agent Manager** panel could list the agents, allow the user to toggle certain agents on/off or set their aggressiveness (maybe a slider for how proactive the Refactoring agent should be). This gives advanced users control and trust – they can turn off help they don’t want.

Another tool is the **Knowledge Browser**, which is essentially a read-only view into the LCN. This would list all concepts TORI knows about (perhaps grouped by type or source). A user could search their domain concepts here to see if TORI has captured them. This doubles as a debugging tool for concept integration – if TORI missed a key concept, the user might notice it here and know to feed TORI more info.

Finally, a **Settings/Privacy Panel** ensures the user can confirm that all operations are local (or connect only to specified resources), aligning with our privacy principle.

*(Current UI Status:* The standalone chat UI is at a very early stage – currently just a simple web page interface used for testing. We will use part of the sprint to build it out (likely leveraging web technologies for cross-platform ease). The integrated chat in IDE currently just opens a window to a dummy chatbot; hooking it to the real TORI Chat backend will happen once the core QA pipeline is solid. We anticipate rapid progress here once the rest is in place, since it’s largely an integration task with perhaps some front-end coding for polish.)

**Multi-Agent Orchestration**

One of TORI’s defining features is its **multi-agent system** – the idea that TORI’s intelligence is composed of multiple specialized agents working together under an orchestrator. This design mimics the human cognitive strategy of dividing problem-solving into sub-tasks (and even reflects Minsky’s Society of Mind theory, where many simple processes together yield intelligenc[en.wikipedia.org](https://en.wikipedia.org/wiki/Society_of_Mind#:~:text=In%20his%20book%20of%20the,2)】).

**Central Orchestrator:** At the center of the multi-agent layer is the orchestrator, which acts like a conductor for an orchestra of agents. The orchestrator’s responsibilities include: activating the appropriate agents for a given situation, mediating conflicts between agents, and integrating their outputs into a coherent result or user-facing action. It runs within the ALAN core and has access to the global state (concept graph, oscillator states, etc.), which it uses to inform decisions. For instance, when a code analysis event occurs (user saved a file), the orchestrator might simultaneously notify the Refactoring Agent, Documentation Agent, and Test Agent to do their analysis. If they produce outputs, the orchestrator collects them. If two outputs conflict (one agent says “remove this function” and another says “improve this function”), the orchestrator uses predefined rules or meta-reasoning to resolve it (maybe it decides improvement is the safer route and discards the removal suggestion, or flags the conflict for the user to decide).

The orchestrator can also sequence agents. Suppose answering a chat query might involve multiple steps: retrieving info, then reasoning, then simplifying the answer. Instead of one monolithic agent doing all, the orchestrator can call a KnowledgeRetrieval agent first, then pass the findings to a Reasoning agent, then perhaps to an AnswerFormatter agent. This pipeline is orchestrated such that each agent does what it’s best at. The user just sees the final answer, but behind the scenes it was a team effort.

**Specialized Agents:** We have several agents implemented or planned:

* *Refactoring Agent:* Focuses on code quality and maintainability. It scans the concept graph for code smells: duplicated logic, long functions, inconsistent naming, etc. It leverages the semantic info – e.g., if two concepts (functions) have very similar subgraphs (same calls, similar documentation), that’s a hint to refactor. It then suggests a refactoring (or even generates the refactored code via transformation rules). It can also respond to user commands like “refactor this file” by applying known refactoring patterns.
* *Documentation Agent:* Ensures that the knowledge in the code is reflected in documentation. When code changes, it updates or marks outdated any concept documentation nodes. It can generate summaries for functions by looking at related concept linkages (like pre/post conditions, usage examples from code, etc.). It might also assist in creating architecture docs by summarizing clusters of concepts (for example, produce a brief description of the “Payment subsystem” concept cluster).
* *Testing (QA) Agent:* (Not to be confused with the Q&A nature of chat – here QA means Quality Assurance.) It looks at the concept graph to find untested concepts or potential bugs. If a concept (like a function or a requirement) has no associated “test case” concept, it flags it. It can even attempt to generate test cases. For example, it might identify boundary conditions for a function by analyzing its preconditions concept, then propose a test for each boundary. It works closely with the core analysis; if, say, the Koopman analysis finds an unstable mode related to input size, the Test agent could suggest a stress test on that input range.
* *Security/Compliance Agent:* (Planned) Focuses on domain rules like “no sensitive data logged” or “function X must always be called before Y.” These could be programmed via ELFIN or policy files and this agent checks them continuously. It highlights any violations. For instance, if concept “PersonalData” flows into a function not marked as secure, it will warn.
* *Knowledge Retrieval Agent:* (Primarily for Chat) When a question is asked, this agent searches the LCN and external sources (if allowed) for relevant information. It might fetch definitions, or recall that “Oh, the user’s asking about Concept Z, which was referenced in Document Q.” It basically does the research part, so the reasoning agent can focus on logic.
* *Reasoning Agent:* (Primarily for Chat) This agent takes the assembled facts and tries to logically deduce or answer the question. It might chain together multiple facts (kind of like a theorem prover or rule engine). For code questions, it could perform static analysis steps to find an answer (like simulate execution in concept-space, etc.).
* *Persona/Stylist Agent:* This agent modulates how answers are delivered. For example, if the user selected an “Expert” persona vs a “Tutor” persona, the stylist agent will adjust wording and the level of detail. It doesn’t change the content (which comes from the reasoning agent), just the packaging. In multi-agent terms, it might even be considered a post-processing agent on the answer.
* *MPorter (Meta-Strategy) Agent:* This is the strategy agent mentioned earlier. It monitors high-level goals and progress. In the IDE, it might notice the developer has been ignoring a particular critical warning and might gently resurface it or reprioritize agent outputs (ensuring important things don’t get lost in noise). In Chat, it could break down complex user requests. If a user asks a vague multi-part question, MPorter agent might internally split it into parts and ask the knowledge agent to handle each, then aggregate results. It’s essentially thinking about *how to think* – planning the orchestration of other agents for complex tasks.
* *Attachment (Emotional) Agent:* This agent monitors the user’s emotional state (inferred from their interactions or message tone) and adjusts TORI’s responses accordingly. It introduces an empathic element. For example, if a user in chat says “I’m really stuck and this is frustrating me,” the Attachment agent will influence the response to be more supportive or apologetic (“I understand this can be frustrating. Let’s break it down together.”[en.wikipedia.org](https://en.wikipedia.org/wiki/Affective_computing#:~:text=Affective%20computing%20is%20the%20study,interpret%20the%20emotional%20state%20of)】. In the IDE, if a user seems to be thrashing (repeatedly changing code back and forth), it might suggest taking a step back or offer an explanation of the underlying issue to calm the confusion. This agent does not solve technical problems – it watches the *interaction* and ensures the AI’s behavior maintains a good rapport with the user.

**Conflict Resolution and Synergy:** With many agents, conflicts or redundant efforts can occur. The orchestrator uses a few strategies to handle this:

* *Priority rules:* Some agents have higher priority outputs (e.g., a security violation from the Compliance agent might override a suggestion from the Refactoring agent in terms of urgency).
* *Merging:* If two agents essentially suggest the same thing in different words, the orchestrator merges that into one suggestion to the user (giving credit to both, perhaps).
* *Competition:* In some cases, the orchestrator might let agents “debate.” For instance, one agent might propose solution A to a problem, another proposes B. The orchestrator could let both run through a simulation or argument and see which is more convincing (similar to how adversarial agents work in some AI frameworks). This is experimental, but TORI Chat’s multi-answer consideration could use this – generating answers via different reasoning paths and then choosing the best.
* *User input:* Ultimately, if agents disagree, TORI can surface the disagreement to the user: “There are two ways to resolve this issue: method X (suggested by DesignAgent) and method Y (suggested by PerfAgent). Would you like to consider one?” This turns a conflict into a choice for the human, which is often the right approach when design trade-offs are involved.

**Orchestration in Practice:** Suppose a developer hits “Analyze Project”. The orchestrator triggers: RefactoringAgent (checks code structure), TestAgent (looks for test coverage), ComplianceAgent (checks rules), DocumentationAgent (updates docs), and they all run in parallel. The orchestrator gathers their findings: maybe two refactor suggestions, one compliance warning, and documentation updates. It then prioritizes the compliance warning at top (marked in red in UI), lists refactor suggestions next (maybe in yellow), and notes doc updates silently (perhaps auto-applied). The developer sees a summary in the IDE’s agent console and can address them as needed.

In a chat scenario, say the user asks “How can I improve the performance of the data import process?” Orchestrator engages: KnowledgeAgent (find data import code and any known performance issues), RefactoringAgent (since it’s performance, maybe PerfAgent if we had one, to simulate usage or identify bottlenecks conceptually), ReasoningAgent (to formulate the improvement steps), then PersonaAgent (to format answer). The orchestrator sequences these, ensures the reasoning agent gets the findings from knowledge search (like “the data import reads line by line, which is slow”), and then the reasoning agent might conclude “use bulk operations or streaming”. The persona agent then crafts a response: “I notice the data import process reads data line-by-line. Combining those reads or using batch processing could significantly improve performance. For example, ...” The orchestrator sends that to the user. If the user asks “Why?”, the orchestrator might direct that to the ReasoningAgent or KnowledgeAgent to provide the conceptual reason (maybe citing that line-by-line I/O is conceptually linked to high overhead).

*(Current Status:* The multi-agent framework is in place with basic agents (Refactoring and Documentation running in the IDE context, Knowledge and a simple Q&A reasoning in chat context). Conflict resolution currently is rudimentary (essentially, we have a fixed priority order and we haven’t had complex conflicts in the prototype). Part of the sprint is dedicated to expanding these agents’ logic (as described in the 21-day plan) and refining the orchestrator policies. We’ll also be user-testing the agent outputs to adjust how verbose or aggressive each agent should be by default. The ability for a user to configure agents (turn off or tune) is planned in the UI, as mentioned.)

**Development Roadmap – 21-Day Sprint Plan (Starting May 12, 2025)**

To achieve a fully functional TORI IDE and TORI Chat by the end of the sprint, we have devised a **21-day development plan** divided into daily milestones. The plan assumes **Day 1 is Monday, May 12, 2025**. It is structured as three weeks of focused development, with each week allocating one day (Day 7, Day 14, Day 21) for rest, review, or buffer. Below is the plan for TORI IDE, followed by the plan for TORI Chat. Time-based references (Day 1, Day 2, etc.) are annotated with actual dates for clarity.

**Note:** The first week of the IDE sprint heavily emphasizes completing the partially implemented core features (oscillator sync, spectral analysis, ELFIN integration, orchestrator basics). These are foundational tasks that must be finished early to unblock dependent features. The chat sprint runs in parallel, often focusing on integration and interface aspects while core logic is firmed up in the IDE context.

**TORI IDE – 21-Day Sprint Plan (May 12 – June 1, 2025)**

**Sprint Goal:** Transform TORI IDE from a prototype into a polished v1.0 product, with all core cognitive features implemented, tested, and integrated into the Thoughtboard UI. By Day 21, the IDE should reliably provide intelligent assistance (refactor suggestions, conceptual warnings, etc.), and be ready for initial deployment to friendly users.

* **Day 1 (May 12, 2025)** – *Sprint Kickoff & Environment Setup*  
  **Goal:** Align team on objectives, finalize work assignments, and ensure the development environment is ready for rapid iteration.  
  **Tasks:** Kick off with a team meeting reviewing the Master Plan’s highlights (vision, architecture, critical components to finish). Set up version control, branch strategy, and CI pipelines. Ensure everyone can run the prototype (backend + UI) locally. Address any environment issues (e.g., if someone’s on Windows, make sure dependencies like spaCy, PyQt, etc., install properly). Clear the slate of trivial known bugs from prototypes so baseline is stable.  
  **Checkpoint:** By end of Day 1, the team is synchronized, the latest prototype code is merged to main, and each developer has a working instance of TORI IDE to build upon. Any blocking setup issues are resolved.
* **Day 2 (May 13, 2025)** – *Core Engine Implementation: Phase Synchronization*  
  **Goal:** Implement and verify the **Banksy phase-coupled oscillator synchronization** in ALAN core (replacing any placeholder logic for concept syncing).  
  **Tasks:** Develop the new synchronization algorithm based on the ψ-coupling design. Code the oscillator update loop that runs after each analysis tick: for each linked concept pair, adjust phases toward alignment (or intended phase offset). Incorporate global feedback: e.g., use a convergence factor from spectral analysis (if available) to modulate coupling strength (this ties in Day 3’s tasks). Write unit tests: create a mini concept graph in isolation and simulate oscillator steps, asserting that phases converge for connected nodes and remain independent for unconnected nodes. Remove the old dummy update code and integrate the new one into the main loop (ensuring it runs, say, whenever the concept graph changes).  
  **Checkpoint:** End of Day 2, oscillators in the core actually move and reflect influences. If concept A and B are linked, and A’s state changes, B’s phase will respond (rather than doing nothing as before). We likely log some debug info to see this in action. The code should be free of obvious stability issues (not diverging or oscillating wildly unless intended). The Kuramoto mention in documentation can be updated to note we now use the custom ψ-sync method.
* **Day 3 (May 14, 2025)** – *Core Engine Implementation: Spectral Analysis Pipeline*  
  **Goal:** Implement the **Koopman spectral analysis and Lyapunov stability checks** in the core to enable predictive reasoning.  
  **Tasks:** Build out the data pipeline for capturing state snapshots of the concept network (phases, active concept counts, etc.) over time or events. Implement a function to compute a spectral decomposition on recent state history – this could start simple: e.g., take the last N states, perform an SVD or eigen-decomposition, identify the dominant mode. Mark stable vs unstable trends (maybe define a Lyapunov function L and check L(state\_{t+1}) < L(state\_t) for stability). Integrate this so it runs periodically (not necessarily every code edit, maybe on a timer or when changes stabilize). Use results: e.g., if an unstable mode is detected, store an “alert” in core memory (the orchestrator can later translate that to a suggestion: “System architecture is oscillating – possible unresolved design decision.”). Also, feed spectral insights back into oscillator coupling if applicable (for instance, if one mode corresponds to two subsystems, perhaps temporarily decouple them by lowering coupling strength).  
  **Checkpoint:** End of Day 3, the core produces some spectral output. We should be able to see logs or debug info like “Mode1 eigenvalue = 0.95 (stable), Mode2 = 1.1 (growing unstable)”. If possible, a simple scenario where we *know* a pattern (like we oscillate a small graph input back and forth) confirms the spectral detection (it should spot the oscillation frequency). The code still might use a simplified analysis (e.g., linear trending) if full Koopman integration is complex, but the structure is in place and returning meaningful data.
* **Day 4 (May 15, 2025)** – *Integration of Core Components & Partial Testing*  
  **Goal:** Integrate the newly implemented core features (oscillator + spectral) with the rest of the system and run preliminary tests. Finish any loose ends in core (ELFIN parsing integration, etc.).  
  **Tasks:** Connect the ELFIN DSL parser into the build: ensure that if an .elfin file is present in a project, the integration layer processes it and populates concepts accordingly (this might have been partly done, make sure it’s fully working). Test the full core loop: e.g., load a sample project that has some known conceptual inconsistencies and see if the oscillator + spectral system picks them up (in logs or via some debug UI). Fine-tune parameters (coupling strengths, threshold for instability warnings) based on these tests. Address any core crashes or performance hiccups: e.g., if spectral calc is slow, maybe reduce frequency or data length for now. Essentially, stabilize the core. By now, the core is feature-complete, so begin writing documentation comments for it (to help others understand the complex logic).  
  **Checkpoint:** End of Day 4, the ALAN 2.x core is **fully operational** within the application. We have a stable build where the concept graph updates, oscillators sync, and spectral analysis runs without major issues. We likely produce some internal metrics (maybe a “core health” printout indicating system consistency) that will later be surfaced in UI. The team can proceed confident that this core foundation is ready to support UI and agent features.
* **Day 5 (May 16, 2025)** – *Thoughtboard UI: Concept Explorer & Core Data Wiring*  
  **Goal:** Hook up the Thoughtboard UI to the live data from the ALAN core (concept graph, agent outputs, etc.), focusing on the **Concept Graph Explorer** panel and inline annotations.  
  **Tasks:** Implement an API in the backend to serve concept graph data (e.g., a REST endpoint /concepts returning JSON of nodes/edges, or use an in-memory shared model if in Electron). In the front-end, replace the placeholder graph in the Concept Explorer with data from this API. Enable basic interactions: clicking a node in the graph highlights the corresponding code element in the editor (this requires mapping concept -> source location, which the backend can provide if the concept came from code). Conversely, when a user navigates to a different function in the editor, update the Concept Explorer to focus that function’s node (and its immediate relationships). Also, surface core analysis warnings: if the core has flagged an unstable mode or conflict concept, ensure the UI can highlight it (e.g., concept node might have a red outline, or the editor gutter might get an icon at relevant lines). Start displaying simple agent messages in an output panel (even if just logging them).  
  **Checkpoint:** By end of Day 5, the UI is truly “talking” to the backend: for instance, a developer can open the Concept Explorer and see an accurate graph of, say, the current file’s concepts and their links. It updates reasonably promptly after code changes or navigating to a new file. This is a big step as now TORI’s unique info is visible. Any glaring UI performance issues (like graph too slow on moderate number of nodes) are noted to optimize on Day 10. Inline annotations might still be few (since agents haven’t been fully enabled yet), but the pipeline for showing them (e.g., an underlined range corresponding to a concept with a warning) is in place.
* **Day 6 (May 17, 2025)** – *Multi-Agent System Activation (IDE context)*  
  **Goal:** Activate key **IDE agents** using the now-functional core, and integrate their outputs into the UI. This begins TORI’s intelligent behavior in the IDE.  
  **Tasks:** Enable the **Refactoring Agent** on code analysis events. Remove any placeholder logic; use real graph queries (e.g., find duplicates by concept similarity) to generate suggestions. For now, log these suggestions or list them in the agent console. Do the same for the **Documentation Agent** – e.g., after each analysis, have it update documentation concept nodes and perhaps suggest adding docstrings where missing. Ensure the orchestrator is coordinating: wire up a simple orchestrator that on each “analysis cycle” calls the active agents and collates their suggestions. Then implement UI rendering of these suggestions: e.g., a pane or popup listing “Suggestions: Function foo duplicates bar (RefactorAgent) – [Accept]/[Ignore]”. If accept is clicked, we won’t fully automate refactoring yet (unless trivial), but we could at least merge concept or mark it resolved. Also implement a way to dismiss suggestions. Start small: maybe just printing them to console initially, then improving UI as time allows. The aim is to see at least one concrete agent output by the user.  
  **Checkpoint:** End of Day 6, TORI IDE is **making intelligent suggestions**. For example, on our test project, the refactoring agent might flag two similar functions. We can see this either in the console or in an overlay UI. Multi-agent orchestration is rudimentary (likely sequential or priority-based), but working – meaning no crashes and suggestions seem relevant. This effectively is the first visible “AI assist” moment in the IDE.
* **Day 7 (May 18, 2025)** – *Review & Buffer (Core/Backend)*  
  **Goal:** Review progress of Week 1, fix bugs, and adjust plans if needed. Use as buffer for any unfinished tasks.  
  **Tasks:** Conduct a team demo of what we have so far (core working, UI showing graph, maybe one suggestion coming through). Collect any obvious issues: e.g., “the concept graph for large file is too cluttered – consider auto-grouping”, or “oscillator sync seems slow for 1000 concepts – maybe optimize coupling calc”. Spend time fixing critical bugs discovered in integration (like UI race conditions, or an agent suggestion misidentifying code). Polish some behind-the-scenes aspects: cleanup the code, add comments, perhaps improve logging structure. Update documentation (internal developer docs) to reflect how new core/agents work. Given this is a planned lighter day, ensure the team doesn’t burn out – possibly a half day of work and an afternoon of rest or learning.  
  **Checkpoint:** By end of Day 7, the team has a clear picture of what’s working well and what needs focus next. Any core or integration issues that could impede new feature work are resolved. The plan for Week 2 might be adjusted slightly based on Week 1 outcomes (for instance, if core is slower than anticipated, allocate some optimization time). The project is on track, or if behind, we have identified where to catch up using buffer days.
* **Day 8 (May 19, 2025)** – *Thoughtboard UI Polish & Advanced Features*  
  **Goal:** Improve the UI/UX of the Thoughtboard with better visualizations and controls, focusing on the **Spectral Dashboard** and user controls for agents.  
  **Tasks:** Now that the core provides spectral data, design and implement the **Spectral Debug Panel**. This could be a small graph in the corner showing overall system “coherence” over time (e.g., a line trending upward as things stabilize). Or perhaps list current active modes: e.g., “Mode: ‘Memory Leak potential’ – unstable”. Tie this with UI cues: e.g., if an unstable mode is detected, highlight the related concepts (maybe flashing icon on related modules). Also, implement **user controls for multi-agent**: a simple settings dialog where agents can be toggled on/off or set to auto-apply trivial fixes. Ensure these preferences persist in the session. Enhance the Concept Explorer by adding filtering options (checkboxes to show/hide certain concept types, like maybe turn off showing local variables to declutter). Add the ability to double-click a concept node and keep the explorer centered there (for deep dives). Finally, refine the **suggestion UI**: use icons (a lightbulb for refactor, a book for documentation suggestion, a shield for compliance, etc.) to label each suggestion’s source agent so the user knows why it’s suggested.  
  **Checkpoint:** End of Day 8, the Thoughtboard interface should feel more mature. The spectral panel is live, though it might be fairly simple in this version, it at least gives some feedback (like “System Stable” or “Instability detected in X”). Users (testers) can now control agent involvement somewhat. The concept explorer is more usable with filters. We are essentially moving from raw functionality to a cleaner user experience.
* **Day 9 (May 20, 2025)** – *Agent Enhancements (IDE)*  
  **Goal:** Increase the sophistication of the IDE agents and their coordination. Make their outputs more accurate and useful.  
  **Tasks:** Revisit the **Refactoring Agent** logic with fresh eyes after seeing initial outputs. Improve its detection algorithms: maybe implement a simple AST similarity metric for functions to catch duplicates that textual similarity might miss. Also, add new refactor types: e.g., detect when a class has too many responsibilities (if concept graph shows it’s linked to very disparate concepts, maybe suggest splitting class). Introduce a **Performance Agent** (if not separate, maybe part of refactoring agent): using concept metrics (like if a loop concept is heavily used or if a function’s complexity is high), suggest performance improvements (this can dovetail with insights from spectral if an unstable mode is related to performance). Enhance the **Documentation Agent** to auto-generate missing doc comments using available info (perhaps call a templating system or even an LLM for natural language, constrained by known facts). For multi-agent orchestration, implement a basic conflict resolution: e.g., if the Documentation agent wants to add a docstring to function X that the Refactoring agent wants to remove entirely, orchestrator should prioritize the refactor suggestion and hold off on documentation for now. Additionally, integrate the **Compliance/Security Agent** if any rules are readily available (even a dummy example rule via ELFIN). Test these in combination on sample scenarios.  
  **Checkpoint:** End of Day 9, the IDE’s intelligence feels sharper. We should see, for example, that it not only points out duplicate functions but maybe also says “Function Y is very large (50 lines) – consider refactoring for single responsibility.” Documentation agent might start filling in blanks which we can see in a diff view. We likely now have multiple suggestions possibly affecting the same code, so how orchestrator orders them is set (and visible to user with the labeling). The system’s advice breadth has increased (touching design, performance, etc. in addition to trivial duplicates).
* **Day 10 (May 21, 2025)** – *Performance Optimization & Scalability*  
  **Goal:** Ensure TORI IDE runs efficiently on larger projects and optimize any slow components (core or UI).  
  **Tasks:** Conduct profiling sessions. Load a codebase with, say, a few thousand concept nodes (maybe the TORI codebase itself) and measure UI render time for concept explorer, time taken for analysis loop, memory usage, etc. Optimize **core algorithms**: e.g., if oscillator update is O(n^2) on number of concepts, consider optimizations (since many concept pairs might have zero coupling, we could optimize by tracking edges). Possibly integrate a graph library or C extension for heavy math. Optimize **UI rendering**: use canvas or WebGL for concept graph if DOM approach lags, implement incremental graph updates instead of full redraws. Also optimize multi-agent runs: maybe stagger some agent work to not all hit at once if that causes CPU spikes (or run in parallel if CPU cores free). Check memory: ensure concept nodes are being freed if removed, etc. The goal is that medium projects (hundreds of files) do not bog down the experience. Also, consider toggling off some verbose debug logs now to reduce overhead.  
  **Checkpoint:** End of Day 10, performance is improved. Ideally, initial analysis of a moderately sized project should complete in a matter of seconds, and interactive operations (like editing a file) trigger responses that feel near-instant. If some heavy operations remain (maybe full project re-analysis), they are either optimized or demoted to a background thread so the UI stays responsive. We should document current performance metrics and decide if any features need to be cut or scaled back for v1.0 to maintain responsiveness (for instance, maybe not analyzing *every* file in huge projects, or adding a preference for scope of analysis).
* **Day 11 (May 22, 2025)** – *TORI Chat Integration (IDE & Standalone)*  
  **Goal:** Integrate the TORI Chat back-end with the IDE (embedded chat panel), and bring the standalone TORI Chat application to a functional state using the shared core.  
  **Tasks:** Connect the embedded chat panel in Thoughtboard to the ALAN core’s chat endpoint. Test end-to-end: the user types a question in the IDE, it goes to the chat pipeline (Knowledge agent finds relevant code concepts, reasoning agent forms answer), and a reply appears referencing code. Work on formatting: code excerpts in answers should be displayed nicely (monospaced, maybe syntax highlighted if feasible). Ensure the chat has access to the project’s LCN; this might involve saving a copy of the LCN or querying the live one. For the **standalone Chat app**, implement any missing pieces: perhaps a simple UI (we can reuse some web components from the embedded panel). Focus on the ability to load concept packs: create a UI control to load a file (like a finance ontology JSON), and when loaded, verify chat answers incorporate that knowledge. If the standalone is meant for non-coders, test it on general Q&A and refine response style to be more explanatory since no code context there. Tie up persona selection: add a dropdown or preset buttons in chat UI (e.g., “Detailed mode” vs “Brief mode”) which the Persona agent will interpret to modify answers.  
  **Checkpoint:** End of Day 11, TORI Chat works in both contexts. In the IDE, you can ask something like “What does this function do?” and get a meaningful answer that TORI derived from understanding the code (with maybe references or an offer to navigate to related code). Standalone, you can ask something from a loaded knowledge pack, like “What is KYC?” and get an answer drawn from the ontology loaded (with the answer clearly reflecting that domain knowledge). There may still be room to improve natural language fluency, but correctness and integration are the focus.
* **Day 12 (May 23, 2025)** – *Enterprise Readiness Features*  
  **Goal:** Add features important for enterprise deployments: **audit logging, privacy verification, configuration**.  
  **Tasks:** Implement an **Audit Log** system in the backend that records all suggestions made by TORI and actions taken (by the user or automatically). For example, log lines like “2025-05-23T10:05: RefactoringAgent suggested combining functions A and B [file names], user accepted.” or “ChatAgent provided answer about X using source Y.” Ensure these logs are stored locally in a secure way. Add a **toggle** in settings for “Send anonymous usage stats” (for our telemetry if any) defaulted off, and a clear indicator that TORI doesn’t send code to cloud (maybe a note on first launch to reassure privacy). Build a **configuration UI** to adjust performance-related settings or agent aggressiveness. Possibly integrate a **license management stub** if needed (e.g., a place to input a license key or check subscription, though for an internal plan this might not be needed yet). If time, implement a **safe mode** for Chat where it will not answer certain categories of questions (maybe irrelevant for on-prem, but if needed for liability, e.g., disallow advice on topics outside provided knowledge packs).  
  **Checkpoint:** End of Day 12, TORI is more enterprise-friendly. There is an audit log being generated that an admin could review for compliance (which also doubles as a debugging aid for us). The user can configure the tool’s behavior in meaningful ways through a preferences UI. We double-check that no data is going out that shouldn’t (run tests with firewalls, etc., to see that TORI doesn’t try to reach external servers unless explicitly asked for an update check perhaps). At this point, we are feature-complete for core functionality and we switch into fix/polish/test mode for the final stretch.
* **Day 13 (May 24, 2025)** – *User Testing and UX Refinement*  
  **Goal:** Conduct in-depth internal testing of TORI IDE & Chat with fresh eyes, and refine the UX based on feedback.  
  **Tasks:** Bring in a few colleagues (not on the dev team) for a user testing session. Have them use TORI IDE on a small project and TORI Chat on some questions. Observe where they get confused or where UI is unintuitive. Collect feedback: e.g., maybe they didn’t notice the chat panel icon, or found the concept graph overwhelming. Triage issues into “must-fix” vs “could improve later”. Focus on low-hanging fruit improvements from this feedback: adjusting iconography, adding a one-time tutorial pop-up (like highlighting “This is the Concept Explorer, which shows relationships.”), improving message wording (e.g., suggestions might say “(Refactor)…” which a user might not get; maybe say “TORI Suggestion:” instead). Fix any minor bugs discovered (like UI alignment issues, or an agent firing at the wrong time). Ensure consistency: check all button labels, terminology (“concept” vs “knowledge” – decide on user-facing terms and stick to one). If testers encountered any serious functional bugs (crashes, features not working as expected), address those immediately.  
  **Checkpoint:** End of Day 13, TORI should feel more **user-friendly**. Perhaps we added a quick start tooltip walkthrough that appears on first launch. We likely rename or clarify a few UI elements based on feedback. The application should handle common user behaviors more gracefully now (for example, if a user deletes a whole file, does the concept graph update without errors? Now it should).
* **Day 14 (May 25, 2025)** – *Comprehensive Testing & Bug Fixing (Bug Bash)*  
  **Goal:** Rigorously test all features (IDE and Chat) for bugs or edge cases and fix them.  
  **Tasks:** Do a “bug bash” where the dev team tries to break TORI in every way they can think of. This includes: extremely weird code constructs (does the parser choke?), concurrent actions (edit code while asking a chat question – does anything race or deadlock?), large inputs (open a huge file – does UI lag, and can we mitigate maybe by not highlighting concept for every single token?), error handling (shut down core unexpectedly, does UI show an error or freeze?). Test installation paths (fresh machine setup from our installer, testing user without Python installed if our bundle should cover it, etc.). List out all discovered bugs. Prioritize by severity: crashes and data loss/ corruption issues are highest, then major functional bugs, then minor cosmetic ones. **Fix aggressively** all the severe ones. For moderate ones, fix as many as time permits, especially if low risk. Some minor ones might be deferred if they’re not critical (we keep note to possibly address in a patch). Also run our automated test suite (if we have one by now, which we should for core logic) and ensure all pass. Perhaps incorporate last-minute test cases that cover new ground we learned from user tests. If any agent is misbehaving (like false positives or too many suggestions making noise), fine-tune thresholds or turn it down for release.  
  **Checkpoint:** End of Day 14, we ideally have zero known crash bugs and zero known major malfunctions. The product should be stable in heavy use. There may still be a list of minor quirks or potential improvements, but nothing that would embarrass or seriously frustrate an end user in a v1.0 trial. We are nearly code-complete; as we go into final days, focus shifts to documentation and final polish.
* **Day 15 (May 26, 2025)** – *Documentation & Final Feature Freeze*  
  **Goal:** Prepare comprehensive documentation (user guide, developer notes for internal use) and declare a feature freeze (no new features, only necessary fixes).  
  **Tasks:** Write the **User Guide** for TORI IDE and Chat. This includes: installation instructions, quickstart tutorial (covering basic workflow, e.g., “open your project, TORI will analyze and show suggestions; you can chat with TORI for help; here’s how to interpret the concept graph…”), explanation of each major UI component (maybe with screenshots), and an FAQ for common issues. Also document the **ELFIN DSL** usage (with a section describing how to write an ELFIN file and what TORI does with it). Write a section on **Troubleshooting** (like if suggestions aren’t appearing, check that agents are enabled; if performance is slow, try toggling X, etc.). Ensure the tone is accessible to the target users (developers, and possibly technical leads). Simultaneously, update the internal **Developer Documentation** – especially details of the ALAN core, so future maintainers (or open-source contributors, if it goes that way) can understand the complex parts (phase sync, spectral). This likely means adding comments in code, and maybe a short design doc summary to accompany the code. As feature freeze is now, ensure any pending merge requests for minor fixes are in, and then cut a release candidate build for final testing.  
  **Checkpoint:** End of Day 15, all documentation for release is ready or in final draft form. We have a Release Candidate (RC1) build that’s feature-frozen. The team (and perhaps some test users) will rely on the user guide to use the product – a good test if the guide is clear. We’re confident that if someone new picks up TORI with this documentation, they can get started and understand the basics.
* **Day 16 (May 27, 2025)** – *Final Validation & Beta Release*  
  **Goal:** Validate the RC build against real-world use cases one last time, and if all is good, designate it as the beta/release version.  
  **Tasks:** Do a final full run-through of the user experience: from installation, to opening a project, to acting on suggestions, to using chat, to checking logs and settings. Ensure the audit log is indeed capturing events. Try it on a couple of different projects (maybe one open-source repo, one internal project) to see how it behaves with code we didn’t specifically tailor for. If any critical issues appear, fix them (only critical fixes at this stage). Once satisfied, prepare the build artifacts for release: e.g., compile the Electron app to an installer for Windows/Linux/macOS as needed. Perform hash checks, virus scans, etc., as part of release prep. Label the build version (v1.0.0-beta perhaps). If there’s time, have a brief “beta release rehearsal” – writing the announcement or notes summarizing what’s in this version, known limitations (for transparency).  
  **Checkpoint:** End of Day 16, we have what we believe is the release build. It’s been tested in scenarios beyond our unit tests (some integration tests with sample user behaviors). We have not found any showstopper issues; any that were found are fixed or documented with workarounds if minor. The product is ready to deliver to the initial set of users (which could be internal stakeholders or a closed beta group).
* \*\*Day
* **Day 17 (May 28, 2025)** – *Release Preparation & Packaging*  
  **Goal:** Finalize all packaging and distribution aspects for the upcoming release.  
  **Tasks:** Prepare the installer or distribution packages for TORI IDE (Thoughtboard) on all target platforms (e.g., create a Windows installer, a Mac app bundle, etc.). Verify that each package includes all necessary dependencies (the AI models, knowledge packs, etc.) and that installation is smooth. Double-check that the audit logs and configuration files go to appropriate paths (e.g., user’s AppData or ~/.tori). Create a **Release Notes** document highlighting new features, known issues, and usage tips. Coordinate with any internal stakeholders about the release schedule.  
  **Checkpoint:** By end of Day 17, installable packages for TORI IDE and TORI Chat are ready. We have tested installing and uninstalling them on fresh machines. The release notes and any internal briefing materials are prepared. We’re essentially ready to roll out the software pending final sign-off.
* **Day 18 (May 29, 2025)** – *Internal Launch & Training*  
  **Goal:** Conduct an internal launch of TORI with a controlled group of users and provide training.  
  **Tasks:** Host an internal presentation or demo for stakeholders and potential pilot users. Walk through the vision, demonstrate the product live (showing how Thoughtboard identifies a bug and how Chat can explain a piece of code). Distribute the user guide and have a Q&A session. Set up a feedback channel (like an internal forum or email) for pilot users to report issues or suggestions. This day also serves as a buffer in case any last-minute critical bug appears during the demo – we can address it immediately.  
  **Checkpoint:** End of Day 18, the pilot users have the software in hand and are excited about its capabilities. They know how to use it at a basic level thanks to the training. We gather their initial reactions (which seem positive, say, enthusiasm about seeing conceptual links in their code). The team is on standby to support them as they start using TORI in daily work.
* **Day 19 (May 30, 2025)** – *Feedback Analysis & Minor Fixes*  
  **Goal:** Collect and analyze initial feedback from pilot users, and address any minor issues that surface.  
  **Tasks:** Monitor the feedback channels. If a pilot user reports a bug (“the concept graph viewer crashes on my project”), attempt to reproduce and fix it promptly. If any usability issues are noted (e.g., “I didn’t realize I had to click the lightbulb for suggestions”), consider quick tweaks or clarifications to the UI or documentation. Since we are technically post-release internally, limit changes to low-risk, high-value fixes (no new features). Possibly issue a quick patch update to pilot users if needed. Also, log enhancement ideas that are out of scope now but valuable for future versions (e.g., more agents for other domains).  
  **Checkpoint:** End of Day 19, any immediate post-launch fires are put out. Pilot users are not blocked by any major issues. The initial feedback is incorporated into our planning backlog for future improvements, but the current version remains stable.
* **Day 20 (May 31, 2025)** – *Go/No-Go Meeting & Launch Prep*  
  **Goal:** Decide on proceeding to broader release and prepare any external launch materials (if applicable).  
  **Tasks:** Hold a meeting with stakeholders to review pilot phase results. If the software has proven stable and valuable, get approval to move to the next stage (wider enterprise deployment or perhaps an external beta). If any serious issues arose, decide if they must be fixed before wider release or if documentation/training can mitigate them. Assuming a “Go” decision, prepare the external-facing materials: a polished announcement highlighting TORI’s strengths (e.g., a blog post or press release draft emphasizing “first cognitive OS IDE with explainable AI” etc.), and any marketing collateral like screenshots or short demo videos. Ensure legal compliance for release (open-source licenses in order, disclaimers included).  
  **Checkpoint:** End of Day 20, we have a green light to launch TORI to its intended wider audience. All necessary materials and logistical considerations for the launch day are ready. We’re essentially counting down to introducing TORI beyond the pilot group.
* **Day 21 (June 1, 2025)** – *Launch & Retrospective*  
  **Goal:** Officially launch TORI IDE and TORI Chat v1.0 and conduct a team retrospective on the sprint.  
  **Tasks:** Flip the switch on distribution – e.g., publish download links internally or externally, send out the announcement communications. The team will be actively monitoring for any issues as new users onboard (ready to provide support). Also, hold a **sprint retrospective** meeting: discuss what went well during the 21-day execution, what could be improved in our process, and gather lessons learned for the next development cycle. Acknowledge the team’s accomplishment (perhaps a small celebration for delivering this ambitious project on schedule!).  
  **Checkpoint:** By the end of Day 21, TORI IDE and ALAN 2.x (with TORI Chat) are officially released to their initial audience. The development sprint concludes with a functioning, innovative product in users’ hands. The team has documented insights to guide future sprints. We have successfully turned the TORI vision into a reality, setting the stage for further evolution.

**SWOT Analysis**

After completing the sprint and preparing for launch, we revisit TORI’s **Strengths, Weaknesses, Opportunities, and Threats** in the current context (mid-2025):

* **Strengths:** TORI’s core strength is its fundamentally different approach to AI assistance. Instead of relying solely on black-box predictions, TORI builds a *conceptual model* of the code and conversation, enabling **deep semantic understanding**. This yields more insightful and context-aware assistance than pattern-matching tools. Moreover, TORI’s design is **transparent and explainable** – every suggestion can be traced to a reasoning path in the concept graph, addressing a key demand in industry for explainable AI solutio[ibm.com](https://www.ibm.com/think/topics/black-box-ai#:~:text=White%20box%20AI%2C%20also%20called,Users)0】. The system’s **phase-synchronized reasoning** provides deterministic, reproducible results, which is a boon for enterprise users who need consistent behavior. Another major strength is TORI’s **local-first architecture**: all processing can be done on-premises, meaning companies can use TORI without exposing proprietary code or data externally. This privacy-preserving design is a differentiator, as it sidesteps cloud compliance issues and drastically reduces security concerns. Additionally, TORI’s multi-agent modularity makes it *extensible* – new capabilities can be added as separate agents without overhauling the core. Finally, TORI effectively combines development tooling with AI assistance; it’s a unified platform, which means users don’t juggle separate tools for coding, debugging, and asking questions – boosting productivity by keeping the workflow in one place.
* **Weaknesses:** TORI is an early-stage product and faces several limitations. The most immediate weakness is the **unproven nature** of its approach – while we’ve tested it on pilot projects, it lacks extensive real-world validation across diverse codebases. Its novel algorithms (oscillator networks, spectral analysis) may have edge cases we haven’t encountered; in contrast, traditional tools are very battle-tested. In terms of product maturity, TORI IDE currently lacks many of the ecosystem conveniences that established IDEs have (e.g., a rich plugin ecosystem, ultra-refined UI elements). We support only a handful of programming languages robustly (Python and ELFIN fully, others like C++ in a basic way), so users working in other languages might find TORI less helpful – this **limited language support** could narrow our initial user base and give an impression of incompletenefile-bnjjcfgff6iusnbzdwqayo2】. Performance, while much improved, could become an issue on extremely large projects – our advanced analysis might be slower than simpler tools for codebases with tens of thousands of files (something we will need to optimize as we scale). Another weakness is **user adoption risk**: developers have ingrained workflows, and some might be resistant to adopting a new IDE, especially one that introduces a radically different paradigm of AI involvement. Convincing them to trust TORI’s suggestions (and not see it as overly intrusive or gimmicky) will require education and gradual proof of value. Lastly, as a small team/new entrant, we have limited resources for support – any bugs or shortcomings early on could sour user perception if not addressed rapidly, so we must be very responsive, which can be challenging.
* **Opportunities:** The environment in 2025 is ripe for TORI’s unique value proposition. **Enterprise demand for AI** that is both powerful and private is high – many companies hesitate to use cloud-based AI coding tools due to intellectual property concerns. TORI can fill that gap by offering a **secure, on-prem AI assistant** that integrates with their internal knowledge (via concept packs) and doesn’t leak data. This is an opportunity to become the AI solution of choice for industries like finance, healthcare, or defense, where data control is non-negotiable. There is also a broader industry trend toward **augmented developer tooling**; TORI’s emphasis on explainability and determinism could position it as the *trusted* alternative to tools like GitHub Copilot (which sometimes outputs code without rationale). We can capitalize on growing awareness of AI’s “Clever Hans” problems (when models give the right answer for the wrong reasons) by showing how TORI avoids that and provides proof for its answe[ibm.com](https://www.ibm.com/think/topics/black-box-ai#:~:text=Reduced%20trust%20in%20model%20outputs)8】. Another opportunity is to extend TORI’s concept-based approach beyond code – for example, into data science notebooks, or integration with design tools – essentially, TORI could evolve into a general “cognitive assistant” platform. The multi-agent architecture means we could relatively easily add specialization for other domains (documentation writing, requirement analysis, etc.), expanding our market reach. Additionally, there is potential for **community and open-source contributions**: by sharing some of TORI’s DSL (ELFIN) or concept libraries, we could build a community that contributes domain packs (imagine domain experts encoding best practices in ELFIN for others to use). Lastly, if TORI demonstrates success, there is an opportunity for partnerships with larger IDE vendors or cloud providers – either integrating TORI’s technology into their offerings or collaborating to offer a hybrid solution (with TORI providing on-prem cognition and a partner providing cloud-scale resources if needed).
* **Threats:** The software development tooling market is highly competitive and is seeing rapid innovation in AI. A key threat is that **incumbent IDE vendors and big tech companies** will quickly advance their own AI-assisted tools. For example, Microsoft’s GitHub Copilot and Visual Studio integration is continuously improving, and JetBrains (IntelliJ) is introducing AI features – these players have enormous user bases and resources. If they solve the explainability or on-premise issues before TORI gains traction, TORI’s unique selling points could be diminished. We already see early moves in this direction, with competitors adding features to inspect and constrain AI suggestions. Another threat is **user skepticism or resistance** to TORI’s novel concepts. Developers might trust traditional linters or simpler AI assistants more because TORI’s internal workings (oscillator phases, spectral modes) are new and might seem arcane. We have to prove that these concepts genuinely translate to better results; otherwise users may dismiss TORI as over-engineered. There’s also the risk of **technical adoption barriers**: some IT departments might be slow to approve a new tool that has so many components (they might worry “Is it stable? Is it secure?” – thorough testing and certification may be demanded). Additionally, as TORI handles critical knowledge, any mistake (like an incorrect refactoring suggestion that introduces a bug) could undermine confidence – while our aim is to minimize such errors via careful design, no system is perfect. Larger competitors might also leverage their ecosystems to lock users in (for instance, if a competitor’s AI integrates seamlessly with their cloud devops pipeline, that convenience might outweigh TORI’s benefits for some teams). Finally, from a business perspective, if TORI gains attention, others may attempt to emulate our conceptual approach (we must continue innovating and possibly secure intellectual property, though patents in this space are tricky). In summary, the threats are the fast-moving competitive landscape and the need to win hearts and minds in a conservative segment of users.

**Conclusion:** Over the 21-day master plan, we have brought the TORI vision to life – creating a **cognitive operating system** that turns development and chat interfaces into truly intelligent, collaborative experiences. The resulting TORI IDE and TORI Chat (powered by the ALAN 2.x engine) demonstrate the feasibility and value of concept-oriented AI: they can understand code at a deeper level, explain their reasoning, and adapt to user needs, all while keeping the user in control. Initial testing indicates that TORI can catch subtle issues and provide insights that traditional tools miss, validating our paradigm shift towards *“concepts over tokens, memory over stateless prediction.”*

As we launch TORI to a broader audience, we will closely support our users and incorporate their feedback. Our SWOT analysis shows we are entering the market with a strong, differentiated solution, though we remain mindful of the challenges ahead. By maintaining our focus on technical rigor (grounded in proven science like Koopman theory and Lyapunov stability) and user-centric design (transparency, privacy, integration into workflows), TORI is well-positioned to carve out a niche of loyal users who need trustworthy AI assistance.

In the coming months, completing any partially implemented components and broadening language support will be priorities before resuming rapid feature development. With the foundation built in this master plan, those next steps will be on solid ground. TORI aims to herald a new era of developer tools – one where our IDE is not just a static environment, but a **cognitive partner** that evolves with us, ultimately making software development and knowledge work more intuitive, reliable, and powerful than ever before.

**Comprehensive Master Plan and Strategic Analysis – TORI IDE & ALAN 2.x Cognitive OS**

**Executive Summary**

**TORI** is a next-generation *cognitive operating system* comprising two flagship products: **TORI IDE** (an AI-augmented Integrated Development Environment) and **TORI Chat** (an intelligent chat assistant, with an enterprise-grade variant). This master guide serves as a comprehensive report, technical specification, development roadmap, and marketing playbook for the TORI suite. It encapsulates the visionary paradigm behind TORI, the current architecture and module breakdown, a 21-day sprint plan for rapid development, SWOT analyses, and go-to-market strategies for each product.

* **Vision:** TORI embodies a paradigm shift from traditional AI tools. Rather than relying on rote token prediction, TORI’s core is grounded in **concepts over tokens** and **memory over stateless prediction**[medium.com](https://medium.com/syncedreview/from-token-to-conceptual-the-rise-of-metas-large-concept-models-in-multilingual-ai-b32acbfeb792#:~:text=generalization%20across%20languages%2C%20outperforming%20existing,LLMs%20of%20comparable%20size). The vision is an “AI-native” development experience in which the system truly understands code and conversations at a conceptual level, enabling unprecedented levels of assistance and insight. Developers move from using tools to collaborating with a *cognitive partner*.
* **Architecture:** At TORI’s heart is the **ALAN 2.x cognitive engine**, a *spectral-phase reasoning* core that maintains a rich **Large Concept Network (LCN)** of knowledge. This core powers both the IDE and Chat, providing deterministic, auditable reasoning steps instead of opaque neural guesswork[ibm.com](https://www.ibm.com/think/topics/black-box-ai#:~:text=White%20box%20AI%2C%20also%20called,Users). Surrounding the core are modular subsystems for language processing, user interaction, and multi-agent orchestration. In the IDE, this means deeply understanding code structure and evolution; in Chat, it means maintaining contextual, persona-driven dialogue. Both products share a **local-first, privacy-conscious design** that can operate fully on-premises without cloud dependence, keeping user data and code private.
* **Current Status:** The TORI codebase has been through early prototyping phases. Key modules for concept parsing, memory graph management, and a basic front-end exist, though some components remain mock or placeholder implementations pending integration of the full ALAN engine. For example, a document ingestion pipeline can index PDFs into the concept network, but the **phase oscillator network** underpinning dynamic reasoning is only *partially implemented*. The current TORI IDE UI is functional as a prototype “**Thoughtboard**” dynamic code canvas, albeit without full polish or all expected IDE features. TORI Chat’s core reasoning back-end is in place, but its user-facing chat interface and multi-agent persona system are at a rudimentary stage. These gaps are identified in the module-by-module analysis, and the plan is to replace placeholder code with production-ready logic as a prerequisite to the development sprints.
* **Roadmap:** A detailed **21-day sprint plan** for each product outlines the path from the current state to a viable v1.0 release (with Day 1 starting May 12, 2025). Each sprint plan is organized into daily milestones (with one rest/review day built in per week). The TORI IDE plan focuses on integrating the ALAN core with the IDE front-end (Thoughtboard UI), implementing conceptual debugging tools, and eliminating any “mock” data paths. The TORI Chat plan emphasizes developing a robust conversational interface, integrating domain-specific “concept packs,” and adding enterprise features like audit logs and on-prem deployment support. Both plans culminate in a synchronized launch, ensuring the IDE and Chat can leverage each other (e.g. the IDE generating chat-ready knowledge graphs, and the Chat agent assisting within the IDE). **Before these sprints formally begin, however, the partially implemented core components (oscillator sync, spectral engine, ELFIN parser, orchestrator) must be completed to provide a solid foundation for rapid feature development.**
* **SWOT Highlights:** TORI’s **Strengths** lie in its unprecedented approach: a phase-synchronized reasoning core that yields deterministic, explainable insights (a “glass-box” AI approach)[ibm.com](https://www.ibm.com/think/topics/black-box-ai#:~:text=White%20box%20AI%2C%20also%20called,Users), and a local-first architecture that sidesteps the compliance and cost issues of cloud-only AI solutions. **Weaknesses** relate to typical early-stage challenges: limited UI polish compared to incumbent tools, a narrow language support (only Python and the custom ELFIN DSL are fully supported at launch), and the need for real-world benchmarking to validate the novel reasoning approach. **Opportunities** include a growing demand for AI tools that can be deployed on-prem for privacy, and integration into enterprise workflows that require transparency and auditability. **Threats** include rapid moves by industry giants to add similar AI capabilities to established platforms, and skepticism toward unproven AI paradigms – challenges the plan addresses via an aggressive development and validation strategy.

**Vision and Guiding Principles**

**TORI’s vision** is to create a *cognitive operating system* that transforms our interaction with software development and AI. This vision is grounded in two key ideas: **conceptual understanding over surface-level output**, and **persistent learning over stateless interaction**. Traditional AI coding tools operate on the level of syntax and tokens, predicting text without true understanding. TORI takes a different path – it builds and manipulates rich *conceptual models*. In the IDE, this means understanding the abstract architecture and intent behind code; in Chat, it means grasping the semantics and context of the conversation, not just stringing sentences together. By elevating to the conceptual level, TORI can provide insights and assistance that feel genuinely intelligent and context-aware, rather than generative but shallow. *“You’ve articulated not just a technical blueprint, but a paradigm shift in how we think about code, tools, and developer cognition,”* as one early reviewer noted.

Furthermore, instead of treating each user query or coding session as an isolated event, TORI maintains a **persistent memory** of interactions and knowledge. This is not a simple transcript log; it’s a structured memory in the form of the LCN (Large Concept Network), phase-state information, and learned patterns. TORI remembers *why* a piece of code was written or *how* a conclusion in chat was reached, enabling it to explain its reasoning and build on past knowledge. This contrasts with stateless large language model tools that have no inherent memory of past interactions. TORI’s approach yields an AI that can learn and adapt over time – accumulating **wisdom** rather than resetting at each prompt.

These founding ideas lead to a core vision statement: **TORI is not just a development tool or chatbot; it is a thinking partner that grows with you.** The IDE is envisioned as a “*Conceptual Canvas*” where code and design interact dynamically. The chat is envisioned as a multi-mind conversational agent that can reason through complex problems with traceable logic. Together, they form a unified cognitive workbench for creators.

From this vision, several **guiding principles** direct our design and development:

* **Human-Centric Design:** TORI keeps the human user in control and informed at all times. A guiding principle is to *“keep humans in the loop”* throughout the experience. Features like explainable suggestions, the ability to inspect the concept graph, or toggling agent assistance on/off are critical. The user should feel TORI is amplifying their abilities, not automating them away without transparency.
* **Transparency and Explainability:** Every AI-driven action in TORI should be explainable via the concept graph or a clear rationale. If TORI IDE suggests a refactoring, it can point to the concept nodes and phase relationships that led to that suggestion (e.g. a function concept that is out-of-phase with others, indicating a design inconsistency). TORI Chat, similarly, doesn’t just answer — it can provide a brief explanation or show which internal “train of thought” (which agent or knowledge source) produced each part of the answer. This fosters trust and positions TORI as a *glass-box AI*, in contrast to the opaque *“black-box magic”* of many AI assistants[ibm.com](https://www.ibm.com/think/topics/black-box-ai#:~:text=White%20box%20AI%2C%20also%20called,Users).
* **Deterministic Core, Creative Edges:** TORI’s architecture emphasizes *deterministic reasoning* in its core algorithms (the spectral-phase logic, rule-based inferences, etc.) to ensure reproducibility and auditability. However, we recognize the need for creativity and flexibility at the user interface; therefore, stochastic AI elements are allowed at the periphery (for instance, a generative language model might be used to polish natural language summaries or suggest variable names). The principle is to use randomness only where it’s safe and beneficial (creative suggestions), but never in the core understanding of code or context. This yields a system that is both **reliable** and **inventive**.
* **Local-First and Privacy:** TORI is built with a *local-first* mindset – users retain ownership of their code, data, and even the AI’s reasoning process. Both TORI IDE and TORI Chat can run fully locally or on-premises, meaning no sensitive data leaves the user’s machine unless explicitly allowed. This addresses the needs of enterprises and individuals concerned with cloud privacy. By avoiding cloud dependence, TORI also achieves lower latency and tighter integration with the user’s environment (accessing local files, dev tools, etc.), all while ensuring compliance with data governance policies.
* **Modularity and Extensibility:** The cognitive OS is modular by design. Each subsystem (parsing, concept storage, reasoning agents, UI) is a separate component with well-defined interfaces. This ensures that future improvements or replacements (e.g., swapping in a new code parser for a different language, or adding a new reasoning agent) can be done without disrupting the whole. It also means parts of TORI can be extended or used independently – for example, third-party developers might write new ALAN agents or integrate TORI’s concept graph into their own tools. Modularity also improves maintainability; given TORI’s ambitious scope, being able to upgrade pieces in isolation is crucial.
* **Consistency with Developer Workflow:** Especially for TORI IDE, a key principle is *“meet developers where they are.”* TORI introduces new concepts (like spectral debugging or concept navigation), but it also supports familiar workflows (editing code text, using Git, running tests) so as not to alienate users. TORI’s advanced features augment – rather than replace – a developer’s normal tasks. For instance, the concept graph is accessible for insight, but it doesn’t force a new paradigm on the developer; one can code normally and tap into TORI’s conceptual insights on demand. Likewise, TORI Chat is designed to integrate with tools developers already use, like Slack or other team communication platforms (for example, via a plug-in or API that allows TORI Chat’s capabilities to be accessed in a Slack channel). The motto here is *“low floor, high ceiling, wide walls”* – easy to start using, with immense power as you learn more, and broadly applicable to many tasks.
* **Inspiration and Playfulness:** While scholarly rigor underpins TORI (drawing on cognitive science and spectral theory), the project also embraces a creative, playful spirit. This is reflected in internal code names and metaphors (“**Banksy**” oscillators, “concept cartridges,” etc.), and should come through in the product identity. We believe a touch of playfulness encourages exploration – for example, TORI might include Easter eggs or humorous messages in its UI (configurable based on user preference). This principle helps differentiate TORI’s brand: it’s deep tech with a human touch and a wink of fun where appropriate. (As one quip in the planning stage put it, *“TORI isn’t a tool; it’s the mirror in which your code practices mindfulness.”*)

These principles ensure that as we develop TORI, we maintain a balance of intellectual rigor (truly innovating in AI) and user empathy (genuinely empowering users in intuitive ways). The next sections translate this vision and these principles into a concrete architecture and development plan.

**Cognitive OS Architecture Overview**

TORI’s architecture is a modular, layered cognitive system designed to support both an IDE and a Chat application via a shared core intelligence. At the highest level, the architecture comprises four integrated subsystems that interact closely:

* **ALAN 2.x Cognitive Core –** the “brain” of the system, responsible for understanding and reasoning over the Large Concept Network.
* **Language/Domain Integration Layer –** feeds the core with domain-specific inputs (code for the IDE, natural language for the Chat) and translates the core’s outputs back into human-friendly form.
* **User Interface & Tools –** the front-end through which users interact (the code editor canvas, debugging dashboards, chat interface, etc.).
* **Multi-Agent Orchestration –** the coordination layer managing specialized AI agents that perform distinct roles (e.g., a Refactoring Agent in the IDE, a persona-based Expert Agent in Chat), synchronized via the ALAN core.

Below, we describe each of these subsystems in detail, then explain how they come together specifically in TORI IDE and TORI Chat.

**ALAN 2.x Cognitive Core (Conceptual Engine)**

The ALAN 2.x core is the heart of TORI: a cognitive engine that represents and reasons about knowledge in a **Large Concept Network (LCN)**. This is effectively a graph of concepts (nodes) and their relationships (edges) spanning code, documentation, and conversational context. The core’s distinguishing feature is its *spectral-phase reasoning* architecture – inspired by dynamical systems theory and cognitive science – which ensures that TORI’s reasoning is grounded in **temporal synchrony** and **spectral analysis** rather than static rules or purely stochastic jumps.

At a high level, the core operates as follows: each concept in the LCN is associated with internal state variables, including one or more **oscillators** that represent the concept’s “phase” or activation rhythm. When the system processes information (new code, a user query, etc.), concepts are activated and interact via these oscillatory states. The engine analyzes the collective behavior of these oscillators using spectral methods to infer coherence, causality, and emerging patterns of thought. In essence, TORI treats reasoning as a *dynamic process*—a kind of controlled resonance among ideas. This approach is loosely inspired by how human brains exhibit synchronized neural oscillations during cognition, where groups of neurons firing in unison can bind together related pieces of information for unified processing[neurosciencenews.com](https://neurosciencenews.com/brain-rhythm-cognition-25941/#:~:text=It%20could%20be%20very%20informative,%E2%80%9D). By simulating a form of phase synchrony among concept nodes, ALAN aims to achieve a machine analog of that cognitive binding: concepts that should logically align will tend toward phase alignment, whereas contradictory concepts will oscillate out-of-sync, flagging inconsistency.

**Phase-Coupled Oscillator Matrix (“Banksy” Core)**

At the core of this dynamic reasoning is the **phase-coupled oscillator matrix**, code-named “Banksy.” This is essentially a network of mathematical oscillators attached to concept nodes, modeling the timing and rhythm of concept interactions. It draws inspiration from the Kuramoto model (a well-known model of coupled oscillators for synchronization) but extends beyond it. In ALAN 2.x, we have **replaced the simple Kuramoto synchronization with a ψ-based spectral coupling approach** – using advanced techniques from Koopman operator theory and Lyapunov stability analysis to govern how oscillators influence each other’s phase. Each concept node may have an oscillator representing its state in a given context or reasoning “mode.” For example, in the IDE context, an oscillator might track how “in sync” a module is with the overall codebase design; in the Chat context, oscillators might represent alignment or divergence of a user’s statements with factual knowledge or a persona’s beliefs.

When concept A influences concept B (say, function A calls function B in code, or a user query references a knowledge concept), the oscillators of A and B are coupled: they tend to synchronize if the relationship is harmonious or diverge if there is tension. By observing phase alignment or divergence across the network, the ALAN core can detect coherence or conflict among ideas (e.g., two design components oscillating out-of-phase might indicate a logical contradiction or an API mismatch that needs resolution). The system’s use of **ψ (psi) functions** refers to the eigenfunctions in the Koopman operator framework (discussed below) that capture the essence of these oscillatory modes.

Critically, this oscillator matrix provides a temporal dimension to reasoning. Rather than evaluating code or chat context with static rules, TORI simulates a **continuous reasoning process** that can converge toward stable conclusions. The introduction of Lyapunov stability metrics ensures that this process remains stable and convergent. In control theory terms, we treat a reasoning session as a dynamical system that should settle into an equilibrium representing a consistent understanding or answer. If a small perturbation (say a minor code change or a slight rephrase of a question) causes only a transient wobble but the phases re-synchronize, the system is stable in the sense of Lyapunov – meaning the reasoning is robust to minor changes[ocw.mit.edu](https://ocw.mit.edu/courses/16-30-feedback-control-systems-fall-2010/463e9559c6ef70f13ea51c3464bdc9c1_MIT16_30F10_lec22.pdf#:~:text=%E2%80%A2%20Stable%20in%20the%20sense,%E2%80%9D%20November%2027%2C%202010). This approach is novel in AI reasoning: we are effectively designing the AI’s thought process to have provable stability properties, which guards against chaotic jumps or oscillatory deadlocks in inference.

*(Implementation Note:* As of May 2025, the phase-coupled oscillator subsystem is **partially implemented**. Basic oscillator objects and their coupling equations exist, but the sophisticated ψ-based synchronization algorithm is still in prototype stage. Earlier prototypes used a plain Kuramoto model for phase synchronization; the new approach will incorporate the **Koopman spectral method** described next, which is a work in progress. Finishing this oscillator sync logic with the new model is a top priority before feature sprint work resumes.)

**Koopman Morphic Engine (Spectral Analyzer)**

Complementing the time-based oscillator model, ALAN 2.x employs a **Koopman morphic engine** for spectral analysis of the concept network’s state. The **Koopman operator theory** provides a powerful framework to analyze nonlinear dynamic systems by lifting them into a higher-dimensional linear space via eigenfunctions[mdpi.com](https://www.mdpi.com/2227-7390/9/19/2495#:~:text=The%20Koopman%20operator%20theory%20is,are%20amenable%20to%20systematic%20analysis). In simpler terms, our engine monitors changes in the concept graph and oscillators and projects those changes into a spectral domain (frequencies/modes) to find underlying patterns or invariants in TORI’s reasoning.

For example, in the IDE, as the codebase evolves over time (commits, refactors, additions), the engine can detect repeating patterns or cycles in development by analyzing the sequence of concept graph states – akin to finding a “frequency” at which a certain design issue reoccurs. In TORI Chat, the Koopman analysis might identify recurring discussion themes or predict how a dialogue could evolve (almost like sensing the “direction” or *momentum* of the conversation). The engine treats conceptual changes like signals, and by applying spectral decomposition, it can forecast future states or recognize known patterns. Essentially, it gives TORI a rudimentary ability to *anticipate* and plan, not just react.

Mathematically, if the concept state at time *t* is represented by some vector of features (active concepts, phase differences, etc.), the Koopman engine tries to approximate a linear operator **K** that evolves these features: xt+1≈Kxt.x\_{t+1} \approx K x\_t.xt+1​≈Kxt​. Even though the underlying system is nonlinear, **K** (the Koopman operator) operates on a lifted representation (comprised of eigenfunction observables, often denoted ψ). This allows use of linear techniques: for instance, we can compute eigenvalues of **K**, which correspond to modes of system behavior (stable modes, oscillatory modes, etc.). The spectral decomposition yields **Koopman eigenfunctions** that represent coherent patterns in the reasoning process[mdpi.com](https://www.mdpi.com/2227-7390/9/19/2495#:~:text=The%20Koopman%20operator%20theory%20is,are%20amenable%20to%20systematic%20analysis). Some of these modes might correspond to stable invariants (e.g., a conclusion that remains true throughout the session), while others might indicate unstable directions (e.g., a line of reasoning that diverges or an idea that keeps oscillating without settling).

We also explicitly integrate **Lyapunov exponents** into this analysis: these measure how small deviations in state evolve (do they die out or amplify?). A negative Lyapunov exponent indicates a stable mode (perturbations decay), whereas a positive one would warn of chaotic tendencies. By monitoring these, TORI can assure that its reasoning stays within stable bounds. In practice, this means if TORI begins to chase a line of thought that is leading to divergence (say, an oscillation that grows or an argument that spirals without resolution), the system can detect it and either dampen that interaction or invoke a conflict-resolution agent to intervene. This is an example of **spectral modal alignment**: ensuring that the modes (eigenvectors) of the reasoning process align with convergence and consistency criteria.

*(Implementation Note:* Currently, the Koopman spectral analysis in ALAN 2.x is also **in prototype form** – a scaffold exists that can record state snapshots and perform a rudimentary spectral transform (e.g., via an FFT or eigen-decomposition) but it returns dummy results. Integrating a true Koopman operator computation (likely leveraging an existing numerical library for eigen-analysis) is planned. We may not implement a full infinite-dimensional Koopman (which is theoretically complex), but a finite approximation capturing dominant modes. This will be developed alongside the oscillator sync. The design goal is that by the end of our core integration, the system’s inference pipeline will be: **concept graph update -> oscillator dynamics -> spectral (Koopman) analysis -> stable state check**, yielding a cycle of inference that is *explainable* and *provably convergent*.)

**Static Knowledge Kernels and Memory**

In addition to the dynamic reasoning modules above, the ALAN core includes a static knowledge comparison component, sometimes referred to as **spectral kernels** or embedding similarity modules. While the oscillator network and Koopman engine provide a view of *dynamic* behavior, the spectral kernels handle *static* semantic comparisons – for example, finding which concepts in the graph are semantically similar, or which past scenarios (code patterns, Q&A pairs) resemble the current one. In implementation, this involves using vector embeddings for concepts (perhaps pre-trained or learned over time) and computing similarities. This subsystem complements the phase-based reasoning with more conventional AI pattern-matching (for instance, identifying that a newly encountered concept “login routine” is similar to a past concept “authentication function” based on their embeddings, and thus inferring related constraints).

The Large Concept Network, combined with these spectral and embedding-based tools, forms a **memory substrate** for TORI. Crucially, this memory is **deterministic** and **replayable**: given the same sequence of inputs and events, the concept graph and its oscillator states should evolve the same way every time. This determinism is important for auditability (especially in enterprise use) – one can trace exactly how a conclusion was reached.

*(Current Implementation:* The LCN data structure and basic semantic embedding functions are functional. For example, TORI can already ingest code and produce a network of concepts and simple similarity links. However, the full closed-loop where the spectral reasoning informs updates to the concept graph is not yet realized. Completing this closed-loop reasoning pipeline is an immediate objective.)

**Language & Domain Integration Layer**

This layer bridges raw inputs (source code, natural language text, etc.) with the ALAN core’s conceptual format. It ensures that no matter what form information comes in – be it a Python file, a snippet of our custom DSL, or an English sentence – it gets translated into the *concepts* that the core can work with, and likewise translates the core’s conceptual outputs back into human-readable answers (code suggestions, explanations, etc.).

**Programming Language Frontends (for TORI IDE):** TORI IDE supports multiple programming languages through dedicated frontends. For each supported language, the frontend includes a parser or utilizes an existing compiler/AST generator to produce an abstract syntax tree (AST). The AST is then traversed and converted into updates to the LCN. For example, when you feed a Python file into TORI, the Python frontend identifies structures like classes, functions, and variables and their relationships (function A calls function B, Class X inherits from Class Y, etc.). Each of these constructs maps to concept nodes in the LCN (with edges such as “Function *encrypt* calls Function *hash*” or “Class *PaymentService* implements Interface *Service*”). The integration layer also extracts any semantic metadata (docstrings, type annotations) to attach to the concept nodes as needed.

In the current implementation, **Python** and our custom **ELFIN** DSL are first-class citizens: they have the most complete parsers and mapping logic. C/C++ and JavaScript/TypeScript are supported in a secondary manner – using external tree-sitter parsers or language server protocols to get an AST, which is then mapped into the concept graph in a generic way. (In other words, TORI can ingest C++ code, but the depth of understanding is currently limited compared to Python/ELFIN where we have custom semantics. Expanding language support will be important post-v1.0.)

**Domain-Specific Ontologies (for TORI Chat & Enterprise):** For TORI Chat, especially in enterprise settings, the integration layer can ingest domain knowledge such as ontologies, taxonomies, or glossary files and incorporate them into the LCN. For instance, a financial institution could feed in an ontology of banking terms, or a medical user could input a set of medical protocols. The integration layer will convert those into concept nodes and edges so that the chat’s reasoning core becomes aware of domain-specific concepts and their relationships. We refer to these packages of domain knowledge as **“concept packs.”** For example, if a *Finance Pack* is loaded, TORI Chat will know that concepts “KYC check” and “AML regulation” are related compliance ideas and should be treated accordingly during reasoning. This greatly enhances the system’s expertise in specialized areas, essentially bootstrapping the LCN with domain-specific subgraphs.

**Natural Language Processing (for TORI Chat):** When a user enters a query or message in TORI Chat, the NLP component of the integration layer processes it. This involves standard steps: tokenization, syntactic parsing (identifying the grammatical structure), and semantic interpretation. Key phrases or entities in the user’s input are mapped to concept nodes in the LCN (either matching existing ones, or creating new nodes if a novel concept is mentioned). For example, if a user asks, “How does the sorting algorithm in my code work?”, the NLP will identify “sorting algorithm” as an entity. If the IDE side has already indexed a concept for the sorting function or algorithm from the code, the question is linked to that existing concept node. If no such concept exists yet, TORI may create a placeholder concept “sorting algorithm (unknown)” which an agent could then try to resolve by searching the code or documentation. Conversely, when TORI Chat formulates a response, the integration layer takes the reasoning output (which might be an abstract argument or a subgraph of concepts that form the answer) and generates fluent natural language from it. We use template-based or controlled language generation to ensure the answer is faithful to the underlying reasoning (rather than a free-form neural generation which might hallucinate details). For instance, if TORI’s reasoning finds that *Function X* is slow because *it uses a quadratic loop*, the answer might be generated as: “Function X is identified as a bottleneck because it uses a nested loop, leading to O(n^2) complexity, which slows down performance.”

**Bidirectional Sync:** A critical aspect of this layer is that integration is *bi-directional* and continuous. As the user updates code in the IDE, those changes stream into the ALAN core via the language frontend (e.g., on each file save or even keystroke, for live analysis). The concept graph and oscillator states update in near real-time, so TORI’s understanding stays in sync with the codebase. Likewise, for Chat, as the conversation progresses, new concepts introduced are added to the LCN, and any conclusions drawn can be fed back (for example, if TORI solves a problem in chat, that knowledge can become a concept node which the IDE might use as a hint or comment in code). This ensures TORI IDE and Chat remain *synchronized* through the shared core knowledge.

*(Current Status:* The Python code parser is **mostly operational** (using AST from ast module plus custom logic to populate the LCN). The ELFIN DSL parser is **in progress** with initial grammar and concept mapping defined, but not fully integrated. Other languages (C/C++, JS) rely on external parsers and have basic mapping; these are less complete. The natural language pipeline for TORI Chat exists (leveraging a spaCy-based parser and simple entity linker), but it’s rudimentary and will need improvement for nuanced understanding and generation. Enhancing the NLP, especially for complex multi-sentence queries and fluid dialogue, is on the roadmap.)

**The ELFIN DSL: A Domain-Specific Language for Cognitive Coding**

One unique element in TORI’s language integration is the inclusion of a custom *domain-specific language* called **ELFIN**. ELFIN is an experimental DSL being developed alongside TORI to explore and demonstrate **concept-first programming** — a style of coding where the fundamental units are abstract concepts rather than concrete syntax. The name “ELFIN” evokes something small and magical, hinting at its role: it is designed to be a concise language that nonetheless wields the “magic” of TORI’s cognitive engine.

**Purpose of ELFIN:** Why create a new programming language? ELFIN serves several purposes. First, it acts as a **testbed** for TORI’s cognitive features. Because we control the language design, we can ensure that ELFIN’s syntax and semantics map cleanly onto the LCN. This allows us to push the limits of concept-based analysis — ELFIN programs can encode high-level intents (for example, one could imagine an ELFIN construct for “goal” or “assumption”) that TORI will directly capture as concept nodes with specific relationships (like a goal node linked to steps to achieve it, or an assumption node linked to where it’s used). In essence, ELFIN is a language built *for* a cognitive OS, meant to express not just computations, but the reasoning behind them.

Second, ELFIN provides a **vehicle for features** that would be hard to integrate into existing languages. For instance, we can experiment with syntax for declaring concept relationships explicitly (perhaps an ELFIN statement to declare two functions conceptually analogous, or to tag why a decision was made). These might look like comments or annotations in other languages, but in ELFIN they can be first-class citizens. This enriches the concept graph beyond what we could infer from standard code.

Third, ELFIN ensures that TORI is not tied to the quirks of any single legacy language. It’s forward-looking: as we improve ALAN’s reasoning, we may extend ELFIN to naturally expose those capabilities. One could envision ELFIN being used for writing *AI workflows* or *agent scripts* for TORI itself. For example, an advanced use-case might be an ELFIN script that describes a development task (in quasi-natural language, but formalized) which TORI can then execute or assist with via its agents. In this sense, ELFIN blurs the line between code and specification, providing a language for *metaprogramming the AI*.

**Integration State:** Currently, the ELFIN language is in an **early stage**. A grammar has been defined and a parser implemented (the elfin\_lang.proto definition and a parser generator were started). Basic ELFIN programs – e.g., simple function definitions and concept declarations – can be parsed into the concept graph. The core concepts of ELFIN (like functions, invariants, or relationships) are mapped to LCN nodes similarly to how Python constructs are mapped. However, more advanced features (and the full runtime semantics) are not finished. For now, ELFIN code doesn’t “run” in the traditional sense; its value is in populating the concept network. We have integrated some initial ELFIN scripts into tests to ensure TORI can digest them. For example, an ELFIN snippet that defines a high-level plan (as a series of steps with goals) successfully creates a chain of concept nodes that the reasoning core can use to guide the Refactoring Agent. But this is a prototype demonstration. Before TORI’s launch, we plan to finalize the ELFIN spec for the most crucial features and ensure the integration layer can handle round-trip conversion (editing ELFIN in the IDE and getting meaningful assistance, much like with Python).

**Link to Concept Graph:** ELFIN is, by design, tightly coupled with the concept graph. Every ELFIN language construct has a direct representation in the LCN. If a developer writes an ELFIN function that “describes” a process (instead of explicitly coding it), TORI will create concept nodes for the abstract process, any sub-tasks, and link them according to the ELFIN syntax. Think of ELFIN as writing pseudo-code that is *immediately semantic*. For example, an ELFIN construct might allow:

javascript

Copy

assume User is\_authenticated

function transferFunds(amount) {

require amount <= Account.balance

ensure Account.balance\_decreased\_by(amount)

//...

}

In this pseudo-ELFIN example, assume and require/ensure might be ELFIN keywords. TORI would map this into concepts like *User (is\_authenticated)* as a fact, *transferFunds* as a function concept, a precondition concept linking *transferFunds* to the requirement on *amount and balance*, and a postcondition concept linking to the balance decrease effect. These become part of the LCN, meaning TORI “knows” the intent and constraints of the code, not just its syntax. During reasoning (say the Test Generator Agent or a bug-finder), TORI can use these concepts to reason about correctness (the assume/require are like formal specs in the concept space).

By linking ELFIN directly into the concept graph, we also make it possible for TORI to explain code behavior in higher-level terms. If part of the system is written in ELFIN, TORI Chat could potentially answer questions about it by directly citing the concepts that ELFIN introduced (e.g., “The transferFunds function ensures post-condition that the balance is reduced by the transferred amount”). In short, ELFIN serves as both a programming language *and* a knowledge encoding scheme for TORI’s AI.

*(Outlook:* While ELFIN will be included in the TORI v1.0 release as an experimental feature, its true potential lies beyond the initial launch. It could evolve into a powerful tool for users to extend TORI’s intelligence, by encoding new knowledge or procedures in a form the cognitive core can inherently work with. In the long run, success of ELFIN will be measured by whether it attracts real usage; for now, it’s a strategic investment to showcase TORI’s concept-oriented philosophy and to ensure we have a maximal internal test of the system’s capabilities.)

**User Interface & Tools (Thoughtboard IDE and Chat UI)**

The User Interface layer encompasses all the front-end components through which users interact with TORI’s intelligence. This includes the **TORI IDE interface (Thoughtboard)**, various panels and visualizations within the IDE, and the **TORI Chat interface** for conversational interactions. A core principle of the UI is to **augment familiar workflows with cognitive superpowers**: developers should feel at home with the basic tools (editor, console, etc.), enhanced subtly and powerfully by TORI.

**TORI IDE – “Thoughtboard” Interface:** Rather than piggybacking on an existing IDE via plugin, we have created an independent, local-first application for TORI IDE, code-named **Thoughtboard**. Thoughtboard is a custom UI (built with a modern tech stack, currently a Python/Electron prototype with plans for a polished cross-platform app) that serves as the primary workspace for the developer. It presents the user’s code in a code editor canvas but layered with TORI’s cognitive insights. Key UI elements include:

* An **editor with inline annotations**: as you write or browse code, TORI can underline sections that have conceptual issues (e.g., out-of-phase functions) or insert subtle markers where an agent has a suggestion. For example, a faded lightbulb icon might appear next to a function that TORI believes could be refactored, and hovering reveals the suggestion.
* A **Concept Graph Explorer**: a sidebar or pop-up view that visualizes the local portion of the concept network related to the current context. If you click on a function, the explorer might show connected concepts like “uses algorithm X” or “related requirement: Y” in a node-link diagram. This allows developers to navigate code semantically, not just by filenames.
* **Spectral Debugging Dashboard**: a panel available during runtime or analysis that displays the oscillator phases and other dynamic metrics. For instance, it could show a live chart of module synchronization – if one module’s oscillator falls out of sync (signaling a potential bug or mismatch), the dashboard highlights it. This is akin to a performance monitor but for conceptual consistency and phase alignment.
* **Project Insights & Warnings**: Much like how an IDE shows compile errors or linter warnings, TORI IDE (Thoughtboard) shows cognitive warnings. For example, “Module *AuthService* is conceptually out-of-phase with Module *UserService* (they interact but have inconsistent assumptions)” or “Concept drift detected: the current code usage of concept *X* differs markedly from past usage – consider reviewing for unintended changes.” These come from the ALAN core’s deeper understanding. Each warning or insight is explainable – the UI lets the user click to see the rationale (which agent or rule triggered it, and what concept relations are involved).
* **Standard IDE features, enhanced by AI:** Thoughtboard also integrates version control, testing, debugging, etc., with AI support. For instance, during debugging, if a test fails, TORI can cross-reference the failing test’s concept footprint with recent code changes and *immediately* suggest which change likely caused the regression (because it knows which concept nodes were affected by the change and are linked to that test). When searching code, instead of simple text search, the user can do a *conceptual search* (“find all code related to user authentication”) and TORI will return results across files and languages, because it knows what parts of the codebase pertain to that concept.

All these features are delivered in a cohesive UI. The design philosophy is to not overwhelm the user – TORI’s help should feel like a natural extension of the IDE. A developer can use Thoughtboard in a minimal way (just as a code editor with some extra info, ignoring most cognitive panels), or dive deep into TORI’s new capabilities as needed. The UI is kept responsive and **local-first**: all analysis is done locally by the ALAN core, so these features work in real-time without cloud latency.

*(Current UI Status:* A prototype of the Thoughtboard app exists. It currently uses a VS Code-like layout for familiarity – a file explorer, editor pane, and an output/terminal pane – but augmented with a concept graph panel and debug phase panel in experimental form. This prototype is functional for basic editing and displaying dummy AI annotations. The **VS Code extension scaffold** that was used early on has been phased out in favor of the standalone Thoughtboard app to ensure a tighter integration and offline-first operation. The UI is not yet feature-complete or polished: for example, the concept graph viewer works (it can show the graph from the last analysis in a separate window) but is not interactive in real-time. Over the next development sprint, a major goal is to flesh out the Thoughtboard UI: integrate the live concept explorer, make the debug phase panel truly live with the oscillator data, and implement all the needed controls and polish. By launch, the IDE should feel like a modern, slick environment where the cognitive features are intuitive and valuable.)

**TORI Chat Interface:** TORI Chat’s UI will allow users to converse with the cognitive engine, either in standalone mode or integrated within Thoughtboard. For v1.0, our plan is to provide **two interfaces** for TORI Chat:

1. **Integrated Chat Panel in Thoughtboard IDE:** This is a panel or sidebar where the user (likely a developer) can ask questions about the code or get assistance. For example, a developer can highlight a block of code and ask in the chat panel, “How does this function work?” or “Find potential bugs in this logic,” and TORI Chat (leveraging the IDE’s context) will respond. This tight integration means the chat has full access to the project’s concept graph and can act as an in-IDE assistant.
2. **Standalone Chat Application:** In addition, we envision a dedicated chat client (likely web-based or Electron-based) for TORI Chat, targeting more general usage or enterprise deployment. This standalone TORI Chat app would function like a powerful AI assistant that can be run on-prem. It would have features like multi-turn conversation memory, persona selection (e.g., you could talk to “TORI the Python expert” or “TORI the Project Manager assistant”), and tools for the user to upload documents or knowledge packs in-chat.

*(Current UI Status:* At present, the chat UI is minimal. We have a basic web page that allows inputting a query and seeing the text response from the backend – mainly used for testing the back-end. It’s essentially a simple chat log with no advanced UI elements. There’s no rich formatting, and features like conversation history beyond a scrollback are rudimentary. During development, initial focus will be on the integrated IDE chat panel (since that directly assists developers alongside the IDE). The standalone app will be scaffolded – likely leveraging some components of the Thoughtboard UI for consistency. Voice input, persona visualization (e.g., an indication of which agent is “speaking” in a multi-agent answer), and other enhancements are slated for later development once core chat functionality is solid. The key for v1.0 is to ensure TORI Chat can handle multi-turn Q&A reliably and that enterprise users can deploy it securely on their own machines with an interface that, while basic, is usable and polished enough for demo purposes.)

**Multi-Agent Orchestration Layer**

One of TORI’s most innovative aspects is its **multi-agent architecture**. Instead of a single monolithic AI trying to do everything, TORI employs multiple specialized AI agents, each with a focused role or expertise, coordinated by a central **Orchestrator**. This approach is analogous to Marvin Minsky’s “Society of Mind” theory, where intelligence emerges from the interaction of many simple agents[en.wikipedia.org](https://en.wikipedia.org/wiki/Society_of_Mind#:~:text=In%20his%20book%20of%20the,2). By having a team of AI sub-components, TORI can tackle complex tasks more robustly: each agent contributes its perspective (coding best practices, testing, high-level strategy, etc.), and the orchestrator mediates their interactions, much like a project lead coordinating team members.

**Orchestrator:** The orchestrator is a supervisory module within ALAN’s core that listens to inputs and events (new code changes, a user question, an inconsistency detected) and decides which agent(s) to invoke, and how to merge their results. It operates in sync with the phase logic — meaning it can use the oscillator network to detect when agents disagree (if their suggested concepts are out-of-phase) and resolve conflicts. The orchestrator ensures the agents work in concert rather than at cross-purposes. If two agents produce conflicting suggestions (e.g., one agent suggests renaming a function while another suggests deleting it), the orchestrator, guided by global rules or meta-agents, will decide on a resolution or ask for clarification from the user.

**Agents in TORI IDE:** In the IDE context, we have several specialized agents either implemented or planned:

* **Refactoring Agent:** Scans the code’s concept graph for code smells or refactoring opportunities. For example, if it finds two separate modules implementing similar concepts, it suggests an abstraction or utility. It also monitors complexity metrics via the concept graph, prompting the user to simplify a function that grows overly complex. *(Status: partially implemented – can detect simple duplications, though deeper conceptual duplications use placeholder logic.)*
* **Debug Advisor Agent:** Watches program execution traces and the oscillator signals during runs to spot likely causes of errors or performance issues. For instance, if during a test run, memory-related concept nodes spike in activity, it might infer a potential memory leak or inefficiency and point the developer to it.
* **Documentation Agent:** Ensures that documentation in the concept graph stays updated with code changes. It can draft docstrings or explanations for new code by synthesizing information from similar past concepts and commit messages. If a developer highlights a function and asks “explain this,” this agent works with the chat subsystem to provide an answer drawn from the concept context. *(Status: currently stubbed – it creates placeholder documentation nodes but doesn’t generate text yet; integration with an LLM for language generation is planned but to be done carefully under the deterministic core principle.)*
* **Test Generator Agent:** Analyzes the concept graph to find untested requirements or edge cases. It proposes new unit tests for parts of the code that lack coverage or identifies when a change in one concept should trigger re-testing of related concepts. For example, if a new concept “encryption” was added, it might suggest tests for that, or if a concept has changed, ensure all dependent concept tests are revisited.

These agents operate continuously or on triggers (like after a code analysis pass). They post their suggestions back into the system as structured data – e.g., the Refactoring Agent might add a “Suggestion” node linked to the functions in question, which the UI then surfaces as a tip to the user.

**Agents in TORI Chat:** The chat context also benefits from specialized agents:

* **Knowledge Retrieval Agent:** When a user asks a question, this agent fetches relevant information from the LCN (and potentially allowed external sources) to gather facts. It’s like a librarian ensuring that if the answer is in the docs or code, TORI finds it.
* **Logic/Reasoning Agent:** This agent attempts to perform multi-step reasoning for complex queries, using the knowledge assembled. Essentially, it chains together pieces of knowledge (possibly using the oscillator network to maintain coherence) to form an answer. In some implementations, this could be a rule-based inference engine or a constraint solver.
* **Persona/Style Agent:** For TORI Chat, especially if responding as different personas (tutor vs. devops assistant, for example), this agent modulates the response style, tone, or perspective. It ensures the answers align with the selected persona or context (e.g., enterprise compliance vs. casual helper).

Beyond these, we have plans for additional agents to broaden TORI’s reasoning scope:

* **MPorter (Strategy Agent):** *Planned.* This is a high-level strategy and planning agent, humorously named in homage to Michael Porter (known for strategic frameworks). The MPorter agent’s role is to reason about overall objectives and plans. In the IDE, it might watch the sequence of development tasks and guide the order of operations (for instance, suggesting “Before refactoring X, address the failing test Y”). In Chat, it could maintain the long-term direction of the conversation or break down a complex query into sub-goals for other agents to solve. Essentially, MPorter thinks about “the big picture” — ensuring TORI’s various actions are aligned with the user’s overarching goals and making strategic choices if there are trade-offs. This agent brings a sort of executive function or meta-reasoner into the system, planning multi-step solutions.
* **Attachment (Emotional Intelligence) Agent:** *Planned.* This agent focuses on the emotional and relational aspect of interactions, drawing from concepts in affective computing. It monitors the user’s tone and sentiment (e.g., frustration, confusion, satisfaction) and the relationship context. In TORI Chat, the Attachment agent would adjust responses to be empathetic and supportive – for example, if a user seems frustrated by errors, the agent might suggest a more reassuring explanation or a gentle prompt to take a break. It can also remember personal preferences or emotional cues (almost like how a good human assistant remembers what frustrates their boss, or what style of feedback they prefer). In the IDE, this agent might manifest as the system “knowing” when to intervene: e.g., if a user has tried to solve a problem multiple times (detected by rapid code changes and undos), the agent might proactively offer help (“It looks like you’re encountering difficulties with this function. Would you like some suggestions?”). The Attachment agent essentially gives TORI a form of **emotional intelligence**, aiming for the AI to not just be correct, but also helpful and attuned to the user’s state[en.wikipedia.org](https://en.wikipedia.org/wiki/Affective_computing#:~:text=Affective%20computing%20is%20the%20study,interpret%20the%20emotional%20state%20of).

All these agents produce outputs that must be combined into a single, coherent assistance experience. The **Multi-Agent Orchestrator** manages this. It can run agents in parallel or sequence as needed. For example, when a new code analysis is done, the orchestrator might run the Refactoring, Test, and Documentation agents in parallel to gather all their findings, then integrate those findings (resolving any conflicts) before presenting to the user. In chat, orchestrator might orchestrate a debate between a “strict expert” agent and a “simplifier” agent if a question needs both perspectives, ultimately merging their answers.

*(Current Status:* The multi-agent system is implemented in a rudimentary form. In the IDE prototype, there are hooks for a Refactor and Debug agent that run on each code analysis cycle, but their coordination is basic (essentially a priority rule: refactor suggestions get logged first, etc.). The orchestrator logic that uses oscillator synchronization to mediate agents is not yet fully realized. We have simulated multi-agent reasoning in chat using a pipeline (knowledge retrieval → reasoning → answer formatting), but it’s not truly concurrent agents. During the next development phase, implementing the orchestrator’s conflict-resolution and expanding the agent roster (especially integrating MPorter and the Attachment agent logic) will be crucial. The end goal is that by Day 15 of the sprint, as scheduled, the multi-agent enhancements will allow TORI to handle conflicting suggestions and leverage multiple specialized agents seamlessly.)

**Why Multi-Agent?** This design is motivated by both technical and cognitive factors. Technically, it’s easier to maintain and improve specialized agents one by one than a giant AI that does everything. New agents can be added as plugins to increase functionality without overhauling the whole system. Cognitively, it mirrors how humans solve problems: we subconsciously employ different modes of thinking (analytical, recall-based, emotional judgement) and reconcile them. Indeed, the Society-of-Mind approach suggests intelligence can emerge from a colony of simple processes[en.wikipedia.org](https://en.wikipedia.org/wiki/Society_of_Mind#:~:text=In%20his%20book%20of%20the,2). By architecting TORI as a society of agents, we aim for a system that is **robust, extensible, and interpretable** – we can trace which agent contributed to which decision, and agents can even explain their reasoning steps (e.g., the Test agent could say “I suggested this test because I noticed concept X wasn’t covered by any existing tests”).

**Development Roadmap – 21-Day Sprint Plan (Starting May 12, 2025)**

To bring TORI from its current prototype state to a production-ready v1.0, we have outlined a rigorous 21-day development sprint for **TORI IDE** and a parallel 21-day sprint for **TORI Chat**. Each plan assumes **Day 1 is May 12, 2025 (Monday)**, and is structured as three focused work weeks with periodic review and buffer days. Below, time-based references (Day 1, Day 2, …) are annotated with their calendar dates for clarity.

**Note:** The sprint plans below presume that foundational work on the partially implemented core components (oscillator sync, Koopman spectral engine, ELFIN DSL parser, and orchestrator logic) is completed or nearly completed on Days 1–3. These core tasks are emphasized at the start of the IDE sprint. Without them, subsequent feature development would be building on unstable ground. Thus, the first week’s milestones heavily focus on finalizing the core.

**TORI IDE – 21-Day Sprint Plan (May 12 – June 1, 2025)**

**Sprint Goal:** Transform the TORI IDE from a promising prototype into a polished, usable product (v1.0) with all core features implemented and all placeholder components replaced. We allocate 21 days of effort (3 weeks) with one rest/review day each week (Day 7 and Day 14 as lighter days). The plan ensures continuous integration of features and frequent testing.

1. **Day 1 (May 12, 2025) – Sprint Kickoff & Core Setup**  
   *Goal:* Kick off development and ensure the team is aligned and the environment is ready. Also begin integration of core components.  
   *Tasks:*
   * Conduct a project kickoff meeting to review this Master Plan, clarifying the vision, architecture, and critical tasks for the sprint.
   * Finalize the development environment setup for all team members. This includes ensuring everyone can run the existing TORI IDE prototype (both backend and Thoughtboard UI). Resolve any environment issues (path configs, dependencies) so that development can proceed uniformly.
   * Merge any outstanding prototype code into the main branch. (For example, if there were separate branches for the new oscillator sync algorithm or the UI refresh, bring them in now.) We want a single code baseline at the start.
   * **Begin core integration:** If the advanced oscillator sync and spectral analysis components have code ready in prototype form, integrate them behind feature flags. This means plugging in the new Banksy oscillator update loop and Koopman analysis module, but perhaps toggled off until fully tested. This sets the stage for Day 2 and 3 where we’ll flesh them out.  
     *Checkpoint:* By end of Day 1, TORI IDE should run end-to-end in at least one environment with the new core components in place (even if dormant). All developers have a working setup. We have a clear list of the core integration subtasks to tackle in the next two days.
2. **Day 2 (May 13, 2025) – Core Engine Focus: Oscillator Synchronization Implementation**  
   *Goal:* Implement the **Banksy phase-coupled oscillator logic** in the ALAN core (replacing the placeholder with the real synchronization mechanism).  
   *Tasks:*
   * Design and document the updated synchronization algorithm using our ψ-based approach. This includes how oscillators will influence each other’s phase differences and what parameters (coupling strengths, frequencies) are needed. (We draw from known models like Kuramoto for inspiration but incorporate spectral feedback – for instance, use the Koopman-derived modes to adjust coupling dynamically.)
   * Begin coding the oscillator update function in the alan\_core module. Each cycle (or event trigger), this function will update all concept oscillators’ phases. Implement coupling: if two concepts are linked, their phase difference should adjust gradually (converging if harmonious, diverging if conflict signals).
   * Write unit tests for the oscillator update logic. Create a simple scenario: two concept nodes A and B with a link. Simulate a phase update where A is perturbed; verify that B’s phase moves closer to A’s over several iterations if the link indicates they should sync.
   * If time permits, integrate a debug command or log that prints out phase values for a test graph to visually verify behavior.  
     *Checkpoint:* By end of Day 2, the oscillator subsystem produces **non-trivial output** – i.e., phases are no longer static or random; they move over time in response to concept links. We should be able to demonstrate that if concept A and B are linked, their oscillator phases trend toward alignment (or an intended offset). All placeholder “mock oscillator” code is removed or replaced. The Kuramoto mention in code is replaced with our new algorithm (e.g., functions now named update\_phase\_sync instead of update\_phase\_dummy). This establishes the dynamic heartbeat of the cognitive core.
3. **Day 3 (May 14, 2025) – Core Engine Focus: Spectral Analysis & Prediction**  
   *Goal:* Flesh out the **Koopman morphic engine** to utilize concept graph dynamics for prediction and pattern recognition. In short, get the spectral analysis core working with real data.  
   *Tasks:*
   * Implement a basic spectral analysis pipeline: take snapshots of the concept state (e.g., concept activation levels or phase angles at intervals) and perform a simplified Koopman analysis. Concretely, start with something manageable like computing a Fourier transform of a few recent states to see dominant frequencies of change. If feasible, integrate a linear algebra library to compute eigenvalues of a state-transition matrix we define (for a simplified model of concept evolution).
   * Connect this analysis to the system: after each oscillator update (from Day 2’s work), run the spectral analyzer on the latest state. For now, maybe use a sliding window of the last N states. Identify if any eigenvalues/modes stand out (for example, a mode that’s not converging).
   * If full Koopman is too heavy to implement in one day, implement a *proxy*: e.g., track simple trends like “concept X has been steadily increasing in activation for 5 cycles” and flag that as a potential trend prediction (which could mean “we might see concept X trigger something soon”). The key is to get some predictive capability in place, even if rudimentary.
   * Add instrumentation: have the system log any predictions or detected patterns. For instance, “Koopman analysis: mode corresponding to concept ‘Y’ shows periodic oscillation (likely repeating pattern every ~10 commits).” This is mainly for development/testing now.  
     *Checkpoint:* By end of Day 3, the ALAN core should not only update state (from Day 2) but also **analyze state**. We should be able to point to a function or module where given some concept state history, we get a spectral decomposition or equivalent output. There should be no more hard-coded dummy returns in the analysis; even if it’s simplistic, it’s real. The core inference pipeline (concept update -> oscillator sync -> spectral check) is now implemented end-to-end. We can say: “The cognitive core’s major algorithms are in place.” Any issues or limitations encountered are noted to address in optimization days later.
4. **Day 4 (May 15, 2025) – Core Stabilization & Refinement**  
   *Goal:* Polish the core features implemented in Days 2–3, ensure they are stable and integrated. Address any major bugs or inefficiencies in the new core logic.  
   *Tasks:*
   * Review and refactor the oscillator and spectral code for clarity and performance. Since these will run frequently, ensure they are optimized (e.g., using NumPy for vectorized updates of phases instead of Python loops).
   * Add **Lyapunov stability checks** into the core loop if not already: for example, after updating oscillator phases, compute a simple metric of system stability (perhaps total phase variance or similar). Use this to log if the system is trending unstable. This will help later when multiple agents produce changes.
   * Write additional tests for edge cases: e.g., no links between concepts (oscillators should remain as-is), or all concepts fully linked (they should eventually sync up). Also test the spectral analyzer with known input signals (maybe feed it a sine wave and see if it identifies the frequency correctly) to verify it works as expected.
   * Integrate core outputs with a debug interface: perhaps extend the existing CLI or admin panel to display current concept phases and any detected modes. This helps the team manually inspect that the core is behaving plausibly.
   * If any core feature is not working or too noisy (for instance, if the spectral analysis is giving random “predictions”), decide on whether to tweak or gate it behind a feature flag so it doesn’t interfere with the rest of development. It’s okay if some of these advanced aspects are quiet in the background until we fine-tune them.  
     *Checkpoint:* By end of Day 4, the ALAN 2.x core should be *stable*: running it on a representative project or conversation does not crash, and it provides consistent outputs. The team should have confidence that this foundation is ready for layering on the IDE and Chat functionality. The “core to-do list” should now be mostly done, allowing focus to shift to user-facing features from Day 5 onward.
5. **Day 5 (May 16, 2025) – Thoughtboard UI Integration with Core**  
   *Goal:* Connect the updated ALAN core to the Thoughtboard UI, so that core insights (concept graph, warnings, etc.) start flowing to the front-end. Begin implementing interactive UI features that use the core data.  
   *Tasks:*
   * Hook up the **Concept Graph Explorer** panel in the UI to the actual LCN from the core. Replace any fake/demo graph data with real data coming from the backend via an API call. If not already done, create an API endpoint (or use existing one) like /concept\_graph that returns the current concept graph in JSON or protobuf form. The front-end should parse this and render the graph. Test by loading a sample project and opening the concept explorer – verify that it shows nodes corresponding to real classes/functions from that project.
   * Integrate **cognitive warnings** into the editor gutter. The backend (core or agent layer) should now be capable of producing at least basic warnings (e.g., from the Refactoring agent stub or from static checks). Define a data structure for these (if not done): say, a list of issues each with a type, message, and location (file/line or concept reference). Implement a mechanism for the UI to fetch or receive these. Possibly, whenever the core does an analysis pass, it could emit events that the UI listens to via a websocket or polling. Implement the simplest thing (maybe polling every few seconds or on demand). Then display these in the UI: e.g., highlight a line with a lightbulb icon where a suggestion exists. Clicking it could open a panel with the suggestion details.
   * Ensure the **Phase Dashboard** in the UI can show live data. This might involve a simple graph plotting library in the UI. For now, we can feed it a subset of the oscillator info (maybe just show relative phase offsets among major components). Hook it up to the backend: e.g., an endpoint /phases returns current phase values for top-level components. Update the UI at an interval. The goal is to replace placeholder sine wave graphics with actual output from our Day 2 core.
   * UI/UX fixes: as we integrate, likely some UI adjustments are needed (e.g., if the concept names are long, ensure the graph labels handle that, etc.). Tweak CSS or layout as necessary for a clean presentation.  
     *Checkpoint:* By end of Day 5, the Thoughtboard UI should be **functionally connected** to the AI core. We should be able to see real analysis reflected in the interface: for example, open a code file and if there’s a conceptual inconsistency, a warning appears in the UI coming from the core logic. The concept graph viewer should show at least a rudimentary graph of the code. Essentially, the front-end and back-end are talking to each other meaningfully, turning TORI IDE into a truly interactive cognitive IDE (even if the visuals are still rough).
6. **Day 6 (May 17, 2025) – Multi-Agent System Activation (IDE context)**  
   *Goal:* Enable and test the multi-agent framework within the IDE now that the core and UI are connected. Flesh out or refine a couple of key agents (Refactoring, Test generator) so they start providing useful output.  
   *Tasks:*
   * Integrate the **Refactoring Agent** fully: Now that the core is stable, revisit the Refactoring agent code. Improve its logic beyond the trivial duplicate check. For example, implement a check for functions that are conceptually very similar (perhaps using the embedding similarity: if two function concept vectors are very close, suggest factoring them). Also implement a check for any function out-of-phase with others (using the oscillator data: if one function’s oscillator is lagging significantly, maybe it’s not updated with new logic). These conditions can generate suggestions (e.g., “Function X might be outdated relative to related Function Y”).
   * Turn on the **Test Generator Agent** in a basic form: have it analyze the concept graph for any concept that has no associated test concept. If found, it can log a suggestion like “Consider adding tests for module Z; no tests found.” This is simplistic but sets the stage. If possible, integrate with actual test files (e.g., if code is Python, check if there’s a test module for each code module).
   * Implement the Orchestrator’s simple conflict resolution: if two agents try to tag the same code with different suggestions, determine how to handle it. Perhaps decide on a priority (Refactoring suggestions might be shown separately from Testing suggestions). In UI, maybe categorize suggestions by agent.
   * Ensure the UI displays multi-agent output distinctly if needed. Possibly add a filter or label to suggestions (“[Refactor] You have duplicate code in X and Y” vs “[Test] No tests cover module Z”). This transparency aligns with our explainability principle.
   * Start a log or telemetry of agent activity for debugging (e.g., console log: “RefactorAgent: checked 120 concepts, made 2 suggestions; TestAgent: made 1 suggestion.”). Useful for performance tuning later.  
     *Checkpoint:* By end of Day 6, multiple agents should be **active and contributing** in TORI IDE. A developer using the IDE at this point would see, for instance, a refactoring tip or two pop up after analysis, and maybe a note about tests. It need not be perfect or complete, but the multi-agent system is now live. We should also verify that their suggestions are sensible and not too noisy, adjusting thresholds if needed.
7. **Day 7 (May 18, 2025) – Review, Testing & Buffer**  
   *Goal:* Mid-sprint review of progress; fix any accumulated bugs from week one; ensure we haven’t broken basic functionality. Use this as a buffer/rest day.  
   *Tasks:*
   * Run a full **integration test** of TORI IDE as it stands. Open a sample project, use the main features: editing, viewing suggestions, triggering analyses. Note any crashes, UI glitches, or obviously incorrect suggestions.
   * Fix high-priority bugs discovered. For example, if the concept graph fails to update after certain edits, address that. Or if an agent suggestion is clearly wrong due to a logic bug, correct it.
   * If some planned tasks from Days 1-6 slipped, use today to catch up (buffer). For instance, if the Koopman analysis wasn’t fully integrated by Day 4, finish it now.
   * Solicit feedback from any team members or early testers if available. Sometimes a fresh pair of eyes on the UI or behavior can spot issues.
   * Take a breather to document any new technical debt or tasks that arose. Update the plan if needed for week two.  
     *Checkpoint:* End of Day 7 should have a **stable checkpoint build** of TORI IDE. Ideally, this is a point where if we had to do a demo of core capabilities, we could: concept graph viewing, a couple of cognitive suggestions, etc., all working. We also have a clear idea of what adjustments to make in the next phase.
8. **Day 8 (May 19, 2025) – Advanced IDE Features: Concept Explorer & Visualization**  
   *Goal:* Complete the implementation of the **Concept Graph Explorer** and other visualization tools in the IDE. Make interacting with the concept graph intuitive and useful.  
   *Tasks:*
   * Improve the **Concept Graph visualization**: Add interactive features such as zooming, panning, and clicking on nodes to reveal details. If not yet implemented, allow clicking a concept node to highlight the corresponding code in the editor (e.g., clicking a “User” concept node might highlight where the User class is defined or used).
   * Implement filtering in the graph view: for instance, toggle to show only certain types of relationships (maybe the user can filter to just “calls” relationships vs. “concept similarity” edges to reduce clutter).
   * Connect the graph view with the chat/QA: Perhaps allow the user to right-click on a concept in the graph and ask a question about it via TORI Chat (this sets the stage for cross-feature integration, though full chat integration comes Day 13).
   * If time, add a mini-map or hierarchical view for the concept graph (especially if the graph is large, a way to navigate it conceptually, like grouping by modules).
   * Ensure performance is okay: test on a larger project’s concept graph; optimize rendering or data fetching if it’s slow (e.g., implement lazy loading of parts of the graph).  
     *Checkpoint:* By end of Day 8, the **Concept Explorer** should be fully functional and user-friendly. This means a developer can open it, see a representation of their code’s concepts, and interact with it to understand their project’s structure better. It’s no longer a tech demo but a practical tool (even if it will be improved with polish later).
9. **Day 9 (May 20, 2025) – Agent Improvements & Orchestrator Logic**  
   *Goal:* Enhance the intelligence of the agents based on testing so far, and implement more sophisticated orchestrator behaviors (particularly conflict resolution and agent synergy).  
   *Tasks:*
   * Revise agent algorithms with insights gleaned from usage. For example, if the Refactoring agent frequently flags trivial issues, adjust its rules to be more meaningful (maybe require a higher similarity threshold for duplicate code suggestions, etc.). If the Test agent is too naive, incorporate more logic (e.g., look at coverage data if available, or at least ensure it doesn't repeatedly suggest the same test each run).
   * Implement agent communication through the orchestrator: design simple protocols where agents can pass hints to each other via the orchestrator. For instance, the Refactoring agent could mark a concept as “to be renamed” and the Documentation agent could pick that up and pre-prepare a new docstring. This could be simulated by having the orchestrator call agents in sequence and share a mutable context object.
   * Conflict resolution: If two agents produce contradictory suggestions (one says “remove function X” another says “improve function X”), determine a strategy. Possibly assign priorities or have the orchestrator present both suggestions but with context. Maybe integrate a simple rule: do no destructive suggestion (removal) if any other agent sees value in the item. Implement this in orchestrator logic.
   * Add at least one new small agent if possible to cover another aspect: e.g., a **Code Consistency Agent** that checks naming conventions or style (to show extensibility). Use it as a test of adding a new agent easily.
   * Continue to refine the **Attachment agent** (even in IDE, it can monitor user frustration by detecting rapid changes, etc.) though it’s more relevant in chat; maybe skip deep implementation here and focus on MPorter in IDE context. For instance, MPorter could monitor if multiple suggestions are pending and prioritize which one the user should handle first (a strategy element).  
     *Checkpoint:* End of Day 9 should see a **smarter multi-agent system**. Fewer irrelevant or conflicting suggestions should reach the user, thanks to orchestrator filtering. New synergies might be observable (perhaps documentation agent now auto-updates comments after a refactor suggestion is applied, etc., if implemented). We essentially have the multi-agent mechanism in a feature-complete state for the IDE side.
10. **Day 10 (May 21, 2025) – Performance Optimization & Scalability**  
    *Goal:* Ensure TORI IDE performs reasonably well on moderate-size projects and optimize any slow parts of the core or UI identified so far.  
    *Tasks:*
    * Profile the IDE on a sample large project (maybe a codebase with a few hundred files). Identify bottlenecks: perhaps concept graph generation is slow, or the UI rendering lags with too many nodes, or agent analysis takes too long.
    * Optimize critical paths: for example, if parsing a large project is slow, see if we can cache intermediate results or load incrementally. If the oscillator network update is taking too much CPU with many concepts, consider reducing frequency or using a more efficient algorithm/maths library (e.g., use numpy arrays for phase updates).
    * Memory usage check: ensure that concept graph storage is not ballooning. Implement pruning if needed (maybe archive parts of the concept graph not used actively, or ensure no memory leaks in our data structures).
    * Consider multi-threading or async improvements: The orchestrator and agents could potentially run in parallel threads since they often work on separate data. If Python GIL is an issue, maybe just ensure the UI thread is separate from analysis thread so UI stays responsive.
    * Optimize the UI rendering of the graph: use techniques like canvas/WebGL rendering for many nodes, or simplify the graph when off-screen.
    * Run automated tests for core algorithms to see if we can increase throughput (e.g., how many concept updates per second can we handle?). Aim for improvements and document the current capacity.  
      *Checkpoint:* By end of Day 10, TORI IDE should feel **snappier** and be able to handle larger inputs more gracefully. Perhaps we have metrics like “Initial analysis of 100-file project completes in X seconds” to gauge. Any optimization trade-offs (like reducing detail for speed) are noted and acceptable. We are now confident in the system’s performance for a demo or pilot usage.
11. **Day 11 (May 22, 2025) – TORI Chat Integration into IDE**  
    *Goal:* Embed TORI Chat capabilities into the IDE as a contextual coding assistant. This bridges the IDE and Chat products, allowing them to leverage each other.  
    *Tasks:*
    * Implement the **in-IDE chat panel** (if not done, or improve it): This panel should allow the user to ask questions about the code or get help. Connect it to the TORI Chat backend. Likely this means running the chat backend alongside the IDE backend, or as a library call to ALAN core in a different mode.
    * Context injection: Ensure that when the user asks something in the IDE’s chat panel, the Chat agent is aware of the current context (open file, selected code, etc.). Implement a mechanism to feed that context – e.g., prepend the query behind the scenes with a summary of the selected code or a reference to a concept node from the IDE.
    * Test queries: e.g., “Explain what this function does.” while a function is highlighted, or “Find any potential bugs in this file.” Verify the answers make sense and use the concept graph (the answer should ideally cite the reasoning, like “This function uses concept X which might be uninitialized, hence potential bug.”).
    * Make the chat output accessible to IDE actions: for instance, if TORI Chat in IDE suggests a code change (like “you could fix it by doing Y”), allow copying that suggestion to clipboard or even a one-click apply if feasible (this might be stretch, but at least plan for it).
    * Harmonize UI: The chat panel in Thoughtboard should have the same look/feel as the standalone chat (once designed). Also, ensure that using chat doesn’t interfere with the ongoing IDE analysis (they should share the ALAN core nicely – e.g., orchestrator should queue chat queries appropriately).  
      *Checkpoint:* By end of Day 11, a developer can seamlessly *talk to TORI from within the IDE*. This is a major milestone demonstrating the unified nature of TORI’s two halves. For example, one could highlight a piece of code and ask in the chat “How can I improve this?” and get a useful answer that draws on the code’s concepts. This dramatically shows off TORI’s capability to explain and assist in situ.
12. **Day 12 (May 23, 2025) – Enterprise Features & Hardening**  
    *Goal:* Add features particularly important for enterprise readiness of TORI IDE (and the integrated Chat), and harden the system for security and robustness.  
    *Tasks:*
    * Implement an **Audit Log** for AI suggestions and automated changes. Many enterprise users will want to track what the AI is doing. So, maintain a log (which could be a simple text file or a structured log) of every suggestion given, every auto-fix applied, every chat Q&A. Include timestamps and summary of reasoning or agent involved. This ties into transparency.
    * Add a **settings/configuration panel** for things like privacy controls (e.g., a toggle to ensure absolutely no external connection is ever made, guaranteeing all data stays local), and knobs for aggressiveness of suggestions (some companies might want only very conservative suggestions from the AI).
    * Implement authentication/permissions for multi-user scenarios (perhaps out of scope for v1.0 if single-user, but at least design how TORI would run in a shared environment or how an admin could limit certain features).
    * Security review: ensure any external inputs (like code being parsed, or chat queries) cannot lead to exploitation. For example, guard against code injection through concept pack files, or denial-of-service if the user writes something that intentionally worst-cases the analysis (we might set reasonable limits, like not analyzing files above a certain size in v1).
    * Hardening: make the system resilient to bad data. If the concept parser encounters a syntax it can’t handle (e.g., an unsupported language feature), it should fail gracefully (perhaps create a generic node or skip, rather than crash). Similarly for chat – if something goes wrong (like the reasoning times out), the UI should handle it (maybe respond with “I’m not sure about that” rather than hang).  
      *Checkpoint:* End of Day 12 should see TORI IDE approach a **production-ready posture** for enterprise trials. We have audit logs, some configuration ability, and the system has been combed for major security holes. While formal security testing is ongoing, we addressed obvious issues. This boosts confidence for enterprise demo on Day 16 and beyond.
13. **Day 13 (May 24, 2025) – TORI Chat Standalone Application**  
    *Goal:* Prepare TORI Chat as a separate deliverable (especially for enterprise use outside the IDE), ensuring it works as a standalone conversational assistant with access to “concept packs” and knowledge base.  
    *Tasks:*
    * Set up the **standalone Chat app** environment. If it shares a lot with the IDE, perhaps it’s the same backend but different front-end. Ensure we can run TORI Chat without the IDE UI – e.g., via a command-line interface or a basic web UI (which we have from prototype).
    * Expand the chat UI (web/Electron) to support multi-turn conversations nicely. Implement displaying the conversation history, and allow resetting context, etc. This UI can be simple but should be clean and user-friendly.
    * Test loading a domain **Concept Pack** in standalone chat. For instance, load a sample medical ontology and ask TORI Chat questions in that domain (“What is the procedure for XYZ?”) to see that it uses the injected knowledge. Fix any issues in concept pack integration (like concept collisions or memory usage).
    * Persona support: Implement a way to choose a persona or role for the chatbot. Maybe in the UI a dropdown “Emulate: [General Assistant, Python Guru, DevOps Coach, *Custom*]”. This will prepend an appropriate system message to steer the style. The Attachment agent’s work ties in here if, say, an “Empathetic listener” persona is chosen, the responses shift tone.
    * Ensure that the Chat system respects the privacy toggles (from Day 12 work) – i.e., verify that it indeed does not call out to any cloud API (we aren’t using any by design, but double-check things like spaCy or other libraries aren’t pulling data online unexpectedly).  
      *Checkpoint:* By end of Day 13, **TORI Chat standalone** should be ready for basic use. One should be able to launch the chat app, perhaps select a knowledge pack, and have a coherent conversation. While the IDE and Chat share a core, this independent packaging is important for demonstration to non-developer stakeholders and for enterprise environments where the chat might be deployed as a knowledge assistant on its own.
14. **Day 14 (May 25, 2025) – Multi-Agent Enhancements & Chat Agents**  
    *Goal:* Improve the multi-agent system in the Chat context, and ensure orchestrator synergy between Chat and IDE agents when needed. Use this day also as a secondary buffer/review point for the second week.  
    *Tasks:*
    * Turn focus to **Chat agents**: ensure the Knowledge Retrieval Agent and Reasoning Agent are working together properly. Perhaps implement a more advanced retrieval (like searching not just exact concepts but related ones, maybe using the embeddings to fetch similar concepts when direct answers aren’t found).
    * If the Attachment Agent (emotional reasoning) is to have any effect, implement simple rules: e.g., if user sentiment (we can guess from text, or even just presence of “!” or words like “frustrated”) is negative, the Chat agent will respond with a more apologetic tone. This could be done by having the orchestrator adjust the persona or insert a directive like “The user seems upset, respond supportively.”
    * **MPorter in Chat:** Possibly integrate the strategy agent to manage long conversations. For instance, if the user’s query is very complex, MPorter could break it into sub-questions for the other agents. We might simulate this by handling compound questions or having the chat orchestrator call the Knowledge agent multiple times for parts of the question.
    * Conflict handling in Chat: If two knowledge sources conflict (say concept pack vs code context), orchestrator should detect and either ask user for clarification or choose one based on priority (perhaps code context overrides general knowledge, etc.). Implement a basic policy and test it (“What is X?” where X is defined differently in user’s project vs general knowledge).
    * Use remaining time to fix any critical issues or do another mini-test of full system (both IDE and Chat) ahead of final stretch.  
      *Checkpoint:* By end of Day 14, the multi-agent system for Chat should be more robust and intelligent. TORI Chat can incorporate multiple perspectives or specialized sub-agents (e.g., retrieval vs reasoning) in one conversation without confusing the userfile-bnjjcfgff6iusnbzdwqayo. We should also now have both IDE and Chat components largely feature-complete, setting the stage for final refinements and testing in the last week.
15. **Day 15 (May 26, 2025) – Documentation, Tutorials & UX Polish**  
    *Goal:* Prepare user-facing documentation and improve the user experience with small tweaks and fixes. Make TORI feel more polished.  
    *Tasks:*
    * Write a **quickstart guide** and in-app tutorial hints. E.g., when TORI IDE first launches, show a “Welcome to TORI” message that highlights key areas (concept explorer, chat panel, etc.). Also prepare a README or user manual covering installation, basic usage, and explanation of TORI’s key concepts (for early adopters trying it out).
    * Ensure all UI text is clear and user-friendly. Replace any placeholder labels or cryptic terms. For example, if a suggestion says “Phase conflict in module X,” perhaps rephrase to “Inconsistency detected between module X and Y (they may need synchronization)” – more understandable language.
    * Add tooltips or help icons where needed, especially for novel features. If a user hovers over the oscillator dashboard, a tooltip could explain “This visualization shows the synchronization of key components – when waves line up, components are in sync.”
    * Finalize persona and branding in UI: name, logos, color scheme consistent. Possibly incorporate a small TORI logo or avatar in the chat interface for personality.
    * If time, record or script some example **use-case scenarios** to validate workflow: e.g., “User story: debugging a problem with TORI’s help” – walk through it and ensure the UI/agents support it smoothly. Refine any steps that felt clunky.  
      *Checkpoint:* By end of Day 15, TORI should *feel* more like a polished product rather than a research demo. New users should have some guidance (docs/tutorial) and the overall UX should be refined. We want to be able to hand TORI to a friendly tester and have them understand what to do without devs explaining everything in person.
16. **Day 16 (May 27, 2025) – Internal Testing & Bug Bash**  
    *Goal:* Conduct intensive testing of all features (IDE and Chat), fix bugs, and ensure quality. Essentially a “bug bash” day.  
    *Tasks:*
    * Have team members (and possibly a small group of internal users) use TORI for real tasks for a few hours. Collect their feedback and any issues encountered.
    * Triage bugs: categorize into must-fix for launch vs can-fix-later. Focus on must-fix now.
    * Fix critical bugs across the board: crashes, incorrect analyses, UI misalignments, etc. Pay special attention to anything that could lead to wrong behavior (e.g., an agent suggesting something dangerous in code).
    * Cross-platform test if possible: run on Windows, Mac, Linux to ensure no environment-specific bugs (since local-first means environment differences matter).
    * Test performance in some edge cases (very large file, extremely long chat conversation) to make sure nothing catastrophic occurs (if it does, perhaps implement safety cut-offs, like “Results truncated…”).  
      *Checkpoint:* By end of Day 16, the bug count should be vastly reduced. Ideally, no showstopper bugs remain. TORI is now in a *release candidate* state for both IDE and Chat. We should have a list of known issues that are either minor or will be addressed with lower priority.
17. **Day 17 (May 28, 2025) – Deployment Preparation & Packaging**  
    *Goal:* Prepare TORI for distribution: installer, packaging, environment setup scripts, etc., and ensure it can be easily deployed by users or on servers.  
    *Tasks:*
    * Create an **installer or package** for TORI IDE (Thoughtboard). Possibly using PyInstaller for a one-file executable or creating an Electron app package. Ensure that the ALAN core (Python backend) starts up with the UI seamlessly.
    * Package TORI Chat standalone similarly, if it will be delivered separately (maybe a Docker image or a simple executable that launches the chat server and web UI).
    * Write deployment docs (for IT teams): e.g., how to install on an offline machine, required system resources, how to configure on-prem.
    * Test installation on a fresh system: does it run out of the box? Catch any missing dependencies or config steps and adjust the installer accordingly.
    * If licensing or activation is needed (for enterprise, maybe not yet), at least design where that would hook in. Possibly just stub an “enter license” dialog if appropriate (though might not be needed for v1 tech preview).
    * Ensure logging (from Day 12) goes to proper files in the installed locations and not to console, etc., as expected.  
      *Checkpoint:* By end of Day 17, we should have **installable builds** of TORI IDE and TORI Chat. This means we could send a file (or Docker image) to someone and they could get TORI running without our direct involvement. It’s a key step towards any sort of user testing or pilot program.
18. **Day 18 (May 29, 2025) – Documentation & Final Touches**  
    *Goal:* Finish comprehensive documentation (technical and user), and add any final touches or small features that were deferred but are quick wins.  
    *Tasks:*
    * Complete the **user manual** covering all features of v1.0. Include screenshots of the UI, examples of how to interpret the outputs (like an example of the oscillator dashboard and what it meant in a scenario), and FAQs for common questions.
    * Write a section in docs for “Under the hood: How TORI works” for the curious user or stakeholder, explaining concepts like the concept graph and phase reasoning in approachable terms (this leverages our deep knowledge but presents it simply).
    * Ensure the in-app help links (if any) point to the correct documentation or have tooltips (from Day 15).
    * Implement any **small deferred enhancements** if time permits. For example, maybe adding a keyboard shortcut to open the concept explorer, or a command palette entry for “Ask TORI” to quickly query something without clicking the chat panel. These little touches improve UX if easily done.
    * Final branding pass: ensure the app names, version numbers, and about dialogs are correct (“TORI IDE v1.0 (ALAN 2.x core)” etc.).  
      *Checkpoint:* By end of Day 18, documentation should be **complete and polished**. All those using TORI (developers, early adopters) have the materials to understand and troubleshoot it. The product should have a fit-and-finish that we’re proud to present. We should feel confident that no major feature is undocumented or totally unknown to the user.
19. **Day 19 (May 30, 2025) – Final Testing & UX Review**  
    *Goal:* Do a final end-to-end test pass as if we were the end-user. Refine any last UX issues and make sure we’re truly ready to release.  
    *Tasks:*
    * Perform a scenario-based test: e.g., “Use TORI IDE to improve a small project from scratch,” and “Use TORI Chat to learn about a domain.” Follow these scenarios step by step, noting any awkwardness or minor bugs.
    * Fix *minor bugs* that are quick (typos, alignment issues, log verbosity, etc.). Anything more than minor should already have been caught, but if something appears now that is critical, we still have a bit of buffer to address it.
    * UI/UX refinement: tweak colors, spacing, icons to ensure the interface looks clean and professional. Make sure the visual theme is consistent (fonts, sizes).
    * Confirm all *integrations* work one more time: The chat panel in IDE still works in the packaged build, the concept pack loading in Chat works, etc., in the final packaged environment.
    * By mid-day, *freeze the code*. Enter code freeze for release candidate except for emergency fixes. From this point, focus on stability.
    * If time, do a dry-run of a **demo or presentation**. Usually, we’d want to simulate what we will show stakeholders or potential users and ensure TORI can be showcased without hiccups. This doubles as a final test.  
      *Checkpoint:* End of Day 19, we have what we consider the **release candidate build** of TORI IDE and TORI Chat v1.0. The team should feel confident in doing a live demo to stakeholders without making excuses for rough edges. We likely will tag this version in our repo as v1.0-rc.
20. **Day 20 (May 31, 2025) – Buffer, Polishing & Pre-Launch Prep**  
    *Goal:* Use this day as a buffer for any last-minute critical issues and prepare for launch activities (like presentation, marketing collaterals).  
    *Tasks:*
    * If any critical bug was found late Day 19 or by additional testers (perhaps someone from another team trying it on Day 19 evening), fix it with priority.
    * Finalize the **presentation/demo script** for the launch meeting or recording. Ensure all materials (slides, if any, or sample projects to demo on) are ready.
    * Create some **marketing content** if applicable at this stage: perhaps a one-pager summary of TORI’s value prop to accompany the release, or internal announcement text. This may involve working with a product or marketing person, but the technical team provides inputs (features, differentiators).
    * Double-check that all **SWOT points** have been addressed in some form in the product or documentation (e.g., weakness of limited language support is documented and openly acknowledged with plans, etc., so it doesn’t catch anyone by surprise).
    * Rest (if everything is truly done). The team has pushed hard; if all is in order, a lighter day to recuperate ensures we approach launch with clear minds.  
      *Checkpoint:* End of Day 20 should have zero critical open issues. We are essentially ready to launch the product the next day. All that remains is the actual launch event or final sign-off.
21. **Day 21 (June 1, 2025) – Sprint Completion & Launch Readiness**  
    *Goal:* Final day to wrap up the sprint, do any last reviews, and *prepare for the launch/presentation*. If possible, also allow the team a brief respite before the big day.  
    *Tasks:*
    * **Final sanity test:** Run through the core functionality one more time quickly in the morning. If anything unexpected occurs, decide if it must be fixed now or can be deferred (ideally deferred unless it’s truly showstopping, as we shouldn’t be making code changes on launch day).
    * Package the final release builds, double-check their integrity (hashes, can be installed fresh, etc.).
    * Ensure all documentation and materials are accessible (upload to internal site or include in the installer package as appropriate).
    * Team meeting to recap the sprint accomplishments, and discuss support plan post-launch (e.g., how to handle user feedback, quick patches if needed).
    * If this is an internal launch, coordinate with any stakeholders on the launch event details. If external, perhaps push the repository to a private share or similar.
    * **Rest and mental preparation:** Encourage the team to take it easy in the latter half of the day if everything is done. We want everyone fresh for the launch or next phase.  
      *Checkpoint:* By end of Day 21, the 21-day sprint is **successfully completed**. TORI IDE and ALAN 2.x (with TORI Chat) are ready for deployment/presentation. The team can confidently say we have met the goals of the sprint and are prepared to introduce TORI to its initial users.

**TORI Chat – 21-Day Sprint Plan (May 12 – June 1, 2025)**

*(The TORI Chat development ran in parallel to the IDE sprint, with some shared efforts on core integration, multi-agent system, etc. The numbering here mirrors the days, but many Chat tasks were done concurrently by a sub-team focusing on the chat interface and capabilities.)*

**Sprint Goal:** Evolve TORI Chat from a rudimentary prototype into a robust conversational assistant with enterprise-ready features and seamless integration with the TORI cognitive core. Ensure TORI Chat can operate both embedded in TORI IDE and as a standalone product.

1. **Day 1 (May 12, 2025) – Chat Sprint Kickoff & Environment Setup**  
   *Goal:* Align on TORI Chat objectives and set up the development environment for the chat application.  
   *Tasks:*
   * Kickoff meeting in sync with the IDE team, focusing on Chat-specific deliverables (like persona handling, concept packs, etc.). Ensure cross-team understanding of what components are shared.
   * Set up the chat server framework (likely extending the same ALAN backend). If using a web UI for chat, set up the development environment for that (Node/Electron or pure web).
   * Ensure the concept network and core from ALAN can be accessed in a stateless way (since chat queries might be one-off calls to the core). Prepare scaffolding for maintaining chat conversation state (likely a context object storing recent conversation or a pointer to relevant concept subgraph).
   * Merge any baseline code for chat from prototypes into main, analogous to the IDE merging step.  
     *Checkpoint:* Chat team has a running development instance of TORI Chat (even if it’s just echoing or a basic template) and is in sync with the overall project plan. Environment issues resolved.
2. **Day 2 (May 13, 2025) – Core Alignment for Chat (Oscillators & Memory)**  
   *Goal:* Tie the updated core (oscillator sync, etc.) into the chat flow. Ensure that chat inputs create proper concept activations and that the conversation context is maintained in the core.  
   *Tasks:*
   * Modify the NLP pipeline so

**Comprehensive Master Plan and Strategic Analysis – TORI IDE & ALAN 2.x Cognitive OS**

**Executive Summary**

**TORI** is a next-generation *cognitive operating system* comprising two flagship products: **TORI IDE** (an AI-augmented Integrated Development Environment) and **TORI Chat** (an intelligent chat assistant, including an enterprise-grade variant). This master guide serves as a comprehensive report, technical specification, development roadmap, and marketing playbook for the TORI suite. It encapsulates the visionary paradigm behind TORI, the current architecture and module breakdown, a 21-day sprint plan for rapid development, SWOT analyses, and go-to-market strategies for each product.

* **Vision:** TORI embodies a paradigm shift from traditional AI tools. Rather than relying on rote token prediction, TORI’s core is grounded in **concepts over tokens** and \**memory over stateless prediction*[medium.com](https://medium.com/syncedreview/from-token-to-conceptual-the-rise-of-metas-large-concept-models-in-multilingual-ai-b32acbfeb792#:~:text=generalization%20across%20languages%2C%20outperforming%20existing,LLMs%20of%20comparable%20size)】. The vision is an “AI-native” development experience in which the system truly understands code and conversations at a conceptual level, enabling unprecedented levels of assistance and insight. Developers move from using tools to collaborating with a *cognitive partner*.
* **Architecture:** At TORI’s heart is the **ALAN 2.x cognitive engine**, a spectral-phase reasoning core that maintains a rich **Large Concept Network (LCN)** of knowledge. This core powers both the IDE and Chat, providing deterministic, auditable reasoning steps instead of opaque neural guesswor[ibm.com](https://www.ibm.com/think/topics/black-box-ai#:~:text=White%20box%20AI%2C%20also%20called,Users)】. Surrounding the core are modular subsystems for language processing, user interaction, and multi-agent orchestration. In the IDE, this means deeply understanding code structure and evolution; in Chat, it means maintaining contextual, persona-driven dialogue. Both products share a **local-first, privacy-conscious design** that can operate on-premises without cloud dependence.
* **Current Status:** The TORI codebase has been through early prototyping phases. Key modules for concept parsing, memory graph management, and a basic front-end exist, though some components remain mock or placeholder implementations pending integration of the full ALAN engine. For example, a document ingestion pipeline successfully indexes PDFs into the concept network, but the **phase oscillator network** underpinning dynamic reasoning is only *partially implemented*. The current TORI IDE UI is functional as a prototype “**Thoughtboard**” dynamic code canvas, albeit without full polish or all expected IDE features. TORI Chat’s core reasoning back-end is in place, but its user-facing chat interface and multi-agent persona system are at a rudimentary stage. These gaps are identified in the module-by-module analysis, and the plan is to replace placeholder code with production-ready logic during the upcoming development sprints. *Before sprinting on new features, completing these core components (oscillator sync, spectral engine, ELFIN parser, orchestrator) is imperative, to ensure the foundation is solid.*
* **Roadmap:** A detailed **21-day sprint plan** for each product outlines the path from the current state to a viable v1.0 release. Each sprint plan is organized into daily milestones with one rest/review day built-in per week. (For reference, **Day 1 corresponds to May 12, 2025**.) The TORI IDE plan focuses on integrating the ALAN core with the IDE front-end, implementing conceptual debugging tools, and eliminating any “mock” data paths. The TORI Chat plan emphasizes developing a robust conversational interface, domain-specific **“concept pack”** integration, and enterprise features like audit logs and on-prem deployment support. Both plans culminate in a synchronized launch, ensuring the IDE and Chat leverage each other (e.g. the IDE generating chat-ready knowledge graphs, and the Chat agent assisting within the IDE).
* **SWOT Highlights:** TORI’s **Strengths** lie in its unprecedented approach: a phase-synchronized reasoning core that yields deterministic, explainable insights (a “glass-box” AI approach[ibm.com](https://www.ibm.com/think/topics/black-box-ai#:~:text=White%20box%20AI%2C%20also%20called,Users)】, and a local-first architecture that sidesteps the compliance and cost issues of cloud-only AI solutions. **Weaknesses** relate to typical early-stage challenges: limited UI polish compared to incumbent tools, narrow language support at launch (only Python and the custom ELFIN DSL are first-class), and the need for extensive real-world validation of its novel techniques. **Opportunities** include growing demand for on-premises AI solutions (for privacy and control), and synergy between development and conversational AI (TORI uniquely bridges these). **Threats** include rapid moves by major IDE vendors to integrate similar AI capabilities, and skepticism toward unproven AI paradigms in conservative industries. Our strategy and roadmap specifically address these points through aggressive development, transparency, and engagement with early adopters.

**Vision and Guiding Principles**

**TORI’s vision** is to create a *cognitive operating system* that transforms how we interact with software development and knowledge work. This vision is grounded in two core ideas: focusing on **conceptual understanding** rather than surface-level output, and building **persistent knowledge** rather than stateless interactions. Traditional AI coding assistants operate at the level of syntax and tokens, predicting text without true understanding. TORI takes a different path – it builds and manipulates rich *conceptual models*. In the IDE, this means understanding the abstract architecture and intent behind code; in Chat, it means grasping the semantics and context of a conversation, not just producing plausible sentences. By elevating to the conceptual level, TORI provides insights and assistance that feel genuinely intelligent and context-aware, rather than generative but shallow. As one early reviewer noted, TORI represents not just a toolset, but *a paradigm shift in how we think about code, conversations, and AI assistance*.

In addition, instead of treating each user query or coding session as an isolated event, TORI maintains a **persistent memory** of interactions and learned concepts. This is not simply a chat history or cache, but a structured memory in the form of the LCN, phase states, and accumulated patterns. TORI remembers *why* a piece of code was written or *how* a conclusion in chat was reached, enabling it to explain its reasoning and build on past knowledge. This contrasts with stateless large language model tools that have no inherent memory of past interactions. TORI’s approach yields an AI that can truly learn and adapt over time – accumulating **wisdom** rather than resetting after each prompt.

From this vision, we derive a single unifying goal: **TORI is not just a development tool or a chatbot; it is a thinking partner that grows with the user.** The IDE is envisioned as a *“Conceptual Canvas”* where code and design interplay dynamically with the AI’s understanding. The Chat is envisioned as a multi-mind conversational agent that can reason through complex problems with traceable logic. Together, they form a unified cognitive workbench.

Several **guiding principles** direct our design and implementation:

* **Human-Centric Design:** TORI keeps the human user in control and informed at all times. We adhere to the principle of *“human in the loop”*. Features like explainable suggestions, the ability to inspect the concept graph, and toggles for AI assistance are built-in. The user should feel TORI is *amplifying* their abilities, not automating tasks away without consent or clarity.
* **Transparency and Explainability:** Every AI-driven action by TORI is explainable via its concept graph or a rationale. For example, if TORI IDE suggests a code refactor, it can point to the specific concepts and oscillator signals that led to that suggestion (e.g., a function concept that is out-of-phase with related functions, indicating a design inconsistency). Similarly, TORI Chat doesn’t just answer questions – it can provide a brief explanation or source (e.g., which internal knowledge or which agent’s logic contributed to the answer). This fosters trust and positions TORI as a “glass-box” AI, in contrast to opaque *black-box* model[ibm.com](https://www.ibm.com/think/topics/black-box-ai#:~:text=White%20box%20AI%2C%20also%20called,Users)】.
* **Deterministic Core, Creative Edges:** TORI’s architecture emphasizes *deterministic reasoning* in its core (the spectral-phase logic, rule-based multi-agent decisions, etc.) to ensure reproducibility and auditability. However, we allow for stochastic creativity at the periphery. For instance, a language model may be used to generate natural language summaries or code comments under TORI’s guidance. The rule is to use randomness only where it’s safe and adds value (e.g., phrasing a response kindly), but never in the core understanding of the code or conversation. This yields a system that is both **reliable** and **inventive** – core logic is stable and verifiable, while user-facing interactions can still have the flexibility and richness of generative AI.
* **Local-First and Privacy-Preserving:** TORI is built with a *local-first* mindset. Users retain complete ownership of their code, data, and the AI’s knowledge about them. Both TORI IDE and TORI Chat can run fully locally or on a private network. No data leaves the user’s machine unless explicitly configured to (for example, if the user chooses to enable cloud connectivity or external APIs). This addresses enterprises’ needs to keep proprietary code and discussions in-house. It also has a technical upside: operating locally yields lower latency and tighter integration with the user’s environment (files, processes, dev tools) since TORI can directly interface with them without network overhead.
* **Modularity and Extensibility:** TORI’s cognitive OS is modular by design. Each subsystem – parsing, concept storage, reasoning agents, UI – is a separate module with well-defined interfaces. This allows future improvements or swaps (e.g., plugging in a new parser for a different language, or adding a specialized agent) without overhauling the entire system. It also means parts of TORI can be extended by third-parties; for instance, developers might create new ALAN agents or integrate TORI’s concept graph with external tools via APIs. This modularity also aids maintainability: given TORI’s broad scope, we must be able to update one component (say, the NLP pipeline) independently of others.
* **Consistency with Developer Workflow:** Especially for TORI IDE, we design to *“meet developers where they are.”* This means TORI’s enhancements are built into familiar workflows. The IDE still allows editing code as text, using Git for version control, running tests, etc. – but TORI augments these activities with cognitive insight. For example, a developer can code normally and optionally inspect TORI’s concept graph for deeper understanding, or receive proactive warnings about design issues. TORI Chat is designed to integrate into the developer’s environment too (for instance, accessible as an in-IDE chat panel or via tools like Slack through integrations) so that conversing with TORI feels like a natural extension of their workflow. The guiding motto is *“low floor, high ceiling, wide walls”* – easy to get started, immense power as you explore, and broadly applicable to many tasks.
* **Inspiration and Playfulness:** While TORI is grounded in rigorous concepts from cognitive science and spectral theory, the project embraces creativity and approachability. This appears in our internal code names (e.g., the oscillator core nicknamed “**Banksy**” for its artful sync patterns, conceptual caches called “cartridges,” etc.) and in the product’s personality. TORI aims to feel engaging, even fun – we believe a degree of playfulness encourages exploration and learning. For example, TORI might include easter eggs or a witty persona remark (tastefully and only when appropriate) to remind users this AI is *approachable*. This helps differentiate TORI: it’s advanced technology with a human touch and a bit of charm, not a cold enterprise robot.

Balancing these principles ensures that as we build TORI, we maintain both **intellectual rigor** (truly innovating and solving hard problems in AI reasoning) and **user empathy** (crafting an experience that empowers and delights). The following sections translate this vision and these principles into TORI’s concrete architecture and development plan.

**Cognitive OS Architecture Overview**

TORI’s architecture is a modular, layered cognitive system designed to support both the IDE and Chat applications through a shared intelligent core. At a high level, the architecture consists of four integrated subsystems:

* **ALAN 2.x Cognitive Core:** The “brain” of TORI – responsible for conceptual understanding, inference, and maintaining the knowledge network.
* **Language/Domain Integration Layer:** Bridges the core with domain-specific inputs (code, documents, natural language) and outputs, translating between raw data (files, text) and TORI’s concept representations.
* **User Interface & Tools:** The front-end components through which users interact with TORI (the Thoughtboard IDE UI, debugging and visualization tools, chat interfaces, etc.).
* **Multi-Agent Orchestration:** The layer managing a collection of specialized AI agents and coordinating their efforts via the shared core (the “mind society” within TORI).

Each of these is detailed below, followed by how they collaborate in the TORI IDE and TORI Chat contexts.

**ALAN 2.x Cognitive Core (Conceptual Engine)**

At the heart of TORI lies the **ALAN 2.x cognitive core**, which maintains a rich, interconnected representation of knowledge called the **Large Concept Network (LCN)**. This core can be thought of as TORI’s brain: it encodes concepts (from code constructs to conversation topics) as nodes in a graph and relationships between those concepts as edges. Beyond static relationships, ALAN 2.x introduces a novel *spectral-phase inference mechanism* that enables dynamic reasoning over time.

In essence, the core treats reasoning as a live process of **synchronization and resonance** among concepts. Every concept node in the LCN is accompanied by an internal state, including a **phase oscillator** representing its current “state of mind” or context. When the system processes new information or code changes, these oscillators are perturbed and evolve. The core monitors the collective behavior of all concept oscillators to identify emerging patterns of coherence or conflict. Intuitively, if two concepts should logically agree (say two functions that need to work in concert), the system expects their oscillators to synchronize. If they drift apart (out of phase), that signals a potential inconsistency or conflict.

This approach draws inspiration from **neural oscillation theories** in cognitive science – where synchronized neural firing is thought to underlie unified perceptions and thought[neurosciencenews.com](https://neurosciencenews.com/brain-rhythm-cognition-25941/#:~:text=It%20could%20be%20very%20informative,%E2%80%9D)】. By analogy, TORI’s synchronized concept oscillators imply a coherent understanding or stable solution, whereas desynchronized oscillators flag issues requiring attention. For example, if a function’s implementation is inconsistent with its specification, the concept representing the implementation may oscillate out-of-sync with the specification concept, alerting TORI that something is off.

**Phase-Coupled Oscillator Matrix (“Banksy Core”):** The engine that manages these oscillators is nicknamed *Banksy* for its focus on rhythmic, artful coordination. Technically, it’s a matrix of coupled oscillators associated with the concept graph. In earlier prototypes we considered using a standard Kuramoto model for synchronizing these oscillators, but we have since moved to a more advanced **ψ-based coupling approach** that leverages spectral analysis for stability and alignment. In this approach, each concept oscillator’s update is influenced not just by direct neighbors (related concepts), but by a global perspective from the system’s spectral mode analysis (explained shortly). The goal is to achieve robust consensus where appropriate, and deliberate divergence where a conflict genuinely exists (as opposed to noise causing false alarms).

Each concept oscillator has a phase (and possibly amplitude) that can be interpreted: two concepts in phase indicate they are in harmony or agreement; out-of-phase might indicate discrepancy or different “contexts.” By adjusting coupling strengths and rules (the core might couple oscillators of related concepts strongly, and unrelated ones weakly or not at all), TORI dynamically enforces a consistency check across the entire knowledge network. If concept A (e.g., a variable’s expected type) and concept B (the variable’s actual usage) are linked, the oscillator engine will try to keep A and B in sync. A sustained phase difference would mean the engine cannot reconcile them – prompting, for example, a warning to the user about a type mismatch. Through this mechanism, TORI’s core provides a continuous, interpretable measure of consistency across the system.

Importantly, the simplistic Kuramoto coupling has been replaced with a more *intelligent synchronization strategy*. We incorporate insights from **Koopman operator theory** and control theory so that the oscillators not only sync or desync, but do so in a stable manner. The system computes collective properties (like an order parameter indicating overall sync) and uses feedback to ensure stability (preventing chaotic swings). It also means we can identify clusters of synchronization – perhaps a subset of concepts oscillate together (indicating a coherent subsystem or topic), separate from another cluster. This clustering can reveal modular structure in code or conversation.

*(Status:* As of now, the phase-coupled oscillator network is partially implemented. The basic data structures exist and oscillators update in a rudimentary way, but the full ψ-based synchronization algorithm is being finalized. Completing this and fully integrating it into the runtime loop is a top priority in the development plan.)

**Koopman Morphic Engine (Spectral Analyzer):** Complementing the time-domain oscillator view, ALAN 2.x employs a **Koopman operator-based spectral analyzer** to study the evolution of the concept network state in a higher-dimensional space. The **Koopman operator** is a theoretical construct that allows us to treat a nonlinear dynamical system (like our knowledge state updates) via an infinite-dimensional linear operator acting on observation function[mdpi.com](https://www.mdpi.com/2227-7390/9/19/2495#:~:text=The%20Koopman%20operator%20theory%20is,are%20amenable%20to%20systematic%20analysis)】. In simpler terms, we lift the system’s state into a space of features (potentially complex eigenfunctions denoted ψ) where the dynamics can be approximated as linear. By doing so, we can apply linear spectral analysis: find eigenvalues and eigenvectors (modes) that describe recurring patterns or steady states in the reasoning process.

Practically, the Koopman engine monitors changes in the concept graph (like concept activation levels, oscillator phases) over time or iterations. It then tries to identify **spectral modes** – for instance, an eigenmode might correspond to a particular pattern of concept activation that repeats or a combination of concepts that tends to increase or decay together. The eigenvalues associated with these modes can tell us about stability: eigenvalues on the unit circle imply a repeating cycle, eigenvalues less than 1 in magnitude indicate a decaying mode (converging to stable state), and greater than 1 would indicate a growing instability. This analysis provides TORI with a predictive foresight. For example, if a particular conceptual inconsistency tends to amplify each iteration (an unstable mode), the system can flag it early as something that will likely cause problems if not addressed (like a bug that will cascade). If a mode corresponds to a settled understanding (convergence), TORI knows those aspects of the problem are resolved and stable.

In essence, the Koopman engine allows TORI to reason not just in the now, but about the *trajectory* of the reasoning process. It can answer questions like: *Is the system of concepts converging to a coherent understanding?* or *Are there oscillatory disagreements that persist?* It’s analogous to how an experienced engineer might sense if a project’s design discussions are converging or endlessly oscillating between options. Mathematically, it provides a global, linearized view of the nonlinear interactions happening in the cor[mdpi.com](https://www.mdpi.com/2227-7390/9/19/2495#:~:text=The%20Koopman%20operator%20theory%20is,are%20amenable%20to%20systematic%20analysis)】.

To ensure reasoning remains controlled and safe, we incorporate **Lyapunov stability measures** into this analysis. A Lyapunov function (think of it as an energy or potential function for the system) is used to prove stability of an equilibrium. In TORI’s context, we design Lyapunov-like metrics to ensure our reasoning process tends toward a stable equilibrium (a consistent state) and not diverge. If the system’s computed Lyapunov function is decreasing over reasoning iterations, it indicates the system is *stable in the sense of Lyapunov*, meaning small perturbations (like minor code edits or rephrasing of a question) will dampen out and not blow u[ocw.mit.edu](https://ocw.mit.edu/courses/16-30-feedback-control-systems-fall-2010/463e9559c6ef70f13ea51c3464bdc9c1_MIT16_30F10_lec22.pdf#:~:text=%E2%80%A2%20Stable%20in%20the%20sense,%E2%80%9D%20November%2027%2C%202010)】. The orchestrator can use these metrics to decide when to stop iterating on an internal problem (e.g., the agents have reached a consensus when the system is stable).

*(Status:* A scaffold of the spectral analysis exists – it can take snapshots of concept states and compute a basic decomposition (e.g., via an SVD or FFT). Currently it returns dummy data for testing the pipeline. Implementing the full Koopman-based analysis and integrating real stability checks is in progress. It’s slated for completion early in the development sprint, after which the core will regularly generate these spectral insights to guide reasoning.)

**Static Knowledge Base & Symbolic Reasoning:** Alongside the dynamic oscillator and spectral components, the ALAN core also maintains a static knowledge base and performs more traditional symbolic reasoning. For example, TORI stores facts and rules (like “if concept X implies concept Y, and X is true, then Y is true”) within the LCN. It can run logical inference on this static graph as needed. Moreover, TORI uses *embedding-based similarity* to relate concepts: each concept may have a vector embedding (learned from code context or language data) so that semantically similar concepts can be identified via cosine similarity. This is useful for suggestions (like finding analogies in code or clarifying ambiguous questions in chat). These static or precomputed relationships act as a baseline layer of “common sense” and domain knowledge, upon which the dynamic reasoning operates.

**Determinism and Auditability:** A key design choice is that, given the same inputs and initial state, the ALAN core’s reasoning process is *deterministic*. That means TORI will produce the same sequence of internal states and same outcomes every time for reproducibility. The apparent randomness or creativity in output (like phrasing) is separated out (e.g., handled by a language model at the very end, if at all) and does not affect the underlying reasoning trace. We log the sequence of reasoning steps (concept activations, agent decisions, etc.) as an **audit trail**. This allows users (especially enterprise users) to inspect *why* TORI arrived at a suggestion – they can trace through a sequence like: *“Function A’s concept was out-of-phase -> Refactoring agent noted duplication with Function B -> Suggested to unify them.”* This level of introspection is rarely possible with traditional AI assistants and is a differentiator for TORI’s cognitive OS approach.

**Language & Domain Integration Layer**

The Language & Domain Integration Layer is responsible for translating real-world inputs into TORI’s conceptual form and vice versa. It ensures that whether the input is source code, a natural language sentence, or a formal specification, TORI can incorporate it into the LCN; and when TORI produces an output (like an answer or code change), it can be rendered in a human-usable format.

**Programming Language Frontends (for TORI IDE):** TORI IDE supports multiple programming languages by using frontends that parse code into the LCN. For each supported programming language, we employ either a custom parser or an existing parser/AST generator to break code down into its structure. For example, for Python, TORI uses the Python AST module to get the syntax tree, then traverses it to create concept nodes for entities like functions, classes, variables, as well as relationships (e.g., “function *foo* calls function *bar*” becomes a link between the concept nodes for *foo* and *bar*). We attach additional semantic information when available: docstrings become documentation concept nodes, types (if inferred or declared) become type-concept links, etc. Our custom DSL ELFIN (detailed below) has its own parser that directly produces a rich concept graph (since ELFIN is designed to express high-level concepts natively).

For other languages like C/C++ or JavaScript/TypeScript, we integrate with established parsing tools (e.g., tree-sitter or language server protocols) to get an AST or symbol table, then map those into TORI’s concepts. In these cases, because we may not capture every nuance, the integration might be a bit shallower (for instance, TORI might not fully understand C++ template metaprogramming yet, but it will still record the relations it can, like function calls and inheritance). Our design makes Python and ELFIN *first-class* (with the richest understanding), while still supporting other popular languages sufficiently to be useful. Over time, we plan to extend first-class support to more languages.

Whenever code is edited or a new file is added, this layer updates the LCN incrementally. For example, adding a new function in code creates a new concept node and appropriate links without rebuilding the entire graph from scratch. This integration is efficient and continuous, feeding changes into the ALAN core’s reasoning loop (which then perhaps triggers re-synchronization or agents to check the implications of that new function).

**Domain Knowledge Ingestion (for TORI Chat and Enterprise):** TORI’s usefulness amplifies when it’s informed about the user’s domain. The integration layer can ingest structured knowledge bases such as **ontologies**, **taxonomies**, or corporate knowledge graphs and integrate them as part of the LCN. We call these packaged knowledge sets **Concept Packs**. For instance, an enterprise could load a “Financial Regulations Pack” that contains concepts like regulations, compliance procedures, definitions of banking terms, etc., all connected in a graph. TORI merges this with its own concept network (or keeps it in a linked subgraph) so that both TORI IDE and Chat can leverage that domain knowledge.

In the IDE, this could mean recognizing that a certain variable or function relates to a concept in the ontology (e.g., variable customerAccount is linked to the “Customer Account” concept in a finance ontology). In Chat, it means if the user asks “What is the KYC process for a new client?”, TORI has the concept of “KYC (Know Your Customer)” in its graph and can reason with it directly, producing a knowledgeable answer. The ingestion process typically involves reading a structured file (like OWL or CSV data for ontologies) and creating concepts and relations accordingly. We preserve any hierarchy (e.g., is-a relationships become concept edges) and key attributes (which might become metadata on concept nodes).

For unstructured but relevant data (like a company’s internal PDFs or manuals), TORI can use an offline process to index those into the concept graph as well (essentially performing entity extraction and linking to build a mini knowledge graph of the documents). This was prototyped with the PDF ingestion pipeline mentioned in the current status. TORI Chat can then retrieve and cite that information during conversation.

**Natural Language Processing (for TORI Chat):** When TORI Chat receives a user query in plain English (or other supported languages), the integration layer’s NLP component kicks in. It parses the sentence to understand its structure (using, for example, a combination of dependency parsing and named entity recognition). Key elements of the query are mapped to TORI’s known concepts. For instance, if the question is “How does the encryption function work in the login module?”, the NLP will identify “encryption function” and “login module” as likely referring to concepts. It will query the LCN to see if concepts exist that match those names or synonyms. If yes (say there’s a concept for a function encryptPassword and a concept for module Login), those will be activated and handed to the core to focus on. If not, the NLP can create placeholder concepts (to represent the user’s query topics) so that the reasoning engine can still work with them (perhaps the reasoning agent will then say “I don’t have information on encryption in login”).

The NLP also handles context continuation – TORI Chat keeps track of the conversation so far (via the LCN as well, where each utterance can be a concept or linked to topics). Pronouns or implicit references (“Can you explain *it* further?”) are resolved by looking at the context concepts (what is *it* likely referring to given what was just discussed). Because TORI uses a concept graph for memory, context handling is robust: instead of raw text, it knows the subject under discussion.

For generating responses, TORI has two pathways: If the answer can be constructed logically (like a series of steps or facts), TORI can compose it from the concept graph relations (ensuring each statement is grounded in a known concept or link). Alternatively, for more open-ended answers or simply to make the output fluent, TORI can utilize a template or a controlled language model to turn the structured answer into a natural-sounding one. The priority is always correctness and faithfulness to the underlying reasoning. We avoid any generative fluff that isn’t backed by the LCN or agents’ outcome. The user can, if desired, ask TORI Chat *why* it gave an answer, and the system can then reveal the chain of concepts or rules (this might be presented as a graph visualization or a step-by-step explanation, leveraging our audit trail).

**Bidirectional Interaction:** Integration isn’t just one-way. TORI not only ingests data into the concept graph, but can also effect changes based on its reasoning. For example, in the IDE, an agent might decide a piece of code should be refactored. TORI could (with user permission) perform that refactoring: the integration layer would then take the concept changes (like “Function X and Y merged into function Z”) and apply them to the actual source code (pretty-printing the new code via an AST transformation). Similarly, in Chat, if TORI comes to a conclusion that requires retrieving more info (like “to answer that, I need the user’s account history”), an agent might prompt the integration layer to fetch a file or call an API if allowed. These actions are mediated through well-defined interfaces so that security and permissions are respected (TORI will not perform an action unless configured to, especially in Chat – it won’t execute arbitrary code or calls just from a question, unless explicitly authorized as, say, a tool-using extension).

*(Current Implementation:* The integration layer is in a mixed state of completion. The Python and ELFIN language frontends are largely operational – they produce the concept graph for code and update it on changes. The C/C++ and JS integrations are basic (parsing via external tools works, but deeper semantic mapping is limited). The NLP pipeline for English is functional at a basic level (entity extraction and mapping to known concepts), using libraries like spaCy for parsing. Domain concept pack ingestion is prototyped (we tested with a small ontology, which TORI successfully merged into its graph). Much of the integration logic will be refined during the sprint as we encounter more real-world cases from user testing. Ensuring that this layer robustly handles imperfect input (e.g., code with syntax errors, or poorly worded questions) and interfaces smoothly with the rest of the system is an ongoing effort.)

**The ELFIN DSL: A Domain-Specific Language for Cognitive Coding**

One special element of TORI’s architecture is the inclusion of **ELFIN**, a custom domain-specific language designed alongside the ALAN core. ELFIN (an acronym loosely standing for *Extensible Language for Integrative Networks*, though informally named to suggest something small and magically helpful) serves as both a demo of TORI’s concept-oriented approach and a tool for users to explicitly script cognitive operations.

**Purpose and Philosophy:** ELFIN’s purpose is to allow developers to write code or specifications in a way that directly informs TORI’s concept model. Traditional programming languages require TORI to infer the concepts and intents from the code. With ELFIN, the language is constructed so that the *concepts are first-class citizens*. In practice, this means ELFIN syntax might allow the programmer to declare intentions or relationships that would be comments or implicit in other languages. For example, one might write in ELFIN:

arduino

Copy

concept DataPrivacy relates UserData, AccessControl

process scrubData(input UserData) ensures no PersonalInfo in output

In this hypothetical snippet, the programmer explicitly introduces a concept DataPrivacy and relates two other concepts to it, and defines a process with a postcondition. TORI would take this and create a DataPrivacy concept node linking UserData and AccessControl, and for the process scrubData, it knows a guarantee (no personal info in output) as a concept relation (perhaps linking the output concept to a “sanitized” concept).

Such expressiveness is valuable in domains where requirements and logic need to be very clear (financial contracts, safety-critical systems, etc.). ELFIN allows capturing those as part of the code itself, eliminating the gap between documentation and implementation. It’s similar in spirit to languages like Alloy or Z (formal specification languages), but integrated into TORI’s runtime reasoning. Indeed, one use-case for ELFIN is writing **specifications or rules** that TORI’s agents can then use. For example, an ELFIN rule might specify a design constraint (“every database query must go through the DataAccess layer concept”) and TORI’s Refactoring or Compliance agent can continuously check that this holds in the code.

**Integration with TORI:** The integration layer treats ELFIN as a first-class language. We have a defined grammar (encoded in something like a ANTLR or a custom parser) that directly produces concept graph structures. Unlike other languages, where we have to interpret the AST into concepts somewhat heuristically, ELFIN’s grammar is essentially *one-to-one mapped* with concept graph operations. If you define a concept or a relation in ELFIN, the parser calls the LCN API to create that node or edge. If you define a process or function, it creates a concept for it and also perhaps a state machine or logical relation for its pre/post conditions.

Because ELFIN is under our control, we can evolve it in tandem with features in ALAN. For instance, if we introduce a new kind of reasoning artifact (say, a “scenario” concept for test cases), we could extend ELFIN to allow writing those scenarios in code. ELFIN thus serves the dual purpose of showcasing TORI’s capabilities and giving power users a way to script or influence TORI’s reasoning. A developer could write an ELFIN snippet to guide the AI: *“Whenever concept X and Y are used together, treat them as interchangeable”* – effectively giving TORI a new rule on the fly.

**Current State:** ELFIN is currently in a prototype stage. The initial grammar is defined, covering core constructs like concept declarations, relationships, simple process definitions, and invariants. The parser can handle these and we’ve tested it on small examples (the resulting concept graph matches expectations). What remains is fleshing out the standard library of ELFIN (predefined concept types, perhaps some syntactic sugar) and integrating the ELFIN runtime semantics. Since ELFIN can describe processes, if one writes an algorithm in ELFIN, we might want to actually execute it or at least verify it. Executing ELFIN code could either mean translating it to Python/another language or interpreting it within TORI (likely out of scope for v1.0 except for trivial cases). For now, ELFIN’s emphasis is on *declarative knowledge*. Users can embed ELFIN snippets in comments or separate files to enhance TORI’s understanding of the system.

We include documentation for ELFIN in the TORI user guide as an advanced feature. We expect early adopters might experiment with it, giving us feedback. Over time, if ELFIN proves useful, it could become a cornerstone of how users explicitly communicate with the AI (almost like an API for knowledge).

**Example:** To illustrate ELFIN, consider a scenario where an enterprise wants to enforce data privacy rules in code. In ELFIN, they write:

cpp

Copy

concept PersonalData

concept EncryptedData

rule "Personal data must be encrypted at rest":

forall x: PersonalData, storageFunc f:

uses(f, x) -> ensures(EncryptedData)

This fictional syntax declares two concepts and a rule: any usage of personal data in a storage function must ensure it’s encrypted. TORI would incorporate this rule into the concept graph (perhaps as a constraint relationship). The Compliance agent, when scanning code, would then look for any function concept labeled as a storageFunc using a concept that is a PersonalData, and check if the output or result is marked as EncryptedData. If it finds a violation, it can flag it. Without ELFIN, such a domain-specific rule would be hard-coded or not known to TORI; with ELFIN, the user has effectively programmed the AI’s reasoning.

This level of customization is part of TORI’s long-term vision: an AI platform that organizations can directly imbue with their policies and knowledge through a language interface, rather than just hoping the AI “learned” it from training data. ELFIN is our first step in that direction.

**User Interface & Tools**

The User Interface and Tools layer encompasses the ways users interact with TORI’s intelligence. This includes the **TORI IDE “Thoughtboard” application**, various UI panels within it (like the concept explorer, spectral dashboard), and the **TORI Chat interfaces** (integrated and standalone).

The guiding design for UI is that it should present TORI’s capabilities in a clear, non-intrusive way, enhancing the user’s workflow without overwhelming them. The UI is also where we surface the inner workings of TORI in an understandable form (closing the loop on explainability).

**TORI IDE – Thoughtboard Interface:** TORI IDE is delivered as an independent desktop application (nickname: *Thoughtboard*). It provides a familiar coding environment augmented with TORI’s cognitive features. Key components of the Thoughtboard UI:

* **Code Editor with Inline AI Annotations:** The main editor looks and feels like a modern text editor/IDE (supporting syntax highlighting, code completion via perhaps existing language servers). TORI’s contributions come as subtle overlays: e.g., underlining code segments that TORI finds semantically problematic (not just syntax errors). If an agent has a suggestion or insight about a line or function, a small icon or colored underline appears. The developer can hover to see a tooltip with TORI’s note (e.g., “This function duplicates logic from function Y.”) or click to get a detailed explanation/fix.
* **Concept Graph Explorer:** A sidebar or pop-up panel that visualizes the portion of the concept graph relevant to the current context. For instance, if the developer is editing a function, they can open the Concept Explorer to see all concepts related to that function (variables, other functions it calls, requirements, etc.) in a node-link diagram. They can navigate this graph: clicking a concept jumps the editor to the corresponding code or brings up its documentation. The graph can be filtered or expanded – e.g., “show me the chain of concepts connecting this module to the security policy concept.” By visualizing code semantically, developers can understand impacts and dependencies that are not obvious from file structure alone.
* **Spectral Panel (Phase/Debug Dashboard):** This panel shows the dynamic analysis from the ALAN core. For example, it might display a real-time gauge of overall system consistency (if oscillators are largely synced or not), or a timeline graph of a particular concept’s phase drift over time as the code changed. In debugging mode, if the user runs the program (note: TORI IDE might integrate with a runtime to get execution traces), this panel could align runtime events with concept oscillations – essentially giving a new dimension to debug: not just variable values, but concept-level “health” signals. Concretely, a graph might show multiple sine-wave-like lines (each representing a high-level component’s phase) and highlight when one goes out of sync (pointing to a possible bug introduction time).
* **Multi-Agent Console:** TORI’s agents might produce textual logs or dialogues for the user. For instance, the Refactoring agent might have a note “I noticed functions X and Y are similar. Would you like me to refactor them?” The Test agent might output “Generated 2 new test cases for module Z.” We provide a console (or notifications area) where such messages are listed, potentially categorized by agent. This console also doubles as a command interface: the user could accept or dismiss suggestions directly from here, or ask follow-up (“Why do you suggest that?” which the agent can respond to via the Chat integration).
* **Integrated Chat Panel:** On one side of the IDE, there is a chat interface where the developer can converse with TORI Chat without leaving the IDE. This is essentially TORI Chat embedded – contextually aware of the open project. The developer could highlight code and ask, “TORI, how can I optimize this?” and the chat panel will respond, leveraging both its general training and specific knowledge from the code. Because it’s in-IDE, TORI Chat can do things like automatically link to relevant code or even execute small analyses (like “find all uses of this function” before answering).
* **Standard IDE Features Enhanced:** We include things like project navigation, file explorer, etc., but enhanced by TORI. For example, the file explorer might have badges on files indicating concept density or anomalies (imagine a file icon tinted red if that file contains concepts that are out-of-phase or marked in an agent warning). Search is semantic – beyond literal text search, you can search “concept: Login process” and TORI will find code related to the login process concept, even if the word “login” isn’t in those files (maybe it’s spread in different names).

The Thoughtboard UI is built with **local-first** principles: it does not require an internet connection to function fully (all heavy lifting is done by the ALAN core locally). This addresses privacy, and also ensures responsiveness. The UI communicates with the ALAN core via a local API or IPC mechanism (maybe an HTTP REST API on localhost, or direct memory calls if integrated).

We aim for the IDE to feel like a **modern, polished environment**. While initial versions may not have every feature of VS Code or IntelliJ, the critical ones are there, and TORI’s unique features are presented cleanly. We also brand the UI appropriately – using TORI’s theme (colors, logo) to give it a distinct identity as the “Cognitive IDE.”

*(Current UI Status:* The Thoughtboard interface exists in prototype. It currently runs as a Python/Electron app that opens a basic editor window and communicates with the core. Key panels like the concept explorer and phase dashboard are implemented in a rudimentary way (with dummy data for now). Over the development sprint, a lot of effort is allocated to hooking these up to real data and refining the UX. For example, as of now, the concept explorer can display a graph but not dynamically update; by mid-sprint it will update live as the code changes or as the user explores. Performance is another focus – ensuring that even with thousands of concept nodes, the UI remains responsive via techniques like virtualization or focusing on relevant subgraphs.)

**TORI Chat Interfaces:** TORI Chat can be accessed in two main ways: embedded in the IDE (as described) and as a **standalone application** for general conversational AI use.

* *Embedded in IDE:* This version of the interface is tailored to a developer’s needs. It might automatically inject context about the code when you ask a question (so you don’t have to type out “in function X, ...” if you have that function selected). It can also present answers with references into the code. For example, if TORI Chat explains an error, it might embed a snippet or a link that when clicked, navigates the IDE to the relevant code location. The chat UI inside IDE is minimalist (to not clutter coding space) but easily expandable.
* *Standalone Chat App:* For more general use (or for non-developers interacting with TORI’s knowledge), we provide TORI Chat as a separate desktop or web application. This has a chat window with history, persona selection, and possibly a way to manage knowledge packs. One could deploy this in an enterprise as an internal assistant: employees could query it about company policies, or troubleshoot issues by describing them in natural language. The standalone UI might allow logging in (if connecting to a central knowledge base), but in our local-first scenario, it could simply run locally with whatever knowledge packs are loaded. It will highlight that all data is local (to build trust for users worried about sensitive queries). The design is akin to popular chatbots, but with an emphasis that it *shows its work*. Perhaps an “inspect reasoning” button is available for power users to see the concept graph path or agent reasoning that led to an answer.

In both interfaces, **persona and multi-agent feedback** are important. TORI Chat can adopt different styles or roles (e.g., “explain like I’m a junior developer” vs “give me a strictly factual answer with references”). The UI might offer a toggle or prompt for that. Under the hood, this corresponds to activating different subsets of agents or applying certain response formating. Also, if multiple agents contribute to an answer (like a compliance check agent adds a caution), the UI could indicate that (“According to PolicyAgent: ...”).

**Visualization and Feedback Tools:** Aside from core features, we have tools for the user to visualize and correct TORI’s behavior. For example, an **Agent Manager** panel could list the agents, allow the user to toggle certain agents on/off or set their aggressiveness (maybe a slider for how proactive the Refactoring agent should be). This gives advanced users control and trust – they can turn off help they don’t want.

Another tool is the **Knowledge Browser**, which is essentially a read-only view into the LCN. This would list all concepts TORI knows about (perhaps grouped by type or source). A user could search their domain concepts here to see if TORI has captured them. This doubles as a debugging tool for concept integration – if TORI missed a key concept, the user might notice it here and know to feed TORI more info.

Finally, a **Settings/Privacy Panel** ensures the user can confirm that all operations are local (or connect only to specified resources), aligning with our privacy principle.

*(Current UI Status:* The standalone chat UI is at a very early stage – currently just a simple web page interface used for testing. We will use part of the sprint to build it out (likely leveraging web technologies for cross-platform ease). The integrated chat in IDE currently just opens a window to a dummy chatbot; hooking it to the real TORI Chat backend will happen once the core QA pipeline is solid. We anticipate rapid progress here once the rest is in place, since it’s largely an integration task with perhaps some front-end coding for polish.)

**Multi-Agent Orchestration**

One of TORI’s defining features is its **multi-agent system** – the idea that TORI’s intelligence is composed of multiple specialized agents working together under an orchestrator. This design mimics the human cognitive strategy of dividing problem-solving into sub-tasks (and even reflects Minsky’s Society of Mind theory, where many simple processes together yield intelligenc[en.wikipedia.org](https://en.wikipedia.org/wiki/Society_of_Mind#:~:text=In%20his%20book%20of%20the,2)】).

**Central Orchestrator:** At the center of the multi-agent layer is the orchestrator, which acts like a conductor for an orchestra of agents. The orchestrator’s responsibilities include: activating the appropriate agents for a given situation, mediating conflicts between agents, and integrating their outputs into a coherent result or user-facing action. It runs within the ALAN core and has access to the global state (concept graph, oscillator states, etc.), which it uses to inform decisions. For instance, when a code analysis event occurs (user saved a file), the orchestrator might simultaneously notify the Refactoring Agent, Documentation Agent, and Test Agent to do their analysis. If they produce outputs, the orchestrator collects them. If two outputs conflict (one agent says “remove this function” and another says “improve this function”), the orchestrator uses predefined rules or meta-reasoning to resolve it (maybe it decides improvement is the safer route and discards the removal suggestion, or flags the conflict for the user to decide).

The orchestrator can also sequence agents. Suppose answering a chat query might involve multiple steps: retrieving info, then reasoning, then simplifying the answer. Instead of one monolithic agent doing all, the orchestrator can call a KnowledgeRetrieval agent first, then pass the findings to a Reasoning agent, then perhaps to an AnswerFormatter agent. This pipeline is orchestrated such that each agent does what it’s best at. The user just sees the final answer, but behind the scenes it was a team effort.

**Specialized Agents:** We have several agents implemented or planned:

* *Refactoring Agent:* Focuses on code quality and maintainability. It scans the concept graph for code smells: duplicated logic, long functions, inconsistent naming, etc. It leverages the semantic info – e.g., if two concepts (functions) have very similar subgraphs (same calls, similar documentation), that’s a hint to refactor. It then suggests a refactoring (or even generates the refactored code via transformation rules). It can also respond to user commands like “refactor this file” by applying known refactoring patterns.
* *Documentation Agent:* Ensures that the knowledge in the code is reflected in documentation. When code changes, it updates or marks outdated any concept documentation nodes. It can generate summaries for functions by looking at related concept linkages (like pre/post conditions, usage examples from code, etc.). It might also assist in creating architecture docs by summarizing clusters of concepts (for example, produce a brief description of the “Payment subsystem” concept cluster).
* *Testing (QA) Agent:* (Not to be confused with the Q&A nature of chat – here QA means Quality Assurance.) It looks at the concept graph to find untested concepts or potential bugs. If a concept (like a function or a requirement) has no associated “test case” concept, it flags it. It can even attempt to generate test cases. For example, it might identify boundary conditions for a function by analyzing its preconditions concept, then propose a test for each boundary. It works closely with the core analysis; if, say, the Koopman analysis finds an unstable mode related to input size, the Test agent could suggest a stress test on that input range.
* *Security/Compliance Agent:* (Planned) Focuses on domain rules like “no sensitive data logged” or “function X must always be called before Y.” These could be programmed via ELFIN or policy files and this agent checks them continuously. It highlights any violations. For instance, if concept “PersonalData” flows into a function not marked as secure, it will warn.
* *Knowledge Retrieval Agent:* (Primarily for Chat) When a question is asked, this agent searches the LCN and external sources (if allowed) for relevant information. It might fetch definitions, or recall that “Oh, the user’s asking about Concept Z, which was referenced in Document Q.” It basically does the research part, so the reasoning agent can focus on logic.
* *Reasoning Agent:* (Primarily for Chat) This agent takes the assembled facts and tries to logically deduce or answer the question. It might chain together multiple facts (kind of like a theorem prover or rule engine). For code questions, it could perform static analysis steps to find an answer (like simulate execution in concept-space, etc.).
* *Persona/Stylist Agent:* This agent modulates how answers are delivered. For example, if the user selected an “Expert” persona vs a “Tutor” persona, the stylist agent will adjust wording and the level of detail. It doesn’t change the content (which comes from the reasoning agent), just the packaging. In multi-agent terms, it might even be considered a post-processing agent on the answer.
* *MPorter (Meta-Strategy) Agent:* This is the strategy agent mentioned earlier. It monitors high-level goals and progress. In the IDE, it might notice the developer has been ignoring a particular critical warning and might gently resurface it or reprioritize agent outputs (ensuring important things don’t get lost in noise). In Chat, it could break down complex user requests. If a user asks a vague multi-part question, MPorter agent might internally split it into parts and ask the knowledge agent to handle each, then aggregate results. It’s essentially thinking about *how to think* – planning the orchestration of other agents for complex tasks.
* *Attachment (Emotional) Agent:* This agent monitors the user’s emotional state (inferred from their interactions or message tone) and adjusts TORI’s responses accordingly. It introduces an empathic element. For example, if a user in chat says “I’m really stuck and this is frustrating me,” the Attachment agent will influence the response to be more supportive or apologetic (“I understand this can be frustrating. Let’s break it down together.”[en.wikipedia.org](https://en.wikipedia.org/wiki/Affective_computing#:~:text=Affective%20computing%20is%20the%20study,interpret%20the%20emotional%20state%20of)】. In the IDE, if a user seems to be thrashing (repeatedly changing code back and forth), it might suggest taking a step back or offer an explanation of the underlying issue to calm the confusion. This agent does not solve technical problems – it watches the *interaction* and ensures the AI’s behavior maintains a good rapport with the user.

**Conflict Resolution and Synergy:** With many agents, conflicts or redundant efforts can occur. The orchestrator uses a few strategies to handle this:

* *Priority rules:* Some agents have higher priority outputs (e.g., a security violation from the Compliance agent might override a suggestion from the Refactoring agent in terms of urgency).
* *Merging:* If two agents essentially suggest the same thing in different words, the orchestrator merges that into one suggestion to the user (giving credit to both, perhaps).
* *Competition:* In some cases, the orchestrator might let agents “debate.” For instance, one agent might propose solution A to a problem, another proposes B. The orchestrator could let both run through a simulation or argument and see which is more convincing (similar to how adversarial agents work in some AI frameworks). This is experimental, but TORI Chat’s multi-answer consideration could use this – generating answers via different reasoning paths and then choosing the best.
* *User input:* Ultimately, if agents disagree, TORI can surface the disagreement to the user: “There are two ways to resolve this issue: method X (suggested by DesignAgent) and method Y (suggested by PerfAgent). Would you like to consider one?” This turns a conflict into a choice for the human, which is often the right approach when design trade-offs are involved.

**Orchestration in Practice:** Suppose a developer hits “Analyze Project”. The orchestrator triggers: RefactoringAgent (checks code structure), TestAgent (looks for test coverage), ComplianceAgent (checks rules), DocumentationAgent (updates docs), and they all run in parallel. The orchestrator gathers their findings: maybe two refactor suggestions, one compliance warning, and documentation updates. It then prioritizes the compliance warning at top (marked in red in UI), lists refactor suggestions next (maybe in yellow), and notes doc updates silently (perhaps auto-applied). The developer sees a summary in the IDE’s agent console and can address them as needed.

In a chat scenario, say the user asks “How can I improve the performance of the data import process?” Orchestrator engages: KnowledgeAgent (find data import code and any known performance issues), RefactoringAgent (since it’s performance, maybe PerfAgent if we had one, to simulate usage or identify bottlenecks conceptually), ReasoningAgent (to formulate the improvement steps), then PersonaAgent (to format answer). The orchestrator sequences these, ensures the reasoning agent gets the findings from knowledge search (like “the data import reads line by line, which is slow”), and then the reasoning agent might conclude “use bulk operations or streaming”. The persona agent then crafts a response: “I notice the data import process reads data line-by-line. Combining those reads or using batch processing could significantly improve performance. For example, ...” The orchestrator sends that to the user. If the user asks “Why?”, the orchestrator might direct that to the ReasoningAgent or KnowledgeAgent to provide the conceptual reason (maybe citing that line-by-line I/O is conceptually linked to high overhead).

*(Current Status:* The multi-agent framework is in place with basic agents (Refactoring and Documentation running in the IDE context, Knowledge and a simple Q&A reasoning in chat context). Conflict resolution currently is rudimentary (essentially, we have a fixed priority order and we haven’t had complex conflicts in the prototype). Part of the sprint is dedicated to expanding these agents’ logic (as described in the 21-day plan) and refining the orchestrator policies. We’ll also be user-testing the agent outputs to adjust how verbose or aggressive each agent should be by default. The ability for a user to configure agents (turn off or tune) is planned in the UI, as mentioned.)

**Development Roadmap – 21-Day Sprint Plan (Starting May 12, 2025)**

To achieve a fully functional TORI IDE and TORI Chat by the end of the sprint, we have devised a **21-day development plan** divided into daily milestones. The plan assumes **Day 1 is Monday, May 12, 2025**. It is structured as three weeks of focused development, with each week allocating one day (Day 7, Day 14, Day 21) for rest, review, or buffer. Below is the plan for TORI IDE, followed by the plan for TORI Chat. Time-based references (Day 1, Day 2, etc.) are annotated with actual dates for clarity.

**Note:** The first week of the IDE sprint heavily emphasizes completing the partially implemented core features (oscillator sync, spectral analysis, ELFIN integration, orchestrator basics). These are foundational tasks that must be finished early to unblock dependent features. The chat sprint runs in parallel, often focusing on integration and interface aspects while core logic is firmed up in the IDE context.

**TORI IDE – 21-Day Sprint Plan (May 12 – June 1, 2025)**

**Sprint Goal:** Transform TORI IDE from a prototype into a polished v1.0 product, with all core cognitive features implemented, tested, and integrated into the Thoughtboard UI. By Day 21, the IDE should reliably provide intelligent assistance (refactor suggestions, conceptual warnings, etc.), and be ready for initial deployment to friendly users.

* **Day 1 (May 12, 2025)** – *Sprint Kickoff & Environment Setup*  
  **Goal:** Align team on objectives, finalize work assignments, and ensure the development environment is ready for rapid iteration.  
  **Tasks:** Kick off with a team meeting reviewing the Master Plan’s highlights (vision, architecture, critical components to finish). Set up version control, branch strategy, and CI pipelines. Ensure everyone can run the prototype (backend + UI) locally. Address any environment issues (e.g., if someone’s on Windows, make sure dependencies like spaCy, PyQt, etc., install properly). Clear the slate of trivial known bugs from prototypes so baseline is stable.  
  **Checkpoint:** By end of Day 1, the team is synchronized, the latest prototype code is merged to main, and each developer has a working instance of TORI IDE to build upon. Any blocking setup issues are resolved.
* **Day 2 (May 13, 2025)** – *Core Engine Implementation: Phase Synchronization*  
  **Goal:** Implement and verify the **Banksy phase-coupled oscillator synchronization** in ALAN core (replacing any placeholder logic for concept syncing).  
  **Tasks:** Develop the new synchronization algorithm based on the ψ-coupling design. Code the oscillator update loop that runs after each analysis tick: for each linked concept pair, adjust phases toward alignment (or intended phase offset). Incorporate global feedback: e.g., use a convergence factor from spectral analysis (if available) to modulate coupling strength (this ties in Day 3’s tasks). Write unit tests: create a mini concept graph in isolation and simulate oscillator steps, asserting that phases converge for connected nodes and remain independent for unconnected nodes. Remove the old dummy update code and integrate the new one into the main loop (ensuring it runs, say, whenever the concept graph changes).  
  **Checkpoint:** End of Day 2, oscillators in the core actually move and reflect influences. If concept A and B are linked, and A’s state changes, B’s phase will respond (rather than doing nothing as before). We likely log some debug info to see this in action. The code should be free of obvious stability issues (not diverging or oscillating wildly unless intended). The Kuramoto mention in documentation can be updated to note we now use the custom ψ-sync method.
* **Day 3 (May 14, 2025)** – *Core Engine Implementation: Spectral Analysis Pipeline*  
  **Goal:** Implement the **Koopman spectral analysis and Lyapunov stability checks** in the core to enable predictive reasoning.  
  **Tasks:** Build out the data pipeline for capturing state snapshots of the concept network (phases, active concept counts, etc.) over time or events. Implement a function to compute a spectral decomposition on recent state history – this could start simple: e.g., take the last N states, perform an SVD or eigen-decomposition, identify the dominant mode. Mark stable vs unstable trends (maybe define a Lyapunov function L and check L(state\_{t+1}) < L(state\_t) for stability). Integrate this so it runs periodically (not necessarily every code edit, maybe on a timer or when changes stabilize). Use results: e.g., if an unstable mode is detected, store an “alert” in core memory (the orchestrator can later translate that to a suggestion: “System architecture is oscillating – possible unresolved design decision.”). Also, feed spectral insights back into oscillator coupling if applicable (for instance, if one mode corresponds to two subsystems, perhaps temporarily decouple them by lowering coupling strength).  
  **Checkpoint:** End of Day 3, the core produces some spectral output. We should be able to see logs or debug info like “Mode1 eigenvalue = 0.95 (stable), Mode2 = 1.1 (growing unstable)”. If possible, a simple scenario where we *know* a pattern (like we oscillate a small graph input back and forth) confirms the spectral detection (it should spot the oscillation frequency). The code still might use a simplified analysis (e.g., linear trending) if full Koopman integration is complex, but the structure is in place and returning meaningful data.
* **Day 4 (May 15, 2025)** – *Integration of Core Components & Partial Testing*  
  **Goal:** Integrate the newly implemented core features (oscillator + spectral) with the rest of the system and run preliminary tests. Finish any loose ends in core (ELFIN parsing integration, etc.).  
  **Tasks:** Connect the ELFIN DSL parser into the build: ensure that if an .elfin file is present in a project, the integration layer processes it and populates concepts accordingly (this might have been partly done, make sure it’s fully working). Test the full core loop: e.g., load a sample project that has some known conceptual inconsistencies and see if the oscillator + spectral system picks them up (in logs or via some debug UI). Fine-tune parameters (coupling strengths, threshold for instability warnings) based on these tests. Address any core crashes or performance hiccups: e.g., if spectral calc is slow, maybe reduce frequency or data length for now. Essentially, stabilize the core. By now, the core is feature-complete, so begin writing documentation comments for it (to help others understand the complex logic).  
  **Checkpoint:** End of Day 4, the ALAN 2.x core is **fully operational** within the application. We have a stable build where the concept graph updates, oscillators sync, and spectral analysis runs without major issues. We likely produce some internal metrics (maybe a “core health” printout indicating system consistency) that will later be surfaced in UI. The team can proceed confident that this core foundation is ready to support UI and agent features.
* **Day 5 (May 16, 2025)** – *Thoughtboard UI: Concept Explorer & Core Data Wiring*  
  **Goal:** Hook up the Thoughtboard UI to the live data from the ALAN core (concept graph, agent outputs, etc.), focusing on the **Concept Graph Explorer** panel and inline annotations.  
  **Tasks:** Implement an API in the backend to serve concept graph data (e.g., a REST endpoint /concepts returning JSON of nodes/edges, or use an in-memory shared model if in Electron). In the front-end, replace the placeholder graph in the Concept Explorer with data from this API. Enable basic interactions: clicking a node in the graph highlights the corresponding code element in the editor (this requires mapping concept -> source location, which the backend can provide if the concept came from code). Conversely, when a user navigates to a different function in the editor, update the Concept Explorer to focus that function’s node (and its immediate relationships). Also, surface core analysis warnings: if the core has flagged an unstable mode or conflict concept, ensure the UI can highlight it (e.g., concept node might have a red outline, or the editor gutter might get an icon at relevant lines). Start displaying simple agent messages in an output panel (even if just logging them).  
  **Checkpoint:** By end of Day 5, the UI is truly “talking” to the backend: for instance, a developer can open the Concept Explorer and see an accurate graph of, say, the current file’s concepts and their links. It updates reasonably promptly after code changes or navigating to a new file. This is a big step as now TORI’s unique info is visible. Any glaring UI performance issues (like graph too slow on moderate number of nodes) are noted to optimize on Day 10. Inline annotations might still be few (since agents haven’t been fully enabled yet), but the pipeline for showing them (e.g., an underlined range corresponding to a concept with a warning) is in place.
* **Day 6 (May 17, 2025)** – *Multi-Agent System Activation (IDE context)*  
  **Goal:** Activate key **IDE agents** using the now-functional core, and integrate their outputs into the UI. This begins TORI’s intelligent behavior in the IDE.  
  **Tasks:** Enable the **Refactoring Agent** on code analysis events. Remove any placeholder logic; use real graph queries (e.g., find duplicates by concept similarity) to generate suggestions. For now, log these suggestions or list them in the agent console. Do the same for the **Documentation Agent** – e.g., after each analysis, have it update documentation concept nodes and perhaps suggest adding docstrings where missing. Ensure the orchestrator is coordinating: wire up a simple orchestrator that on each “analysis cycle” calls the active agents and collates their suggestions. Then implement UI rendering of these suggestions: e.g., a pane or popup listing “Suggestions: Function foo duplicates bar (RefactorAgent) – [Accept]/[Ignore]”. If accept is clicked, we won’t fully automate refactoring yet (unless trivial), but we could at least merge concept or mark it resolved. Also implement a way to dismiss suggestions. Start small: maybe just printing them to console initially, then improving UI as time allows. The aim is to see at least one concrete agent output by the user.  
  **Checkpoint:** End of Day 6, TORI IDE is **making intelligent suggestions**. For example, on our test project, the refactoring agent might flag two similar functions. We can see this either in the console or in an overlay UI. Multi-agent orchestration is rudimentary (likely sequential or priority-based), but working – meaning no crashes and suggestions seem relevant. This effectively is the first visible “AI assist” moment in the IDE.
* **Day 7 (May 18, 2025)** – *Review & Buffer (Core/Backend)*  
  **Goal:** Review progress of Week 1, fix bugs, and adjust plans if needed. Use as buffer for any unfinished tasks.  
  **Tasks:** Conduct a team demo of what we have so far (core working, UI showing graph, maybe one suggestion coming through). Collect any obvious issues: e.g., “the concept graph for large file is too cluttered – consider auto-grouping”, or “oscillator sync seems slow for 1000 concepts – maybe optimize coupling calc”. Spend time fixing critical bugs discovered in integration (like UI race conditions, or an agent suggestion misidentifying code). Polish some behind-the-scenes aspects: cleanup the code, add comments, perhaps improve logging structure. Update documentation (internal developer docs) to reflect how new core/agents work. Given this is a planned lighter day, ensure the team doesn’t burn out – possibly a half day of work and an afternoon of rest or learning.  
  **Checkpoint:** By end of Day 7, the team has a clear picture of what’s working well and what needs focus next. Any core or integration issues that could impede new feature work are resolved. The plan for Week 2 might be adjusted slightly based on Week 1 outcomes (for instance, if core is slower than anticipated, allocate some optimization time). The project is on track, or if behind, we have identified where to catch up using buffer days.
* **Day 8 (May 19, 2025)** – *Thoughtboard UI Polish & Advanced Features*  
  **Goal:** Improve the UI/UX of the Thoughtboard with better visualizations and controls, focusing on the **Spectral Dashboard** and user controls for agents.  
  **Tasks:** Now that the core provides spectral data, design and implement the **Spectral Debug Panel**. This could be a small graph in the corner showing overall system “coherence” over time (e.g., a line trending upward as things stabilize). Or perhaps list current active modes: e.g., “Mode: ‘Memory Leak potential’ – unstable”. Tie this with UI cues: e.g., if an unstable mode is detected, highlight the related concepts (maybe flashing icon on related modules). Also, implement **user controls for multi-agent**: a simple settings dialog where agents can be toggled on/off or set to auto-apply trivial fixes. Ensure these preferences persist in the session. Enhance the Concept Explorer by adding filtering options (checkboxes to show/hide certain concept types, like maybe turn off showing local variables to declutter). Add the ability to double-click a concept node and keep the explorer centered there (for deep dives). Finally, refine the **suggestion UI**: use icons (a lightbulb for refactor, a book for documentation suggestion, a shield for compliance, etc.) to label each suggestion’s source agent so the user knows why it’s suggested.  
  **Checkpoint:** End of Day 8, the Thoughtboard interface should feel more mature. The spectral panel is live, though it might be fairly simple in this version, it at least gives some feedback (like “System Stable” or “Instability detected in X”). Users (testers) can now control agent involvement somewhat. The concept explorer is more usable with filters. We are essentially moving from raw functionality to a cleaner user experience.
* **Day 9 (May 20, 2025)** – *Agent Enhancements (IDE)*  
  **Goal:** Increase the sophistication of the IDE agents and their coordination. Make their outputs more accurate and useful.  
  **Tasks:** Revisit the **Refactoring Agent** logic with fresh eyes after seeing initial outputs. Improve its detection algorithms: maybe implement a simple AST similarity metric for functions to catch duplicates that textual similarity might miss. Also, add new refactor types: e.g., detect when a class has too many responsibilities (if concept graph shows it’s linked to very disparate concepts, maybe suggest splitting class). Introduce a **Performance Agent** (if not separate, maybe part of refactoring agent): using concept metrics (like if a loop concept is heavily used or if a function’s complexity is high), suggest performance improvements (this can dovetail with insights from spectral if an unstable mode is related to performance). Enhance the **Documentation Agent** to auto-generate missing doc comments using available info (perhaps call a templating system or even an LLM for natural language, constrained by known facts). For multi-agent orchestration, implement a basic conflict resolution: e.g., if the Documentation agent wants to add a docstring to function X that the Refactoring agent wants to remove entirely, orchestrator should prioritize the refactor suggestion and hold off on documentation for now. Additionally, integrate the **Compliance/Security Agent** if any rules are readily available (even a dummy example rule via ELFIN). Test these in combination on sample scenarios.  
  **Checkpoint:** End of Day 9, the IDE’s intelligence feels sharper. We should see, for example, that it not only points out duplicate functions but maybe also says “Function Y is very large (50 lines) – consider refactoring for single responsibility.” Documentation agent might start filling in blanks which we can see in a diff view. We likely now have multiple suggestions possibly affecting the same code, so how orchestrator orders them is set (and visible to user with the labeling). The system’s advice breadth has increased (touching design, performance, etc. in addition to trivial duplicates).
* **Day 10 (May 21, 2025)** – *Performance Optimization & Scalability*  
  **Goal:** Ensure TORI IDE runs efficiently on larger projects and optimize any slow components (core or UI).  
  **Tasks:** Conduct profiling sessions. Load a codebase with, say, a few thousand concept nodes (maybe the TORI codebase itself) and measure UI render time for concept explorer, time taken for analysis loop, memory usage, etc. Optimize **core algorithms**: e.g., if oscillator update is O(n^2) on number of concepts, consider optimizations (since many concept pairs might have zero coupling, we could optimize by tracking edges). Possibly integrate a graph library or C extension for heavy math. Optimize **UI rendering**: use canvas or WebGL for concept graph if DOM approach lags, implement incremental graph updates instead of full redraws. Also optimize multi-agent runs: maybe stagger some agent work to not all hit at once if that causes CPU spikes (or run in parallel if CPU cores free). Check memory: ensure concept nodes are being freed if removed, etc. The goal is that medium projects (hundreds of files) do not bog down the experience. Also, consider toggling off some verbose debug logs now to reduce overhead.  
  **Checkpoint:** End of Day 10, performance is improved. Ideally, initial analysis of a moderately sized project should complete in a matter of seconds, and interactive operations (like editing a file) trigger responses that feel near-instant. If some heavy operations remain (maybe full project re-analysis), they are either optimized or demoted to a background thread so the UI stays responsive. We should document current performance metrics and decide if any features need to be cut or scaled back for v1.0 to maintain responsiveness (for instance, maybe not analyzing *every* file in huge projects, or adding a preference for scope of analysis).
* **Day 11 (May 22, 2025)** – *TORI Chat Integration (IDE & Standalone)*  
  **Goal:** Integrate the TORI Chat back-end with the IDE (embedded chat panel), and bring the standalone TORI Chat application to a functional state using the shared core.  
  **Tasks:** Connect the embedded chat panel in Thoughtboard to the ALAN core’s chat endpoint. Test end-to-end: the user types a question in the IDE, it goes to the chat pipeline (Knowledge agent finds relevant code concepts, reasoning agent forms answer), and a reply appears referencing code. Work on formatting: code excerpts in answers should be displayed nicely (monospaced, maybe syntax highlighted if feasible). Ensure the chat has access to the project’s LCN; this might involve saving a copy of the LCN or querying the live one. For the **standalone Chat app**, implement any missing pieces: perhaps a simple UI (we can reuse some web components from the embedded panel). Focus on the ability to load concept packs: create a UI control to load a file (like a finance ontology JSON), and when loaded, verify chat answers incorporate that knowledge. If the standalone is meant for non-coders, test it on general Q&A and refine response style to be more explanatory since no code context there. Tie up persona selection: add a dropdown or preset buttons in chat UI (e.g., “Detailed mode” vs “Brief mode”) which the Persona agent will interpret to modify answers.  
  **Checkpoint:** End of Day 11, TORI Chat works in both contexts. In the IDE, you can ask something like “What does this function do?” and get a meaningful answer that TORI derived from understanding the code (with maybe references or an offer to navigate to related code). Standalone, you can ask something from a loaded knowledge pack, like “What is KYC?” and get an answer drawn from the ontology loaded (with the answer clearly reflecting that domain knowledge). There may still be room to improve natural language fluency, but correctness and integration are the focus.
* **Day 12 (May 23, 2025)** – *Enterprise Readiness Features*  
  **Goal:** Add features important for enterprise deployments: **audit logging, privacy verification, configuration**.  
  **Tasks:** Implement an **Audit Log** system in the backend that records all suggestions made by TORI and actions taken (by the user or automatically). For example, log lines like “2025-05-23T10:05: RefactoringAgent suggested combining functions A and B [file names], user accepted.” or “ChatAgent provided answer about X using source Y.” Ensure these logs are stored locally in a secure way. Add a **toggle** in settings for “Send anonymous usage stats” (for our telemetry if any) defaulted off, and a clear indicator that TORI doesn’t send code to cloud (maybe a note on first launch to reassure privacy). Build a **configuration UI** to adjust performance-related settings or agent aggressiveness. Possibly integrate a **license management stub** if needed (e.g., a place to input a license key or check subscription, though for an internal plan this might not be needed yet). If time, implement a **safe mode** for Chat where it will not answer certain categories of questions (maybe irrelevant for on-prem, but if needed for liability, e.g., disallow advice on topics outside provided knowledge packs).  
  **Checkpoint:** End of Day 12, TORI is more enterprise-friendly. There is an audit log being generated that an admin could review for compliance (which also doubles as a debugging aid for us). The user can configure the tool’s behavior in meaningful ways through a preferences UI. We double-check that no data is going out that shouldn’t (run tests with firewalls, etc., to see that TORI doesn’t try to reach external servers unless explicitly asked for an update check perhaps). At this point, we are feature-complete for core functionality and we switch into fix/polish/test mode for the final stretch.
* **Day 13 (May 24, 2025)** – *User Testing and UX Refinement*  
  **Goal:** Conduct in-depth internal testing of TORI IDE & Chat with fresh eyes, and refine the UX based on feedback.  
  **Tasks:** Bring in a few colleagues (not on the dev team) for a user testing session. Have them use TORI IDE on a small project and TORI Chat on some questions. Observe where they get confused or where UI is unintuitive. Collect feedback: e.g., maybe they didn’t notice the chat panel icon, or found the concept graph overwhelming. Triage issues into “must-fix” vs “could improve later”. Focus on low-hanging fruit improvements from this feedback: adjusting iconography, adding a one-time tutorial pop-up (like highlighting “This is the Concept Explorer, which shows relationships.”), improving message wording (e.g., suggestions might say “(Refactor)…” which a user might not get; maybe say “TORI Suggestion:” instead). Fix any minor bugs discovered (like UI alignment issues, or an agent firing at the wrong time). Ensure consistency: check all button labels, terminology (“concept” vs “knowledge” – decide on user-facing terms and stick to one). If testers encountered any serious functional bugs (crashes, features not working as expected), address those immediately.  
  **Checkpoint:** End of Day 13, TORI should feel more **user-friendly**. Perhaps we added a quick start tooltip walkthrough that appears on first launch. We likely rename or clarify a few UI elements based on feedback. The application should handle common user behaviors more gracefully now (for example, if a user deletes a whole file, does the concept graph update without errors? Now it should).
* **Day 14 (May 25, 2025)** – *Comprehensive Testing & Bug Fixing (Bug Bash)*  
  **Goal:** Rigorously test all features (IDE and Chat) for bugs or edge cases and fix them.  
  **Tasks:** Do a “bug bash” where the dev team tries to break TORI in every way they can think of. This includes: extremely weird code constructs (does the parser choke?), concurrent actions (edit code while asking a chat question – does anything race or deadlock?), large inputs (open a huge file – does UI lag, and can we mitigate maybe by not highlighting concept for every single token?), error handling (shut down core unexpectedly, does UI show an error or freeze?). Test installation paths (fresh machine setup from our installer, testing user without Python installed if our bundle should cover it, etc.). List out all discovered bugs. Prioritize by severity: crashes and data loss/ corruption issues are highest, then major functional bugs, then minor cosmetic ones. **Fix aggressively** all the severe ones. For moderate ones, fix as many as time permits, especially if low risk. Some minor ones might be deferred if they’re not critical (we keep note to possibly address in a patch). Also run our automated test suite (if we have one by now, which we should for core logic) and ensure all pass. Perhaps incorporate last-minute test cases that cover new ground we learned from user tests. If any agent is misbehaving (like false positives or too many suggestions making noise), fine-tune thresholds or turn it down for release.  
  **Checkpoint:** End of Day 14, we ideally have zero known crash bugs and zero known major malfunctions. The product should be stable in heavy use. There may still be a list of minor quirks or potential improvements, but nothing that would embarrass or seriously frustrate an end user in a v1.0 trial. We are nearly code-complete; as we go into final days, focus shifts to documentation and final polish.
* **Day 15 (May 26, 2025)** – *Documentation & Final Feature Freeze*  
  **Goal:** Prepare comprehensive documentation (user guide, developer notes for internal use) and declare a feature freeze (no new features, only necessary fixes).  
  **Tasks:** Write the **User Guide** for TORI IDE and Chat. This includes: installation instructions, quickstart tutorial (covering basic workflow, e.g., “open your project, TORI will analyze and show suggestions; you can chat with TORI for help; here’s how to interpret the concept graph…”), explanation of each major UI component (maybe with screenshots), and an FAQ for common issues. Also document the **ELFIN DSL** usage (with a section describing how to write an ELFIN file and what TORI does with it). Write a section on **Troubleshooting** (like if suggestions aren’t appearing, check that agents are enabled; if performance is slow, try toggling X, etc.). Ensure the tone is accessible to the target users (developers, and possibly technical leads). Simultaneously, update the internal **Developer Documentation** – especially details of the ALAN core, so future maintainers (or open-source contributors, if it goes that way) can understand the complex parts (phase sync, spectral). This likely means adding comments in code, and maybe a short design doc summary to accompany the code. As feature freeze is now, ensure any pending merge requests for minor fixes are in, and then cut a release candidate build for final testing.  
  **Checkpoint:** End of Day 15, all documentation for release is ready or in final draft form. We have a Release Candidate (RC1) build that’s feature-frozen. The team (and perhaps some test users) will rely on the user guide to use the product – a good test if the guide is clear. We’re confident that if someone new picks up TORI with this documentation, they can get started and understand the basics.
* **Day 16 (May 27, 2025)** – *Final Validation & Beta Release*  
  **Goal:** Validate the RC build against real-world use cases one last time, and if all is good, designate it as the beta/release version.  
  **Tasks:** Do a final full run-through of the user experience: from installation, to opening a project, to acting on suggestions, to using chat, to checking logs and settings. Ensure the audit log is indeed capturing events. Try it on a couple of different projects (maybe one open-source repo, one internal project) to see how it behaves with code we didn’t specifically tailor for. If any critical issues appear, fix them (only critical fixes at this stage). Once satisfied, prepare the build artifacts for release: e.g., compile the Electron app to an installer for Windows/Linux/macOS as needed. Perform hash checks, virus scans, etc., as part of release prep. Label the build version (v1.0.0-beta perhaps). If there’s time, have a brief “beta release rehearsal” – writing the announcement or notes summarizing what’s in this version, known limitations (for transparency).  
  **Checkpoint:** End of Day 16, we have what we believe is the release build. It’s been tested in scenarios beyond our unit tests (some integration tests with sample user behaviors). We have not found any showstopper issues; any that were found are fixed or documented with workarounds if minor. The product is ready to deliver to the initial set of users (which could be internal stakeholders or a closed beta group).
* \*\*Day
* **Day 17 (May 28, 2025)** – *Release Preparation & Packaging*  
  **Goal:** Finalize all packaging and distribution aspects for the upcoming release.  
  **Tasks:** Prepare the installer or distribution packages for TORI IDE (Thoughtboard) on all target platforms (e.g., create a Windows installer, a Mac app bundle, etc.). Verify that each package includes all necessary dependencies (the AI models, knowledge packs, etc.) and that installation is smooth. Double-check that the audit logs and configuration files go to appropriate paths (e.g., user’s AppData or ~/.tori). Create a **Release Notes** document highlighting new features, known issues, and usage tips. Coordinate with any internal stakeholders about the release schedule.  
  **Checkpoint:** By end of Day 17, installable packages for TORI IDE and TORI Chat are ready. We have tested installing and uninstalling them on fresh machines. The release notes and any internal briefing materials are prepared. We’re essentially ready to roll out the software pending final sign-off.
* **Day 18 (May 29, 2025)** – *Internal Launch & Training*  
  **Goal:** Conduct an internal launch of TORI with a controlled group of users and provide training.  
  **Tasks:** Host an internal presentation or demo for stakeholders and potential pilot users. Walk through the vision, demonstrate the product live (showing how Thoughtboard identifies a bug and how Chat can explain a piece of code). Distribute the user guide and have a Q&A session. Set up a feedback channel (like an internal forum or email) for pilot users to report issues or suggestions. This day also serves as a buffer in case any last-minute critical bug appears during the demo – we can address it immediately.  
  **Checkpoint:** End of Day 18, the pilot users have the software in hand and are excited about its capabilities. They know how to use it at a basic level thanks to the training. We gather their initial reactions (which seem positive, say, enthusiasm about seeing conceptual links in their code). The team is on standby to support them as they start using TORI in daily work.
* **Day 19 (May 30, 2025)** – *Feedback Analysis & Minor Fixes*  
  **Goal:** Collect and analyze initial feedback from pilot users, and address any minor issues that surface.  
  **Tasks:** Monitor the feedback channels. If a pilot user reports a bug (“the concept graph viewer crashes on my project”), attempt to reproduce and fix it promptly. If any usability issues are noted (e.g., “I didn’t realize I had to click the lightbulb for suggestions”), consider quick tweaks or clarifications to the UI or documentation. Since we are technically post-release internally, limit changes to low-risk, high-value fixes (no new features). Possibly issue a quick patch update to pilot users if needed. Also, log enhancement ideas that are out of scope now but valuable for future versions (e.g., more agents for other domains).  
  **Checkpoint:** End of Day 19, any immediate post-launch fires are put out. Pilot users are not blocked by any major issues. The initial feedback is incorporated into our planning backlog for future improvements, but the current version remains stable.
* **Day 20 (May 31, 2025)** – *Go/No-Go Meeting & Launch Prep*  
  **Goal:** Decide on proceeding to broader release and prepare any external launch materials (if applicable).  
  **Tasks:** Hold a meeting with stakeholders to review pilot phase results. If the software has proven stable and valuable, get approval to move to the next stage (wider enterprise deployment or perhaps an external beta). If any serious issues arose, decide if they must be fixed before wider release or if documentation/training can mitigate them. Assuming a “Go” decision, prepare the external-facing materials: a polished announcement highlighting TORI’s strengths (e.g., a blog post or press release draft emphasizing “first cognitive OS IDE with explainable AI” etc.), and any marketing collateral like screenshots or short demo videos. Ensure legal compliance for release (open-source licenses in order, disclaimers included).  
  **Checkpoint:** End of Day 20, we have a green light to launch TORI to its intended wider audience. All necessary materials and logistical considerations for the launch day are ready. We’re essentially counting down to introducing TORI beyond the pilot group.
* **Day 21 (June 1, 2025)** – *Launch & Retrospective*  
  **Goal:** Officially launch TORI IDE and TORI Chat v1.0 and conduct a team retrospective on the sprint.  
  **Tasks:** Flip the switch on distribution – e.g., publish download links internally or externally, send out the announcement communications. The team will be actively monitoring for any issues as new users onboard (ready to provide support). Also, hold a **sprint retrospective** meeting: discuss what went well during the 21-day execution, what could be improved in our process, and gather lessons learned for the next development cycle. Acknowledge the team’s accomplishment (perhaps a small celebration for delivering this ambitious project on schedule!).  
  **Checkpoint:** By the end of Day 21, TORI IDE and ALAN 2.x (with TORI Chat) are officially released to their initial audience. The development sprint concludes with a functioning, innovative product in users’ hands. The team has documented insights to guide future sprints. We have successfully turned the TORI vision into a reality, setting the stage for further evolution.

**SWOT Analysis**

After completing the sprint and preparing for launch, we revisit TORI’s **Strengths, Weaknesses, Opportunities, and Threats** in the current context (mid-2025):

* **Strengths:** TORI’s core strength is its fundamentally different approach to AI assistance. Instead of relying solely on black-box predictions, TORI builds a *conceptual model* of the code and conversation, enabling **deep semantic understanding**. This yields more insightful and context-aware assistance than pattern-matching tools. Moreover, TORI’s design is **transparent and explainable** – every suggestion can be traced to a reasoning path in the concept graph, addressing a key demand in industry for explainable AI solutio[ibm.com](https://www.ibm.com/think/topics/black-box-ai#:~:text=White%20box%20AI%2C%20also%20called,Users)0】. The system’s **phase-synchronized reasoning** provides deterministic, reproducible results, which is a boon for enterprise users who need consistent behavior. Another major strength is TORI’s **local-first architecture**: all processing can be done on-premises, meaning companies can use TORI without exposing proprietary code or data externally. This privacy-preserving design is a differentiator, as it sidesteps cloud compliance issues and drastically reduces security concerns. Additionally, TORI’s multi-agent modularity makes it *extensible* – new capabilities can be added as separate agents without overhauling the core. Finally, TORI effectively combines development tooling with AI assistance; it’s a unified platform, which means users don’t juggle separate tools for coding, debugging, and asking questions – boosting productivity by keeping the workflow in one place.
* **Weaknesses:** TORI is an early-stage product and faces several limitations. The most immediate weakness is the **unproven nature** of its approach – while we’ve tested it on pilot projects, it lacks extensive real-world validation across diverse codebases. Its novel algorithms (oscillator networks, spectral analysis) may have edge cases we haven’t encountered; in contrast, traditional tools are very battle-tested. In terms of product maturity, TORI IDE currently lacks many of the ecosystem conveniences that established IDEs have (e.g., a rich plugin ecosystem, ultra-refined UI elements). We support only a handful of programming languages robustly (Python and ELFIN fully, others like C++ in a basic way), so users working in other languages might find TORI less helpful – this **limited language support** could narrow our initial user base and give an impression of incompletenefile-bnjjcfgff6iusnbzdwqayo2】. Performance, while much improved, could become an issue on extremely large projects – our advanced analysis might be slower than simpler tools for codebases with tens of thousands of files (something we will need to optimize as we scale). Another weakness is **user adoption risk**: developers have ingrained workflows, and some might be resistant to adopting a new IDE, especially one that introduces a radically different paradigm of AI involvement. Convincing them to trust TORI’s suggestions (and not see it as overly intrusive or gimmicky) will require education and gradual proof of value. Lastly, as a small team/new entrant, we have limited resources for support – any bugs or shortcomings early on could sour user perception if not addressed rapidly, so we must be very responsive, which can be challenging.
* **Opportunities:** The environment in 2025 is ripe for TORI’s unique value proposition. **Enterprise demand for AI** that is both powerful and private is high – many companies hesitate to use cloud-based AI coding tools due to intellectual property concerns. TORI can fill that gap by offering a **secure, on-prem AI assistant** that integrates with their internal knowledge (via concept packs) and doesn’t leak data. This is an opportunity to become the AI solution of choice for industries like finance, healthcare, or defense, where data control is non-negotiable. There is also a broader industry trend toward **augmented developer tooling**; TORI’s emphasis on explainability and determinism could position it as the *trusted* alternative to tools like GitHub Copilot (which sometimes outputs code without rationale). We can capitalize on growing awareness of AI’s “Clever Hans” problems (when models give the right answer for the wrong reasons) by showing how TORI avoids that and provides proof for its answe[ibm.com](https://www.ibm.com/think/topics/black-box-ai#:~:text=Reduced%20trust%20in%20model%20outputs)8】. Another opportunity is to extend TORI’s concept-based approach beyond code – for example, into data science notebooks, or integration with design tools – essentially, TORI could evolve into a general “cognitive assistant” platform. The multi-agent architecture means we could relatively easily add specialization for other domains (documentation writing, requirement analysis, etc.), expanding our market reach. Additionally, there is potential for **community and open-source contributions**: by sharing some of TORI’s DSL (ELFIN) or concept libraries, we could build a community that contributes domain packs (imagine domain experts encoding best practices in ELFIN for others to use). Lastly, if TORI demonstrates success, there is an opportunity for partnerships with larger IDE vendors or cloud providers – either integrating TORI’s technology into their offerings or collaborating to offer a hybrid solution (with TORI providing on-prem cognition and a partner providing cloud-scale resources if needed).
* **Threats:** The software development tooling market is highly competitive and is seeing rapid innovation in AI. A key threat is that **incumbent IDE vendors and big tech companies** will quickly advance their own AI-assisted tools. For example, Microsoft’s GitHub Copilot and Visual Studio integration is continuously improving, and JetBrains (IntelliJ) is introducing AI features – these players have enormous user bases and resources. If they solve the explainability or on-premise issues before TORI gains traction, TORI’s unique selling points could be diminished. We already see early moves in this direction, with competitors adding features to inspect and constrain AI suggestions. Another threat is **user skepticism or resistance** to TORI’s novel concepts. Developers might trust traditional linters or simpler AI assistants more because TORI’s internal workings (oscillator phases, spectral modes) are new and might seem arcane. We have to prove that these concepts genuinely translate to better results; otherwise users may dismiss TORI as over-engineered. There’s also the risk of **technical adoption barriers**: some IT departments might be slow to approve a new tool that has so many components (they might worry “Is it stable? Is it secure?” – thorough testing and certification may be demanded). Additionally, as TORI handles critical knowledge, any mistake (like an incorrect refactoring suggestion that introduces a bug) could undermine confidence – while our aim is to minimize such errors via careful design, no system is perfect. Larger competitors might also leverage their ecosystems to lock users in (for instance, if a competitor’s AI integrates seamlessly with their cloud devops pipeline, that convenience might outweigh TORI’s benefits for some teams). Finally, from a business perspective, if TORI gains attention, others may attempt to emulate our conceptual approach (we must continue innovating and possibly secure intellectual property, though patents in this space are tricky). In summary, the threats are the fast-moving competitive landscape and the need to win hearts and minds in a conservative segment of users.

**Conclusion:** Over the 21-day master plan, we have brought the TORI vision to life – creating a **cognitive operating system** that turns development and chat interfaces into truly intelligent, collaborative experiences. The resulting TORI IDE and TORI Chat (powered by the ALAN 2.x engine) demonstrate the feasibility and value of concept-oriented AI: they can understand code at a deeper level, explain their reasoning, and adapt to user needs, all while keeping the user in control. Initial testing indicates that TORI can catch subtle issues and provide insights that traditional tools miss, validating our paradigm shift towards *“concepts over tokens, memory over stateless prediction.”*

As we launch TORI to a broader audience, we will closely support our users and incorporate their feedback. Our SWOT analysis shows we are entering the market with a strong, differentiated solution, though we remain mindful of the challenges ahead. By maintaining our focus on technical rigor (grounded in proven science like Koopman theory and Lyapunov stability) and user-centric design (transparency, privacy, integration into workflows), TORI is well-positioned to carve out a niche of loyal users who need trustworthy AI assistance.

In the coming months, completing any partially implemented components and broadening language support will be priorities before resuming rapid feature development. With the foundation built in this master plan, those next steps will be on solid ground. TORI aims to herald a new era of developer tools – one where our IDE is not just a static environment, but a **cognitive partner** that evolves with us, ultimately making software development and knowledge work more intuitive, reliable, and powerful than ever before.

**Comprehensive Master Plan and Strategic Analysis – TORI IDE & ALAN 2.x Cognitive OS**

**Executive Summary**

**TORI** is a next-generation *cognitive operating system* comprising two flagship products: **TORI IDE** (an AI-augmented Integrated Development Environment) and **TORI Chat** (an intelligent chat assistant, with an enterprise-grade variant). This master guide serves as a comprehensive report, technical specification, development roadmap, and marketing playbook for the TORI suite. It encapsulates the visionary paradigm behind TORI, the current architecture and module breakdown, a 21-day sprint plan for rapid development, SWOT analyses, and go-to-market strategies for each product.

* **Vision:** TORI embodies a paradigm shift from traditional AI tools. Rather than relying on rote token prediction, TORI’s core is grounded in **concepts over tokens** and **memory over stateless prediction**[medium.com](https://medium.com/syncedreview/from-token-to-conceptual-the-rise-of-metas-large-concept-models-in-multilingual-ai-b32acbfeb792#:~:text=generalization%20across%20languages%2C%20outperforming%20existing,LLMs%20of%20comparable%20size). The vision is an “AI-native” development experience in which the system truly understands code and conversations at a conceptual level, enabling unprecedented levels of assistance and insight. Developers move from using tools to collaborating with a *cognitive partner*.
* **Architecture:** At TORI’s heart is the **ALAN 2.x cognitive engine**, a *spectral-phase reasoning* core that maintains a rich **Large Concept Network (LCN)** of knowledge. This core powers both the IDE and Chat, providing deterministic, auditable reasoning steps instead of opaque neural guesswork[ibm.com](https://www.ibm.com/think/topics/black-box-ai#:~:text=White%20box%20AI%2C%20also%20called,Users). Surrounding the core are modular subsystems for language processing, user interaction, and multi-agent orchestration. In the IDE, this means deeply understanding code structure and evolution; in Chat, it means maintaining contextual, persona-driven dialogue. Both products share a **local-first, privacy-conscious design** that can operate fully on-premises without cloud dependence, keeping user data and code private.
* **Current Status:** The TORI codebase has been through early prototyping phases. Key modules for concept parsing, memory graph management, and a basic front-end exist, though some components remain mock or placeholder implementations pending integration of the full ALAN engine. For example, a document ingestion pipeline can index PDFs into the concept network, but the **phase oscillator network** underpinning dynamic reasoning is only *partially implemented*. The current TORI IDE UI is functional as a prototype “**Thoughtboard**” dynamic code canvas, albeit without full polish or all expected IDE features. TORI Chat’s core reasoning back-end is in place, but its user-facing chat interface and multi-agent persona system are at a rudimentary stage. These gaps are identified in the module-by-module analysis, and the plan is to replace placeholder code with production-ready logic as a prerequisite to the development sprints.
* **Roadmap:** A detailed **21-day sprint plan** for each product outlines the path from the current state to a viable v1.0 release (with Day 1 starting May 12, 2025). Each sprint plan is organized into daily milestones (with one rest/review day built in per week). The TORI IDE plan focuses on integrating the ALAN core with the IDE front-end (Thoughtboard UI), implementing conceptual debugging tools, and eliminating any “mock” data paths. The TORI Chat plan emphasizes developing a robust conversational interface, integrating domain-specific “concept packs,” and adding enterprise features like audit logs and on-prem deployment support. Both plans culminate in a synchronized launch, ensuring the IDE and Chat can leverage each other (e.g. the IDE generating chat-ready knowledge graphs, and the Chat agent assisting within the IDE). **Before these sprints formally begin, however, the partially implemented core components (oscillator sync, spectral engine, ELFIN parser, orchestrator) must be completed to provide a solid foundation for rapid feature development.**
* **SWOT Highlights:** TORI’s **Strengths** lie in its unprecedented approach: a phase-synchronized reasoning core that yields deterministic, explainable insights (a “glass-box” AI approach)[ibm.com](https://www.ibm.com/think/topics/black-box-ai#:~:text=White%20box%20AI%2C%20also%20called,Users), and a local-first architecture that sidesteps the compliance and cost issues of cloud-only AI solutions. **Weaknesses** relate to typical early-stage challenges: limited UI polish compared to incumbent tools, a narrow language support (only Python and the custom ELFIN DSL are fully supported at launch), and the need for real-world benchmarking to validate the novel reasoning approach. **Opportunities** include a growing demand for AI tools that can be deployed on-prem for privacy, and integration into enterprise workflows that require transparency and auditability. **Threats** include rapid moves by industry giants to add similar AI capabilities to established platforms, and skepticism toward unproven AI paradigms – challenges the plan addresses via an aggressive development and validation strategy.

**Vision and Guiding Principles**

**TORI’s vision** is to create a *cognitive operating system* that transforms our interaction with software development and AI. This vision is grounded in two key ideas: **conceptual understanding over surface-level output**, and **persistent learning over stateless interaction**. Traditional AI coding tools operate on the level of syntax and tokens, predicting text without true understanding. TORI takes a different path – it builds and manipulates rich *conceptual models*. In the IDE, this means understanding the abstract architecture and intent behind code; in Chat, it means grasping the semantics and context of the conversation, not just stringing sentences together. By elevating to the conceptual level, TORI can provide insights and assistance that feel genuinely intelligent and context-aware, rather than generative but shallow. *“You’ve articulated not just a technical blueprint, but a paradigm shift in how we think about code, tools, and developer cognition,”* as one early reviewer noted.

Furthermore, instead of treating each user query or coding session as an isolated event, TORI maintains a **persistent memory** of interactions and knowledge. This is not a simple transcript log; it’s a structured memory in the form of the LCN (Large Concept Network), phase-state information, and learned patterns. TORI remembers *why* a piece of code was written or *how* a conclusion in chat was reached, enabling it to explain its reasoning and build on past knowledge. This contrasts with stateless large language model tools that have no inherent memory of past interactions. TORI’s approach yields an AI that can learn and adapt over time – accumulating **wisdom** rather than resetting at each prompt.

These founding ideas lead to a core vision statement: **TORI is not just a development tool or chatbot; it is a thinking partner that grows with you.** The IDE is envisioned as a “*Conceptual Canvas*” where code and design interact dynamically. The chat is envisioned as a multi-mind conversational agent that can reason through complex problems with traceable logic. Together, they form a unified cognitive workbench for creators.

From this vision, several **guiding principles** direct our design and development:

* **Human-Centric Design:** TORI keeps the human user in control and informed at all times. A guiding principle is to *“keep humans in the loop”* throughout the experience. Features like explainable suggestions, the ability to inspect the concept graph, or toggling agent assistance on/off are critical. The user should feel TORI is amplifying their abilities, not automating them away without transparency.
* **Transparency and Explainability:** Every AI-driven action in TORI should be explainable via the concept graph or a clear rationale. If TORI IDE suggests a refactoring, it can point to the concept nodes and phase relationships that led to that suggestion (e.g. a function concept that is out-of-phase with others, indicating a design inconsistency). TORI Chat, similarly, doesn’t just answer — it can provide a brief explanation or show which internal “train of thought” (which agent or knowledge source) produced each part of the answer. This fosters trust and positions TORI as a *glass-box AI*, in contrast to the opaque *“black-box magic”* of many AI assistants[ibm.com](https://www.ibm.com/think/topics/black-box-ai#:~:text=White%20box%20AI%2C%20also%20called,Users).
* **Deterministic Core, Creative Edges:** TORI’s architecture emphasizes *deterministic reasoning* in its core algorithms (the spectral-phase logic, rule-based inferences, etc.) to ensure reproducibility and auditability. However, we recognize the need for creativity and flexibility at the user interface; therefore, stochastic AI elements are allowed at the periphery (for instance, a generative language model might be used to polish natural language summaries or suggest variable names). The principle is to use randomness only where it’s safe and beneficial (creative suggestions), but never in the core understanding of code or context. This yields a system that is both **reliable** and **inventive**.
* **Local-First and Privacy:** TORI is built with a *local-first* mindset – users retain ownership of their code, data, and even the AI’s reasoning process. Both TORI IDE and TORI Chat can run fully locally or on-premises, meaning no sensitive data leaves the user’s machine unless explicitly allowed. This addresses the needs of enterprises and individuals concerned with cloud privacy. By avoiding cloud dependence, TORI also achieves lower latency and tighter integration with the user’s environment (accessing local files, dev tools, etc.), all while ensuring compliance with data governance policies.
* **Modularity and Extensibility:** The cognitive OS is modular by design. Each subsystem (parsing, concept storage, reasoning agents, UI) is a separate component with well-defined interfaces. This ensures that future improvements or replacements (e.g., swapping in a new code parser for a different language, or adding a new reasoning agent) can be done without disrupting the whole. It also means parts of TORI can be extended or used independently – for example, third-party developers might write new ALAN agents or integrate TORI’s concept graph into their own tools. Modularity also improves maintainability; given TORI’s ambitious scope, being able to upgrade pieces in isolation is crucial.
* **Consistency with Developer Workflow:** Especially for TORI IDE, a key principle is *“meet developers where they are.”* TORI introduces new concepts (like spectral debugging or concept navigation), but it also supports familiar workflows (editing code text, using Git, running tests) so as not to alienate users. TORI’s advanced features augment – rather than replace – a developer’s normal tasks. For instance, the concept graph is accessible for insight, but it doesn’t force a new paradigm on the developer; one can code normally and tap into TORI’s conceptual insights on demand. Likewise, TORI Chat is designed to integrate with tools developers already use, like Slack or other team communication platforms (for example, via a plug-in or API that allows TORI Chat’s capabilities to be accessed in a Slack channel). The motto here is *“low floor, high ceiling, wide walls”* – easy to start using, with immense power as you learn more, and broadly applicable to many tasks.
* **Inspiration and Playfulness:** While scholarly rigor underpins TORI (drawing on cognitive science and spectral theory), the project also embraces a creative, playful spirit. This is reflected in internal code names and metaphors (“**Banksy**” oscillators, “concept cartridges,” etc.), and should come through in the product identity. We believe a touch of playfulness encourages exploration – for example, TORI might include Easter eggs or humorous messages in its UI (configurable based on user preference). This principle helps differentiate TORI’s brand: it’s deep tech with a human touch and a wink of fun where appropriate. (As one quip in the planning stage put it, *“TORI isn’t a tool; it’s the mirror in which your code practices mindfulness.”*)

These principles ensure that as we develop TORI, we maintain a balance of intellectual rigor (truly innovating in AI) and user empathy (genuinely empowering users in intuitive ways). The next sections translate this vision and these principles into a concrete architecture and development plan.

**Cognitive OS Architecture Overview**

TORI’s architecture is a modular, layered cognitive system designed to support both an IDE and a Chat application via a shared core intelligence. At the highest level, the architecture comprises four integrated subsystems that interact closely:

* **ALAN 2.x Cognitive Core –** the “brain” of the system, responsible for understanding and reasoning over the Large Concept Network.
* **Language/Domain Integration Layer –** feeds the core with domain-specific inputs (code for the IDE, natural language for the Chat) and translates the core’s outputs back into human-friendly form.
* **User Interface & Tools –** the front-end through which users interact (the code editor canvas, debugging dashboards, chat interface, etc.).
* **Multi-Agent Orchestration –** the coordination layer managing specialized AI agents that perform distinct roles (e.g., a Refactoring Agent in the IDE, a persona-based Expert Agent in Chat), synchronized via the ALAN core.

Below, we describe each of these subsystems in detail, then explain how they come together specifically in TORI IDE and TORI Chat.

**ALAN 2.x Cognitive Core (Conceptual Engine)**

The ALAN 2.x core is the heart of TORI: a cognitive engine that represents and reasons about knowledge in a **Large Concept Network (LCN)**. This is effectively a graph of concepts (nodes) and their relationships (edges) spanning code, documentation, and conversational context. The core’s distinguishing feature is its *spectral-phase reasoning* architecture – inspired by dynamical systems theory and cognitive science – which ensures that TORI’s reasoning is grounded in **temporal synchrony** and **spectral analysis** rather than static rules or purely stochastic jumps.

At a high level, the core operates as follows: each concept in the LCN is associated with internal state variables, including one or more **oscillators** that represent the concept’s “phase” or activation rhythm. When the system processes information (new code, a user query, etc.), concepts are activated and interact via these oscillatory states. The engine analyzes the collective behavior of these oscillators using spectral methods to infer coherence, causality, and emerging patterns of thought. In essence, TORI treats reasoning as a *dynamic process*—a kind of controlled resonance among ideas. This approach is loosely inspired by how human brains exhibit synchronized neural oscillations during cognition, where groups of neurons firing in unison can bind together related pieces of information for unified processing[neurosciencenews.com](https://neurosciencenews.com/brain-rhythm-cognition-25941/#:~:text=It%20could%20be%20very%20informative,%E2%80%9D). By simulating a form of phase synchrony among concept nodes, ALAN aims to achieve a machine analog of that cognitive binding: concepts that should logically align will tend toward phase alignment, whereas contradictory concepts will oscillate out-of-sync, flagging inconsistency.

**Phase-Coupled Oscillator Matrix (“Banksy” Core)**

At the core of this dynamic reasoning is the **phase-coupled oscillator matrix**, code-named “Banksy.” This is essentially a network of mathematical oscillators attached to concept nodes, modeling the timing and rhythm of concept interactions. It draws inspiration from the Kuramoto model (a well-known model of coupled oscillators for synchronization) but extends beyond it. In ALAN 2.x, we have **replaced the simple Kuramoto synchronization with a ψ-based spectral coupling approach** – using advanced techniques from Koopman operator theory and Lyapunov stability analysis to govern how oscillators influence each other’s phase. Each concept node may have an oscillator representing its state in a given context or reasoning “mode.” For example, in the IDE context, an oscillator might track how “in sync” a module is with the overall codebase design; in the Chat context, oscillators might represent alignment or divergence of a user’s statements with factual knowledge or a persona’s beliefs.

When concept A influences concept B (say, function A calls function B in code, or a user query references a knowledge concept), the oscillators of A and B are coupled: they tend to synchronize if the relationship is harmonious or diverge if there is tension. By observing phase alignment or divergence across the network, the ALAN core can detect coherence or conflict among ideas (e.g., two design components oscillating out-of-phase might indicate a logical contradiction or an API mismatch that needs resolution). The system’s use of **ψ (psi) functions** refers to the eigenfunctions in the Koopman operator framework (discussed below) that capture the essence of these oscillatory modes.

Critically, this oscillator matrix provides a temporal dimension to reasoning. Rather than evaluating code or chat context with static rules, TORI simulates a **continuous reasoning process** that can converge toward stable conclusions. The introduction of Lyapunov stability metrics ensures that this process remains stable and convergent. In control theory terms, we treat a reasoning session as a dynamical system that should settle into an equilibrium representing a consistent understanding or answer. If a small perturbation (say a minor code change or a slight rephrase of a question) causes only a transient wobble but the phases re-synchronize, the system is stable in the sense of Lyapunov – meaning the reasoning is robust to minor changes[ocw.mit.edu](https://ocw.mit.edu/courses/16-30-feedback-control-systems-fall-2010/463e9559c6ef70f13ea51c3464bdc9c1_MIT16_30F10_lec22.pdf#:~:text=%E2%80%A2%20Stable%20in%20the%20sense,%E2%80%9D%20November%2027%2C%202010). This approach is novel in AI reasoning: we are effectively designing the AI’s thought process to have provable stability properties, which guards against chaotic jumps or oscillatory deadlocks in inference.

*(Implementation Note:* As of May 2025, the phase-coupled oscillator subsystem is **partially implemented**. Basic oscillator objects and their coupling equations exist, but the sophisticated ψ-based synchronization algorithm is still in prototype stage. Earlier prototypes used a plain Kuramoto model for phase synchronization; the new approach will incorporate the **Koopman spectral method** described next, which is a work in progress. Finishing this oscillator sync logic with the new model is a top priority before feature sprint work resumes.)

**Koopman Morphic Engine (Spectral Analyzer)**

Complementing the time-based oscillator model, ALAN 2.x employs a **Koopman morphic engine** for spectral analysis of the concept network’s state. The **Koopman operator theory** provides a powerful framework to analyze nonlinear dynamic systems by lifting them into a higher-dimensional linear space via eigenfunctions[mdpi.com](https://www.mdpi.com/2227-7390/9/19/2495#:~:text=The%20Koopman%20operator%20theory%20is,are%20amenable%20to%20systematic%20analysis). In simpler terms, our engine monitors changes in the concept graph and oscillators and projects those changes into a spectral domain (frequencies/modes) to find underlying patterns or invariants in TORI’s reasoning.

For example, in the IDE, as the codebase evolves over time (commits, refactors, additions), the engine can detect repeating patterns or cycles in development by analyzing the sequence of concept graph states – akin to finding a “frequency” at which a certain design issue reoccurs. In TORI Chat, the Koopman analysis might identify recurring discussion themes or predict how a dialogue could evolve (almost like sensing the “direction” or *momentum* of the conversation). The engine treats conceptual changes like signals, and by applying spectral decomposition, it can forecast future states or recognize known patterns. Essentially, it gives TORI a rudimentary ability to *anticipate* and plan, not just react.

Mathematically, if the concept state at time *t* is represented by some vector of features (active concepts, phase differences, etc.), the Koopman engine tries to approximate a linear operator **K** that evolves these features: xt+1≈Kxt.x\_{t+1} \approx K x\_t.xt+1​≈Kxt​. Even though the underlying system is nonlinear, **K** (the Koopman operator) operates on a lifted representation (comprised of eigenfunction observables, often denoted ψ). This allows use of linear techniques: for instance, we can compute eigenvalues of **K**, which correspond to modes of system behavior (stable modes, oscillatory modes, etc.). The spectral decomposition yields **Koopman eigenfunctions** that represent coherent patterns in the reasoning process[mdpi.com](https://www.mdpi.com/2227-7390/9/19/2495#:~:text=The%20Koopman%20operator%20theory%20is,are%20amenable%20to%20systematic%20analysis). Some of these modes might correspond to stable invariants (e.g., a conclusion that remains true throughout the session), while others might indicate unstable directions (e.g., a line of reasoning that diverges or an idea that keeps oscillating without settling).

We also explicitly integrate **Lyapunov exponents** into this analysis: these measure how small deviations in state evolve (do they die out or amplify?). A negative Lyapunov exponent indicates a stable mode (perturbations decay), whereas a positive one would warn of chaotic tendencies. By monitoring these, TORI can assure that its reasoning stays within stable bounds. In practice, this means if TORI begins to chase a line of thought that is leading to divergence (say, an oscillation that grows or an argument that spirals without resolution), the system can detect it and either dampen that interaction or invoke a conflict-resolution agent to intervene. This is an example of **spectral modal alignment**: ensuring that the modes (eigenvectors) of the reasoning process align with convergence and consistency criteria.

*(Implementation Note:* Currently, the Koopman spectral analysis in ALAN 2.x is also **in prototype form** – a scaffold exists that can record state snapshots and perform a rudimentary spectral transform (e.g., via an FFT or eigen-decomposition) but it returns dummy results. Integrating a true Koopman operator computation (likely leveraging an existing numerical library for eigen-analysis) is planned. We may not implement a full infinite-dimensional Koopman (which is theoretically complex), but a finite approximation capturing dominant modes. This will be developed alongside the oscillator sync. The design goal is that by the end of our core integration, the system’s inference pipeline will be: **concept graph update -> oscillator dynamics -> spectral (Koopman) analysis -> stable state check**, yielding a cycle of inference that is *explainable* and *provably convergent*.)

**Static Knowledge Kernels and Memory**

In addition to the dynamic reasoning modules above, the ALAN core includes a static knowledge comparison component, sometimes referred to as **spectral kernels** or embedding similarity modules. While the oscillator network and Koopman engine provide a view of *dynamic* behavior, the spectral kernels handle *static* semantic comparisons – for example, finding which concepts in the graph are semantically similar, or which past scenarios (code patterns, Q&A pairs) resemble the current one. In implementation, this involves using vector embeddings for concepts (perhaps pre-trained or learned over time) and computing similarities. This subsystem complements the phase-based reasoning with more conventional AI pattern-matching (for instance, identifying that a newly encountered concept “login routine” is similar to a past concept “authentication function” based on their embeddings, and thus inferring related constraints).

The Large Concept Network, combined with these spectral and embedding-based tools, forms a **memory substrate** for TORI. Crucially, this memory is **deterministic** and **replayable**: given the same sequence of inputs and events, the concept graph and its oscillator states should evolve the same way every time. This determinism is important for auditability (especially in enterprise use) – one can trace exactly how a conclusion was reached.

*(Current Implementation:* The LCN data structure and basic semantic embedding functions are functional. For example, TORI can already ingest code and produce a network of concepts and simple similarity links. However, the full closed-loop where the spectral reasoning informs updates to the concept graph is not yet realized. Completing this closed-loop reasoning pipeline is an immediate objective.)

**Language & Domain Integration Layer**

This layer bridges raw inputs (source code, natural language text, etc.) with the ALAN core’s conceptual format. It ensures that no matter what form information comes in – be it a Python file, a snippet of our custom DSL, or an English sentence – it gets translated into the *concepts* that the core can work with, and likewise translates the core’s conceptual outputs back into human-readable answers (code suggestions, explanations, etc.).

**Programming Language Frontends (for TORI IDE):** TORI IDE supports multiple programming languages through dedicated frontends. For each supported language, the frontend includes a parser or utilizes an existing compiler/AST generator to produce an abstract syntax tree (AST). The AST is then traversed and converted into updates to the LCN. For example, when you feed a Python file into TORI, the Python frontend identifies structures like classes, functions, and variables and their relationships (function A calls function B, Class X inherits from Class Y, etc.). Each of these constructs maps to concept nodes in the LCN (with edges such as “Function *encrypt* calls Function *hash*” or “Class *PaymentService* implements Interface *Service*”). The integration layer also extracts any semantic metadata (docstrings, type annotations) to attach to the concept nodes as needed.

In the current implementation, **Python** and our custom **ELFIN** DSL are first-class citizens: they have the most complete parsers and mapping logic. C/C++ and JavaScript/TypeScript are supported in a secondary manner – using external tree-sitter parsers or language server protocols to get an AST, which is then mapped into the concept graph in a generic way. (In other words, TORI can ingest C++ code, but the depth of understanding is currently limited compared to Python/ELFIN where we have custom semantics. Expanding language support will be important post-v1.0.)

**Domain-Specific Ontologies (for TORI Chat & Enterprise):** For TORI Chat, especially in enterprise settings, the integration layer can ingest domain knowledge such as ontologies, taxonomies, or glossary files and incorporate them into the LCN. For instance, a financial institution could feed in an ontology of banking terms, or a medical user could input a set of medical protocols. The integration layer will convert those into concept nodes and edges so that the chat’s reasoning core becomes aware of domain-specific concepts and their relationships. We refer to these packages of domain knowledge as **“concept packs.”** For example, if a *Finance Pack* is loaded, TORI Chat will know that concepts “KYC check” and “AML regulation” are related compliance ideas and should be treated accordingly during reasoning. This greatly enhances the system’s expertise in specialized areas, essentially bootstrapping the LCN with domain-specific subgraphs.

**Natural Language Processing (for TORI Chat):** When a user enters a query or message in TORI Chat, the NLP component of the integration layer processes it. This involves standard steps: tokenization, syntactic parsing (identifying the grammatical structure), and semantic interpretation. Key phrases or entities in the user’s input are mapped to concept nodes in the LCN (either matching existing ones, or creating new nodes if a novel concept is mentioned). For example, if a user asks, “How does the sorting algorithm in my code work?”, the NLP will identify “sorting algorithm” as an entity. If the IDE side has already indexed a concept for the sorting function or algorithm from the code, the question is linked to that existing concept node. If no such concept exists yet, TORI may create a placeholder concept “sorting algorithm (unknown)” which an agent could then try to resolve by searching the code or documentation. Conversely, when TORI Chat formulates a response, the integration layer takes the reasoning output (which might be an abstract argument or a subgraph of concepts that form the answer) and generates fluent natural language from it. We use template-based or controlled language generation to ensure the answer is faithful to the underlying reasoning (rather than a free-form neural generation which might hallucinate details). For instance, if TORI’s reasoning finds that *Function X* is slow because *it uses a quadratic loop*, the answer might be generated as: “Function X is identified as a bottleneck because it uses a nested loop, leading to O(n^2) complexity, which slows down performance.”

**Bidirectional Sync:** A critical aspect of this layer is that integration is *bi-directional* and continuous. As the user updates code in the IDE, those changes stream into the ALAN core via the language frontend (e.g., on each file save or even keystroke, for live analysis). The concept graph and oscillator states update in near real-time, so TORI’s understanding stays in sync with the codebase. Likewise, for Chat, as the conversation progresses, new concepts introduced are added to the LCN, and any conclusions drawn can be fed back (for example, if TORI solves a problem in chat, that knowledge can become a concept node which the IDE might use as a hint or comment in code). This ensures TORI IDE and Chat remain *synchronized* through the shared core knowledge.

*(Current Status:* The Python code parser is **mostly operational** (using AST from ast module plus custom logic to populate the LCN). The ELFIN DSL parser is **in progress** with initial grammar and concept mapping defined, but not fully integrated. Other languages (C/C++, JS) rely on external parsers and have basic mapping; these are less complete. The natural language pipeline for TORI Chat exists (leveraging a spaCy-based parser and simple entity linker), but it’s rudimentary and will need improvement for nuanced understanding and generation. Enhancing the NLP, especially for complex multi-sentence queries and fluid dialogue, is on the roadmap.)

**The ELFIN DSL: A Domain-Specific Language for Cognitive Coding**

One unique element in TORI’s language integration is the inclusion of a custom *domain-specific language* called **ELFIN**. ELFIN is an experimental DSL being developed alongside TORI to explore and demonstrate **concept-first programming** — a style of coding where the fundamental units are abstract concepts rather than concrete syntax. The name “ELFIN” evokes something small and magical, hinting at its role: it is designed to be a concise language that nonetheless wields the “magic” of TORI’s cognitive engine.

**Purpose of ELFIN:** Why create a new programming language? ELFIN serves several purposes. First, it acts as a **testbed** for TORI’s cognitive features. Because we control the language design, we can ensure that ELFIN’s syntax and semantics map cleanly onto the LCN. This allows us to push the limits of concept-based analysis — ELFIN programs can encode high-level intents (for example, one could imagine an ELFIN construct for “goal” or “assumption”) that TORI will directly capture as concept nodes with specific relationships (like a goal node linked to steps to achieve it, or an assumption node linked to where it’s used). In essence, ELFIN is a language built *for* a cognitive OS, meant to express not just computations, but the reasoning behind them.

Second, ELFIN provides a **vehicle for features** that would be hard to integrate into existing languages. For instance, we can experiment with syntax for declaring concept relationships explicitly (perhaps an ELFIN statement to declare two functions conceptually analogous, or to tag why a decision was made). These might look like comments or annotations in other languages, but in ELFIN they can be first-class citizens. This enriches the concept graph beyond what we could infer from standard code.

Third, ELFIN ensures that TORI is not tied to the quirks of any single legacy language. It’s forward-looking: as we improve ALAN’s reasoning, we may extend ELFIN to naturally expose those capabilities. One could envision ELFIN being used for writing *AI workflows* or *agent scripts* for TORI itself. For example, an advanced use-case might be an ELFIN script that describes a development task (in quasi-natural language, but formalized) which TORI can then execute or assist with via its agents. In this sense, ELFIN blurs the line between code and specification, providing a language for *metaprogramming the AI*.

**Integration State:** Currently, the ELFIN language is in an **early stage**. A grammar has been defined and a parser implemented (the elfin\_lang.proto definition and a parser generator were started). Basic ELFIN programs – e.g., simple function definitions and concept declarations – can be parsed into the concept graph. The core concepts of ELFIN (like functions, invariants, or relationships) are mapped to LCN nodes similarly to how Python constructs are mapped. However, more advanced features (and the full runtime semantics) are not finished. For now, ELFIN code doesn’t “run” in the traditional sense; its value is in populating the concept network. We have integrated some initial ELFIN scripts into tests to ensure TORI can digest them. For example, an ELFIN snippet that defines a high-level plan (as a series of steps with goals) successfully creates a chain of concept nodes that the reasoning core can use to guide the Refactoring Agent. But this is a prototype demonstration. Before TORI’s launch, we plan to finalize the ELFIN spec for the most crucial features and ensure the integration layer can handle round-trip conversion (editing ELFIN in the IDE and getting meaningful assistance, much like with Python).

**Link to Concept Graph:** ELFIN is, by design, tightly coupled with the concept graph. Every ELFIN language construct has a direct representation in the LCN. If a developer writes an ELFIN function that “describes” a process (instead of explicitly coding it), TORI will create concept nodes for the abstract process, any sub-tasks, and link them according to the ELFIN syntax. Think of ELFIN as writing pseudo-code that is *immediately semantic*. For example, an ELFIN construct might allow:

javascript

Copy

assume User is\_authenticated

function transferFunds(amount) {

require amount <= Account.balance

ensure Account.balance\_decreased\_by(amount)

//...

}

In this pseudo-ELFIN example, assume and require/ensure might be ELFIN keywords. TORI would map this into concepts like *User (is\_authenticated)* as a fact, *transferFunds* as a function concept, a precondition concept linking *transferFunds* to the requirement on *amount and balance*, and a postcondition concept linking to the balance decrease effect. These become part of the LCN, meaning TORI “knows” the intent and constraints of the code, not just its syntax. During reasoning (say the Test Generator Agent or a bug-finder), TORI can use these concepts to reason about correctness (the assume/require are like formal specs in the concept space).

By linking ELFIN directly into the concept graph, we also make it possible for TORI to explain code behavior in higher-level terms. If part of the system is written in ELFIN, TORI Chat could potentially answer questions about it by directly citing the concepts that ELFIN introduced (e.g., “The transferFunds function ensures post-condition that the balance is reduced by the transferred amount”). In short, ELFIN serves as both a programming language *and* a knowledge encoding scheme for TORI’s AI.

*(Outlook:* While ELFIN will be included in the TORI v1.0 release as an experimental feature, its true potential lies beyond the initial launch. It could evolve into a powerful tool for users to extend TORI’s intelligence, by encoding new knowledge or procedures in a form the cognitive core can inherently work with. In the long run, success of ELFIN will be measured by whether it attracts real usage; for now, it’s a strategic investment to showcase TORI’s concept-oriented philosophy and to ensure we have a maximal internal test of the system’s capabilities.)

**User Interface & Tools (Thoughtboard IDE and Chat UI)**

The User Interface layer encompasses all the front-end components through which users interact with TORI’s intelligence. This includes the **TORI IDE interface (Thoughtboard)**, various panels and visualizations within the IDE, and the **TORI Chat interface** for conversational interactions. A core principle of the UI is to **augment familiar workflows with cognitive superpowers**: developers should feel at home with the basic tools (editor, console, etc.), enhanced subtly and powerfully by TORI.

**TORI IDE – “Thoughtboard” Interface:** Rather than piggybacking on an existing IDE via plugin, we have created an independent, local-first application for TORI IDE, code-named **Thoughtboard**. Thoughtboard is a custom UI (built with a modern tech stack, currently a Python/Electron prototype with plans for a polished cross-platform app) that serves as the primary workspace for the developer. It presents the user’s code in a code editor canvas but layered with TORI’s cognitive insights. Key UI elements include:

* An **editor with inline annotations**: as you write or browse code, TORI can underline sections that have conceptual issues (e.g., out-of-phase functions) or insert subtle markers where an agent has a suggestion. For example, a faded lightbulb icon might appear next to a function that TORI believes could be refactored, and hovering reveals the suggestion.
* A **Concept Graph Explorer**: a sidebar or pop-up view that visualizes the local portion of the concept network related to the current context. If you click on a function, the explorer might show connected concepts like “uses algorithm X” or “related requirement: Y” in a node-link diagram. This allows developers to navigate code semantically, not just by filenames.
* **Spectral Debugging Dashboard**: a panel available during runtime or analysis that displays the oscillator phases and other dynamic metrics. For instance, it could show a live chart of module synchronization – if one module’s oscillator falls out of sync (signaling a potential bug or mismatch), the dashboard highlights it. This is akin to a performance monitor but for conceptual consistency and phase alignment.
* **Project Insights & Warnings**: Much like how an IDE shows compile errors or linter warnings, TORI IDE (Thoughtboard) shows cognitive warnings. For example, “Module *AuthService* is conceptually out-of-phase with Module *UserService* (they interact but have inconsistent assumptions)” or “Concept drift detected: the current code usage of concept *X* differs markedly from past usage – consider reviewing for unintended changes.” These come from the ALAN core’s deeper understanding. Each warning or insight is explainable – the UI lets the user click to see the rationale (which agent or rule triggered it, and what concept relations are involved).
* **Standard IDE features, enhanced by AI:** Thoughtboard also integrates version control, testing, debugging, etc., with AI support. For instance, during debugging, if a test fails, TORI can cross-reference the failing test’s concept footprint with recent code changes and *immediately* suggest which change likely caused the regression (because it knows which concept nodes were affected by the change and are linked to that test). When searching code, instead of simple text search, the user can do a *conceptual search* (“find all code related to user authentication”) and TORI will return results across files and languages, because it knows what parts of the codebase pertain to that concept.

All these features are delivered in a cohesive UI. The design philosophy is to not overwhelm the user – TORI’s help should feel like a natural extension of the IDE. A developer can use Thoughtboard in a minimal way (just as a code editor with some extra info, ignoring most cognitive panels), or dive deep into TORI’s new capabilities as needed. The UI is kept responsive and **local-first**: all analysis is done locally by the ALAN core, so these features work in real-time without cloud latency.

*(Current UI Status:* A prototype of the Thoughtboard app exists. It currently uses a VS Code-like layout for familiarity – a file explorer, editor pane, and an output/terminal pane – but augmented with a concept graph panel and debug phase panel in experimental form. This prototype is functional for basic editing and displaying dummy AI annotations. The **VS Code extension scaffold** that was used early on has been phased out in favor of the standalone Thoughtboard app to ensure a tighter integration and offline-first operation. The UI is not yet feature-complete or polished: for example, the concept graph viewer works (it can show the graph from the last analysis in a separate window) but is not interactive in real-time. Over the next development sprint, a major goal is to flesh out the Thoughtboard UI: integrate the live concept explorer, make the debug phase panel truly live with the oscillator data, and implement all the needed controls and polish. By launch, the IDE should feel like a modern, slick environment where the cognitive features are intuitive and valuable.)

**TORI Chat Interface:** TORI Chat’s UI will allow users to converse with the cognitive engine, either in standalone mode or integrated within Thoughtboard. For v1.0, our plan is to provide **two interfaces** for TORI Chat:

1. **Integrated Chat Panel in Thoughtboard IDE:** This is a panel or sidebar where the user (likely a developer) can ask questions about the code or get assistance. For example, a developer can highlight a block of code and ask in the chat panel, “How does this function work?” or “Find potential bugs in this logic,” and TORI Chat (leveraging the IDE’s context) will respond. This tight integration means the chat has full access to the project’s concept graph and can act as an in-IDE assistant.
2. **Standalone Chat Application:** In addition, we envision a dedicated chat client (likely web-based or Electron-based) for TORI Chat, targeting more general usage or enterprise deployment. This standalone TORI Chat app would function like a powerful AI assistant that can be run on-prem. It would have features like multi-turn conversation memory, persona selection (e.g., you could talk to “TORI the Python expert” or “TORI the Project Manager assistant”), and tools for the user to upload documents or knowledge packs in-chat.

*(Current UI Status:* At present, the chat UI is minimal. We have a basic web page that allows inputting a query and seeing the text response from the backend – mainly used for testing the back-end. It’s essentially a simple chat log with no advanced UI elements. There’s no rich formatting, and features like conversation history beyond a scrollback are rudimentary. During development, initial focus will be on the integrated IDE chat panel (since that directly assists developers alongside the IDE). The standalone app will be scaffolded – likely leveraging some components of the Thoughtboard UI for consistency. Voice input, persona visualization (e.g., an indication of which agent is “speaking” in a multi-agent answer), and other enhancements are slated for later development once core chat functionality is solid. The key for v1.0 is to ensure TORI Chat can handle multi-turn Q&A reliably and that enterprise users can deploy it securely on their own machines with an interface that, while basic, is usable and polished enough for demo purposes.)

**Multi-Agent Orchestration Layer**

One of TORI’s most innovative aspects is its **multi-agent architecture**. Instead of a single monolithic AI trying to do everything, TORI employs multiple specialized AI agents, each with a focused role or expertise, coordinated by a central **Orchestrator**. This approach is analogous to Marvin Minsky’s “Society of Mind” theory, where intelligence emerges from the interaction of many simple agents[en.wikipedia.org](https://en.wikipedia.org/wiki/Society_of_Mind#:~:text=In%20his%20book%20of%20the,2). By having a team of AI sub-components, TORI can tackle complex tasks more robustly: each agent contributes its perspective (coding best practices, testing, high-level strategy, etc.), and the orchestrator mediates their interactions, much like a project lead coordinating team members.

**Orchestrator:** The orchestrator is a supervisory module within ALAN’s core that listens to inputs and events (new code changes, a user question, an inconsistency detected) and decides which agent(s) to invoke, and how to merge their results. It operates in sync with the phase logic — meaning it can use the oscillator network to detect when agents disagree (if their suggested concepts are out-of-phase) and resolve conflicts. The orchestrator ensures the agents work in concert rather than at cross-purposes. If two agents produce conflicting suggestions (e.g., one agent suggests renaming a function while another suggests deleting it), the orchestrator, guided by global rules or meta-agents, will decide on a resolution or ask for clarification from the user.

**Agents in TORI IDE:** In the IDE context, we have several specialized agents either implemented or planned:

* **Refactoring Agent:** Scans the code’s concept graph for code smells or refactoring opportunities. For example, if it finds two separate modules implementing similar concepts, it suggests an abstraction or utility. It also monitors complexity metrics via the concept graph, prompting the user to simplify a function that grows overly complex. *(Status: partially implemented – can detect simple duplications, though deeper conceptual duplications use placeholder logic.)*
* **Debug Advisor Agent:** Watches program execution traces and the oscillator signals during runs to spot likely causes of errors or performance issues. For instance, if during a test run, memory-related concept nodes spike in activity, it might infer a potential memory leak or inefficiency and point the developer to it.
* **Documentation Agent:** Ensures that documentation in the concept graph stays updated with code changes. It can draft docstrings or explanations for new code by synthesizing information from similar past concepts and commit messages. If a developer highlights a function and asks “explain this,” this agent works with the chat subsystem to provide an answer drawn from the concept context. *(Status: currently stubbed – it creates placeholder documentation nodes but doesn’t generate text yet; integration with an LLM for language generation is planned but to be done carefully under the deterministic core principle.)*
* **Test Generator Agent:** Analyzes the concept graph to find untested requirements or edge cases. It proposes new unit tests for parts of the code that lack coverage or identifies when a change in one concept should trigger re-testing of related concepts. For example, if a new concept “encryption” was added, it might suggest tests for that, or if a concept has changed, ensure all dependent concept tests are revisited.

These agents operate continuously or on triggers (like after a code analysis pass). They post their suggestions back into the system as structured data – e.g., the Refactoring Agent might add a “Suggestion” node linked to the functions in question, which the UI then surfaces as a tip to the user.

**Agents in TORI Chat:** The chat context also benefits from specialized agents:

* **Knowledge Retrieval Agent:** When a user asks a question, this agent fetches relevant information from the LCN (and potentially allowed external sources) to gather facts. It’s like a librarian ensuring that if the answer is in the docs or code, TORI finds it.
* **Logic/Reasoning Agent:** This agent attempts to perform multi-step reasoning for complex queries, using the knowledge assembled. Essentially, it chains together pieces of knowledge (possibly using the oscillator network to maintain coherence) to form an answer. In some implementations, this could be a rule-based inference engine or a constraint solver.
* **Persona/Style Agent:** For TORI Chat, especially if responding as different personas (tutor vs. devops assistant, for example), this agent modulates the response style, tone, or perspective. It ensures the answers align with the selected persona or context (e.g., enterprise compliance vs. casual helper).

Beyond these, we have plans for additional agents to broaden TORI’s reasoning scope:

* **MPorter (Strategy Agent):** *Planned.* This is a high-level strategy and planning agent, humorously named in homage to Michael Porter (known for strategic frameworks). The MPorter agent’s role is to reason about overall objectives and plans. In the IDE, it might watch the sequence of development tasks and guide the order of operations (for instance, suggesting “Before refactoring X, address the failing test Y”). In Chat, it could maintain the long-term direction of the conversation or break down a complex query into sub-goals for other agents to solve. Essentially, MPorter thinks about “the big picture” — ensuring TORI’s various actions are aligned with the user’s overarching goals and making strategic choices if there are trade-offs. This agent brings a sort of executive function or meta-reasoner into the system, planning multi-step solutions.
* **Attachment (Emotional Intelligence) Agent:** *Planned.* This agent focuses on the emotional and relational aspect of interactions, drawing from concepts in affective computing. It monitors the user’s tone and sentiment (e.g., frustration, confusion, satisfaction) and the relationship context. In TORI Chat, the Attachment agent would adjust responses to be empathetic and supportive – for example, if a user seems frustrated by errors, the agent might suggest a more reassuring explanation or a gentle prompt to take a break. It can also remember personal preferences or emotional cues (almost like how a good human assistant remembers what frustrates their boss, or what style of feedback they prefer). In the IDE, this agent might manifest as the system “knowing” when to intervene: e.g., if a user has tried to solve a problem multiple times (detected by rapid code changes and undos), the agent might proactively offer help (“It looks like you’re encountering difficulties with this function. Would you like some suggestions?”). The Attachment agent essentially gives TORI a form of **emotional intelligence**, aiming for the AI to not just be correct, but also helpful and attuned to the user’s state[en.wikipedia.org](https://en.wikipedia.org/wiki/Affective_computing#:~:text=Affective%20computing%20is%20the%20study,interpret%20the%20emotional%20state%20of).

All these agents produce outputs that must be combined into a single, coherent assistance experience. The **Multi-Agent Orchestrator** manages this. It can run agents in parallel or sequence as needed. For example, when a new code analysis is done, the orchestrator might run the Refactoring, Test, and Documentation agents in parallel to gather all their findings, then integrate those findings (resolving any conflicts) before presenting to the user. In chat, orchestrator might orchestrate a debate between a “strict expert” agent and a “simplifier” agent if a question needs both perspectives, ultimately merging their answers.

*(Current Status:* The multi-agent system is implemented in a rudimentary form. In the IDE prototype, there are hooks for a Refactor and Debug agent that run on each code analysis cycle, but their coordination is basic (essentially a priority rule: refactor suggestions get logged first, etc.). The orchestrator logic that uses oscillator synchronization to mediate agents is not yet fully realized. We have simulated multi-agent reasoning in chat using a pipeline (knowledge retrieval → reasoning → answer formatting), but it’s not truly concurrent agents. During the next development phase, implementing the orchestrator’s conflict-resolution and expanding the agent roster (especially integrating MPorter and the Attachment agent logic) will be crucial. The end goal is that by Day 15 of the sprint, as scheduled, the multi-agent enhancements will allow TORI to handle conflicting suggestions and leverage multiple specialized agents seamlessly.)

**Why Multi-Agent?** This design is motivated by both technical and cognitive factors. Technically, it’s easier to maintain and improve specialized agents one by one than a giant AI that does everything. New agents can be added as plugins to increase functionality without overhauling the whole system. Cognitively, it mirrors how humans solve problems: we subconsciously employ different modes of thinking (analytical, recall-based, emotional judgement) and reconcile them. Indeed, the Society-of-Mind approach suggests intelligence can emerge from a colony of simple processes[en.wikipedia.org](https://en.wikipedia.org/wiki/Society_of_Mind#:~:text=In%20his%20book%20of%20the,2). By architecting TORI as a society of agents, we aim for a system that is **robust, extensible, and interpretable** – we can trace which agent contributed to which decision, and agents can even explain their reasoning steps (e.g., the Test agent could say “I suggested this test because I noticed concept X wasn’t covered by any existing tests”).

**Development Roadmap – 21-Day Sprint Plan (Starting May 12, 2025)**

To bring TORI from its current prototype state to a production-ready v1.0, we have outlined a rigorous 21-day development sprint for **TORI IDE** and a parallel 21-day sprint for **TORI Chat**. Each plan assumes **Day 1 is May 12, 2025 (Monday)**, and is structured as three focused work weeks with periodic review and buffer days. Below, time-based references (Day 1, Day 2, …) are annotated with their calendar dates for clarity.

**Note:** The sprint plans below presume that foundational work on the partially implemented core components (oscillator sync, Koopman spectral engine, ELFIN DSL parser, and orchestrator logic) is completed or nearly completed on Days 1–3. These core tasks are emphasized at the start of the IDE sprint. Without them, subsequent feature development would be building on unstable ground. Thus, the first week’s milestones heavily focus on finalizing the core.

**TORI IDE – 21-Day Sprint Plan (May 12 – June 1, 2025)**

**Sprint Goal:** Transform the TORI IDE from a promising prototype into a polished, usable product (v1.0) with all core features implemented and all placeholder components replaced. We allocate 21 days of effort (3 weeks) with one rest/review day each week (Day 7 and Day 14 as lighter days). The plan ensures continuous integration of features and frequent testing.

1. **Day 1 (May 12, 2025) – Sprint Kickoff & Core Setup**  
   *Goal:* Kick off development and ensure the team is aligned and the environment is ready. Also begin integration of core components.  
   *Tasks:*
   * Conduct a project kickoff meeting to review this Master Plan, clarifying the vision, architecture, and critical tasks for the sprint.
   * Finalize the development environment setup for all team members. This includes ensuring everyone can run the existing TORI IDE prototype (both backend and Thoughtboard UI). Resolve any environment issues (path configs, dependencies) so that development can proceed uniformly.
   * Merge any outstanding prototype code into the main branch. (For example, if there were separate branches for the new oscillator sync algorithm or the UI refresh, bring them in now.) We want a single code baseline at the start.
   * **Begin core integration:** If the advanced oscillator sync and spectral analysis components have code ready in prototype form, integrate them behind feature flags. This means plugging in the new Banksy oscillator update loop and Koopman analysis module, but perhaps toggled off until fully tested. This sets the stage for Day 2 and 3 where we’ll flesh them out.  
     *Checkpoint:* By end of Day 1, TORI IDE should run end-to-end in at least one environment with the new core components in place (even if dormant). All developers have a working setup. We have a clear list of the core integration subtasks to tackle in the next two days.
2. **Day 2 (May 13, 2025) – Core Engine Focus: Oscillator Synchronization Implementation**  
   *Goal:* Implement the **Banksy phase-coupled oscillator logic** in the ALAN core (replacing the placeholder with the real synchronization mechanism).  
   *Tasks:*
   * Design and document the updated synchronization algorithm using our ψ-based approach. This includes how oscillators will influence each other’s phase differences and what parameters (coupling strengths, frequencies) are needed. (We draw from known models like Kuramoto for inspiration but incorporate spectral feedback – for instance, use the Koopman-derived modes to adjust coupling dynamically.)
   * Begin coding the oscillator update function in the alan\_core module. Each cycle (or event trigger), this function will update all concept oscillators’ phases. Implement coupling: if two concepts are linked, their phase difference should adjust gradually (converging if harmonious, diverging if conflict signals).
   * Write unit tests for the oscillator update logic. Create a simple scenario: two concept nodes A and B with a link. Simulate a phase update where A is perturbed; verify that B’s phase moves closer to A’s over several iterations if the link indicates they should sync.
   * If time permits, integrate a debug command or log that prints out phase values for a test graph to visually verify behavior.  
     *Checkpoint:* By end of Day 2, the oscillator subsystem produces **non-trivial output** – i.e., phases are no longer static or random; they move over time in response to concept links. We should be able to demonstrate that if concept A and B are linked, their oscillator phases trend toward alignment (or an intended offset). All placeholder “mock oscillator” code is removed or replaced. The Kuramoto mention in code is replaced with our new algorithm (e.g., functions now named update\_phase\_sync instead of update\_phase\_dummy). This establishes the dynamic heartbeat of the cognitive core.
3. **Day 3 (May 14, 2025) – Core Engine Focus: Spectral Analysis & Prediction**  
   *Goal:* Flesh out the **Koopman morphic engine** to utilize concept graph dynamics for prediction and pattern recognition. In short, get the spectral analysis core working with real data.  
   *Tasks:*
   * Implement a basic spectral analysis pipeline: take snapshots of the concept state (e.g., concept activation levels or phase angles at intervals) and perform a simplified Koopman analysis. Concretely, start with something manageable like computing a Fourier transform of a few recent states to see dominant frequencies of change. If feasible, integrate a linear algebra library to compute eigenvalues of a state-transition matrix we define (for a simplified model of concept evolution).
   * Connect this analysis to the system: after each oscillator update (from Day 2’s work), run the spectral analyzer on the latest state. For now, maybe use a sliding window of the last N states. Identify if any eigenvalues/modes stand out (for example, a mode that’s not converging).
   * If full Koopman is too heavy to implement in one day, implement a *proxy*: e.g., track simple trends like “concept X has been steadily increasing in activation for 5 cycles” and flag that as a potential trend prediction (which could mean “we might see concept X trigger something soon”). The key is to get some predictive capability in place, even if rudimentary.
   * Add instrumentation: have the system log any predictions or detected patterns. For instance, “Koopman analysis: mode corresponding to concept ‘Y’ shows periodic oscillation (likely repeating pattern every ~10 commits).” This is mainly for development/testing now.  
     *Checkpoint:* By end of Day 3, the ALAN core should not only update state (from Day 2) but also **analyze state**. We should be able to point to a function or module where given some concept state history, we get a spectral decomposition or equivalent output. There should be no more hard-coded dummy returns in the analysis; even if it’s simplistic, it’s real. The core inference pipeline (concept update -> oscillator sync -> spectral check) is now implemented end-to-end. We can say: “The cognitive core’s major algorithms are in place.” Any issues or limitations encountered are noted to address in optimization days later.
4. **Day 4 (May 15, 2025) – Core Stabilization & Refinement**  
   *Goal:* Polish the core features implemented in Days 2–3, ensure they are stable and integrated. Address any major bugs or inefficiencies in the new core logic.  
   *Tasks:*
   * Review and refactor the oscillator and spectral code for clarity and performance. Since these will run frequently, ensure they are optimized (e.g., using NumPy for vectorized updates of phases instead of Python loops).
   * Add **Lyapunov stability checks** into the core loop if not already: for example, after updating oscillator phases, compute a simple metric of system stability (perhaps total phase variance or similar). Use this to log if the system is trending unstable. This will help later when multiple agents produce changes.
   * Write additional tests for edge cases: e.g., no links between concepts (oscillators should remain as-is), or all concepts fully linked (they should eventually sync up). Also test the spectral analyzer with known input signals (maybe feed it a sine wave and see if it identifies the frequency correctly) to verify it works as expected.
   * Integrate core outputs with a debug interface: perhaps extend the existing CLI or admin panel to display current concept phases and any detected modes. This helps the team manually inspect that the core is behaving plausibly.
   * If any core feature is not working or too noisy (for instance, if the spectral analysis is giving random “predictions”), decide on whether to tweak or gate it behind a feature flag so it doesn’t interfere with the rest of development. It’s okay if some of these advanced aspects are quiet in the background until we fine-tune them.  
     *Checkpoint:* By end of Day 4, the ALAN 2.x core should be *stable*: running it on a representative project or conversation does not crash, and it provides consistent outputs. The team should have confidence that this foundation is ready for layering on the IDE and Chat functionality. The “core to-do list” should now be mostly done, allowing focus to shift to user-facing features from Day 5 onward.
5. **Day 5 (May 16, 2025) – Thoughtboard UI Integration with Core**  
   *Goal:* Connect the updated ALAN core to the Thoughtboard UI, so that core insights (concept graph, warnings, etc.) start flowing to the front-end. Begin implementing interactive UI features that use the core data.  
   *Tasks:*
   * Hook up the **Concept Graph Explorer** panel in the UI to the actual LCN from the core. Replace any fake/demo graph data with real data coming from the backend via an API call. If not already done, create an API endpoint (or use existing one) like /concept\_graph that returns the current concept graph in JSON or protobuf form. The front-end should parse this and render the graph. Test by loading a sample project and opening the concept explorer – verify that it shows nodes corresponding to real classes/functions from that project.
   * Integrate **cognitive warnings** into the editor gutter. The backend (core or agent layer) should now be capable of producing at least basic warnings (e.g., from the Refactoring agent stub or from static checks). Define a data structure for these (if not done): say, a list of issues each with a type, message, and location (file/line or concept reference). Implement a mechanism for the UI to fetch or receive these. Possibly, whenever the core does an analysis pass, it could emit events that the UI listens to via a websocket or polling. Implement the simplest thing (maybe polling every few seconds or on demand). Then display these in the UI: e.g., highlight a line with a lightbulb icon where a suggestion exists. Clicking it could open a panel with the suggestion details.
   * Ensure the **Phase Dashboard** in the UI can show live data. This might involve a simple graph plotting library in the UI. For now, we can feed it a subset of the oscillator info (maybe just show relative phase offsets among major components). Hook it up to the backend: e.g., an endpoint /phases returns current phase values for top-level components. Update the UI at an interval. The goal is to replace placeholder sine wave graphics with actual output from our Day 2 core.
   * UI/UX fixes: as we integrate, likely some UI adjustments are needed (e.g., if the concept names are long, ensure the graph labels handle that, etc.). Tweak CSS or layout as necessary for a clean presentation.  
     *Checkpoint:* By end of Day 5, the Thoughtboard UI should be **functionally connected** to the AI core. We should be able to see real analysis reflected in the interface: for example, open a code file and if there’s a conceptual inconsistency, a warning appears in the UI coming from the core logic. The concept graph viewer should show at least a rudimentary graph of the code. Essentially, the front-end and back-end are talking to each other meaningfully, turning TORI IDE into a truly interactive cognitive IDE (even if the visuals are still rough).
6. **Day 6 (May 17, 2025) – Multi-Agent System Activation (IDE context)**  
   *Goal:* Enable and test the multi-agent framework within the IDE now that the core and UI are connected. Flesh out or refine a couple of key agents (Refactoring, Test generator) so they start providing useful output.  
   *Tasks:*
   * Integrate the **Refactoring Agent** fully: Now that the core is stable, revisit the Refactoring agent code. Improve its logic beyond the trivial duplicate check. For example, implement a check for functions that are conceptually very similar (perhaps using the embedding similarity: if two function concept vectors are very close, suggest factoring them). Also implement a check for any function out-of-phase with others (using the oscillator data: if one function’s oscillator is lagging significantly, maybe it’s not updated with new logic). These conditions can generate suggestions (e.g., “Function X might be outdated relative to related Function Y”).
   * Turn on the **Test Generator Agent** in a basic form: have it analyze the concept graph for any concept that has no associated test concept. If found, it can log a suggestion like “Consider adding tests for module Z; no tests found.” This is simplistic but sets the stage. If possible, integrate with actual test files (e.g., if code is Python, check if there’s a test module for each code module).
   * Implement the Orchestrator’s simple conflict resolution: if two agents try to tag the same code with different suggestions, determine how to handle it. Perhaps decide on a priority (Refactoring suggestions might be shown separately from Testing suggestions). In UI, maybe categorize suggestions by agent.
   * Ensure the UI displays multi-agent output distinctly if needed. Possibly add a filter or label to suggestions (“[Refactor] You have duplicate code in X and Y” vs “[Test] No tests cover module Z”). This transparency aligns with our explainability principle.
   * Start a log or telemetry of agent activity for debugging (e.g., console log: “RefactorAgent: checked 120 concepts, made 2 suggestions; TestAgent: made 1 suggestion.”). Useful for performance tuning later.  
     *Checkpoint:* By end of Day 6, multiple agents should be **active and contributing** in TORI IDE. A developer using the IDE at this point would see, for instance, a refactoring tip or two pop up after analysis, and maybe a note about tests. It need not be perfect or complete, but the multi-agent system is now live. We should also verify that their suggestions are sensible and not too noisy, adjusting thresholds if needed.
7. **Day 7 (May 18, 2025) – Review, Testing & Buffer**  
   *Goal:* Mid-sprint review of progress; fix any accumulated bugs from week one; ensure we haven’t broken basic functionality. Use this as a buffer/rest day.  
   *Tasks:*
   * Run a full **integration test** of TORI IDE as it stands. Open a sample project, use the main features: editing, viewing suggestions, triggering analyses. Note any crashes, UI glitches, or obviously incorrect suggestions.
   * Fix high-priority bugs discovered. For example, if the concept graph fails to update after certain edits, address that. Or if an agent suggestion is clearly wrong due to a logic bug, correct it.
   * If some planned tasks from Days 1-6 slipped, use today to catch up (buffer). For instance, if the Koopman analysis wasn’t fully integrated by Day 4, finish it now.
   * Solicit feedback from any team members or early testers if available. Sometimes a fresh pair of eyes on the UI or behavior can spot issues.
   * Take a breather to document any new technical debt or tasks that arose. Update the plan if needed for week two.  
     *Checkpoint:* End of Day 7 should have a **stable checkpoint build** of TORI IDE. Ideally, this is a point where if we had to do a demo of core capabilities, we could: concept graph viewing, a couple of cognitive suggestions, etc., all working. We also have a clear idea of what adjustments to make in the next phase.
8. **Day 8 (May 19, 2025) – Advanced IDE Features: Concept Explorer & Visualization**  
   *Goal:* Complete the implementation of the **Concept Graph Explorer** and other visualization tools in the IDE. Make interacting with the concept graph intuitive and useful.  
   *Tasks:*
   * Improve the **Concept Graph visualization**: Add interactive features such as zooming, panning, and clicking on nodes to reveal details. If not yet implemented, allow clicking a concept node to highlight the corresponding code in the editor (e.g., clicking a “User” concept node might highlight where the User class is defined or used).
   * Implement filtering in the graph view: for instance, toggle to show only certain types of relationships (maybe the user can filter to just “calls” relationships vs. “concept similarity” edges to reduce clutter).
   * Connect the graph view with the chat/QA: Perhaps allow the user to right-click on a concept in the graph and ask a question about it via TORI Chat (this sets the stage for cross-feature integration, though full chat integration comes Day 13).
   * If time, add a mini-map or hierarchical view for the concept graph (especially if the graph is large, a way to navigate it conceptually, like grouping by modules).
   * Ensure performance is okay: test on a larger project’s concept graph; optimize rendering or data fetching if it’s slow (e.g., implement lazy loading of parts of the graph).  
     *Checkpoint:* By end of Day 8, the **Concept Explorer** should be fully functional and user-friendly. This means a developer can open it, see a representation of their code’s concepts, and interact with it to understand their project’s structure better. It’s no longer a tech demo but a practical tool (even if it will be improved with polish later).
9. **Day 9 (May 20, 2025) – Agent Improvements & Orchestrator Logic**  
   *Goal:* Enhance the intelligence of the agents based on testing so far, and implement more sophisticated orchestrator behaviors (particularly conflict resolution and agent synergy).  
   *Tasks:*
   * Revise agent algorithms with insights gleaned from usage. For example, if the Refactoring agent frequently flags trivial issues, adjust its rules to be more meaningful (maybe require a higher similarity threshold for duplicate code suggestions, etc.). If the Test agent is too naive, incorporate more logic (e.g., look at coverage data if available, or at least ensure it doesn't repeatedly suggest the same test each run).
   * Implement agent communication through the orchestrator: design simple protocols where agents can pass hints to each other via the orchestrator. For instance, the Refactoring agent could mark a concept as “to be renamed” and the Documentation agent could pick that up and pre-prepare a new docstring. This could be simulated by having the orchestrator call agents in sequence and share a mutable context object.
   * Conflict resolution: If two agents produce contradictory suggestions (one says “remove function X” another says “improve function X”), determine a strategy. Possibly assign priorities or have the orchestrator present both suggestions but with context. Maybe integrate a simple rule: do no destructive suggestion (removal) if any other agent sees value in the item. Implement this in orchestrator logic.
   * Add at least one new small agent if possible to cover another aspect: e.g., a **Code Consistency Agent** that checks naming conventions or style (to show extensibility). Use it as a test of adding a new agent easily.
   * Continue to refine the **Attachment agent** (even in IDE, it can monitor user frustration by detecting rapid changes, etc.) though it’s more relevant in chat; maybe skip deep implementation here and focus on MPorter in IDE context. For instance, MPorter could monitor if multiple suggestions are pending and prioritize which one the user should handle first (a strategy element).  
     *Checkpoint:* End of Day 9 should see a **smarter multi-agent system**. Fewer irrelevant or conflicting suggestions should reach the user, thanks to orchestrator filtering. New synergies might be observable (perhaps documentation agent now auto-updates comments after a refactor suggestion is applied, etc., if implemented). We essentially have the multi-agent mechanism in a feature-complete state for the IDE side.
10. **Day 10 (May 21, 2025) – Performance Optimization & Scalability**  
    *Goal:* Ensure TORI IDE performs reasonably well on moderate-size projects and optimize any slow parts of the core or UI identified so far.  
    *Tasks:*
    * Profile the IDE on a sample large project (maybe a codebase with a few hundred files). Identify bottlenecks: perhaps concept graph generation is slow, or the UI rendering lags with too many nodes, or agent analysis takes too long.
    * Optimize critical paths: for example, if parsing a large project is slow, see if we can cache intermediate results or load incrementally. If the oscillator network update is taking too much CPU with many concepts, consider reducing frequency or using a more efficient algorithm/maths library (e.g., use numpy arrays for phase updates).
    * Memory usage check: ensure that concept graph storage is not ballooning. Implement pruning if needed (maybe archive parts of the concept graph not used actively, or ensure no memory leaks in our data structures).
    * Consider multi-threading or async improvements: The orchestrator and agents could potentially run in parallel threads since they often work on separate data. If Python GIL is an issue, maybe just ensure the UI thread is separate from analysis thread so UI stays responsive.
    * Optimize the UI rendering of the graph: use techniques like canvas/WebGL rendering for many nodes, or simplify the graph when off-screen.
    * Run automated tests for core algorithms to see if we can increase throughput (e.g., how many concept updates per second can we handle?). Aim for improvements and document the current capacity.  
      *Checkpoint:* By end of Day 10, TORI IDE should feel **snappier** and be able to handle larger inputs more gracefully. Perhaps we have metrics like “Initial analysis of 100-file project completes in X seconds” to gauge. Any optimization trade-offs (like reducing detail for speed) are noted and acceptable. We are now confident in the system’s performance for a demo or pilot usage.
11. **Day 11 (May 22, 2025) – TORI Chat Integration into IDE**  
    *Goal:* Embed TORI Chat capabilities into the IDE as a contextual coding assistant. This bridges the IDE and Chat products, allowing them to leverage each other.  
    *Tasks:*
    * Implement the **in-IDE chat panel** (if not done, or improve it): This panel should allow the user to ask questions about the code or get help. Connect it to the TORI Chat backend. Likely this means running the chat backend alongside the IDE backend, or as a library call to ALAN core in a different mode.
    * Context injection: Ensure that when the user asks something in the IDE’s chat panel, the Chat agent is aware of the current context (open file, selected code, etc.). Implement a mechanism to feed that context – e.g., prepend the query behind the scenes with a summary of the selected code or a reference to a concept node from the IDE.
    * Test queries: e.g., “Explain what this function does.” while a function is highlighted, or “Find any potential bugs in this file.” Verify the answers make sense and use the concept graph (the answer should ideally cite the reasoning, like “This function uses concept X which might be uninitialized, hence potential bug.”).
    * Make the chat output accessible to IDE actions: for instance, if TORI Chat in IDE suggests a code change (like “you could fix it by doing Y”), allow copying that suggestion to clipboard or even a one-click apply if feasible (this might be stretch, but at least plan for it).
    * Harmonize UI: The chat panel in Thoughtboard should have the same look/feel as the standalone chat (once designed). Also, ensure that using chat doesn’t interfere with the ongoing IDE analysis (they should share the ALAN core nicely – e.g., orchestrator should queue chat queries appropriately).  
      *Checkpoint:* By end of Day 11, a developer can seamlessly *talk to TORI from within the IDE*. This is a major milestone demonstrating the unified nature of TORI’s two halves. For example, one could highlight a piece of code and ask in the chat “How can I improve this?” and get a useful answer that draws on the code’s concepts. This dramatically shows off TORI’s capability to explain and assist in situ.
12. **Day 12 (May 23, 2025) – Enterprise Features & Hardening**  
    *Goal:* Add features particularly important for enterprise readiness of TORI IDE (and the integrated Chat), and harden the system for security and robustness.  
    *Tasks:*
    * Implement an **Audit Log** for AI suggestions and automated changes. Many enterprise users will want to track what the AI is doing. So, maintain a log (which could be a simple text file or a structured log) of every suggestion given, every auto-fix applied, every chat Q&A. Include timestamps and summary of reasoning or agent involved. This ties into transparency.
    * Add a **settings/configuration panel** for things like privacy controls (e.g., a toggle to ensure absolutely no external connection is ever made, guaranteeing all data stays local), and knobs for aggressiveness of suggestions (some companies might want only very conservative suggestions from the AI).
    * Implement authentication/permissions for multi-user scenarios (perhaps out of scope for v1.0 if single-user, but at least design how TORI would run in a shared environment or how an admin could limit certain features).
    * Security review: ensure any external inputs (like code being parsed, or chat queries) cannot lead to exploitation. For example, guard against code injection through concept pack files, or denial-of-service if the user writes something that intentionally worst-cases the analysis (we might set reasonable limits, like not analyzing files above a certain size in v1).
    * Hardening: make the system resilient to bad data. If the concept parser encounters a syntax it can’t handle (e.g., an unsupported language feature), it should fail gracefully (perhaps create a generic node or skip, rather than crash). Similarly for chat – if something goes wrong (like the reasoning times out), the UI should handle it (maybe respond with “I’m not sure about that” rather than hang).  
      *Checkpoint:* End of Day 12 should see TORI IDE approach a **production-ready posture** for enterprise trials. We have audit logs, some configuration ability, and the system has been combed for major security holes. While formal security testing is ongoing, we addressed obvious issues. This boosts confidence for enterprise demo on Day 16 and beyond.
13. **Day 13 (May 24, 2025) – TORI Chat Standalone Application**  
    *Goal:* Prepare TORI Chat as a separate deliverable (especially for enterprise use outside the IDE), ensuring it works as a standalone conversational assistant with access to “concept packs” and knowledge base.  
    *Tasks:*
    * Set up the **standalone Chat app** environment. If it shares a lot with the IDE, perhaps it’s the same backend but different front-end. Ensure we can run TORI Chat without the IDE UI – e.g., via a command-line interface or a basic web UI (which we have from prototype).
    * Expand the chat UI (web/Electron) to support multi-turn conversations nicely. Implement displaying the conversation history, and allow resetting context, etc. This UI can be simple but should be clean and user-friendly.
    * Test loading a domain **Concept Pack** in standalone chat. For instance, load a sample medical ontology and ask TORI Chat questions in that domain (“What is the procedure for XYZ?”) to see that it uses the injected knowledge. Fix any issues in concept pack integration (like concept collisions or memory usage).
    * Persona support: Implement a way to choose a persona or role for the chatbot. Maybe in the UI a dropdown “Emulate: [General Assistant, Python Guru, DevOps Coach, *Custom*]”. This will prepend an appropriate system message to steer the style. The Attachment agent’s work ties in here if, say, an “Empathetic listener” persona is chosen, the responses shift tone.
    * Ensure that the Chat system respects the privacy toggles (from Day 12 work) – i.e., verify that it indeed does not call out to any cloud API (we aren’t using any by design, but double-check things like spaCy or other libraries aren’t pulling data online unexpectedly).  
      *Checkpoint:* By end of Day 13, **TORI Chat standalone** should be ready for basic use. One should be able to launch the chat app, perhaps select a knowledge pack, and have a coherent conversation. While the IDE and Chat share a core, this independent packaging is important for demonstration to non-developer stakeholders and for enterprise environments where the chat might be deployed as a knowledge assistant on its own.
14. **Day 14 (May 25, 2025) – Multi-Agent Enhancements & Chat Agents**  
    *Goal:* Improve the multi-agent system in the Chat context, and ensure orchestrator synergy between Chat and IDE agents when needed. Use this day also as a secondary buffer/review point for the second week.  
    *Tasks:*
    * Turn focus to **Chat agents**: ensure the Knowledge Retrieval Agent and Reasoning Agent are working together properly. Perhaps implement a more advanced retrieval (like searching not just exact concepts but related ones, maybe using the embeddings to fetch similar concepts when direct answers aren’t found).
    * If the Attachment Agent (emotional reasoning) is to have any effect, implement simple rules: e.g., if user sentiment (we can guess from text, or even just presence of “!” or words like “frustrated”) is negative, the Chat agent will respond with a more apologetic tone. This could be done by having the orchestrator adjust the persona or insert a directive like “The user seems upset, respond supportively.”
    * **MPorter in Chat:** Possibly integrate the strategy agent to manage long conversations. For instance, if the user’s query is very complex, MPorter could break it into sub-questions for the other agents. We might simulate this by handling compound questions or having the chat orchestrator call the Knowledge agent multiple times for parts of the question.
    * Conflict handling in Chat: If two knowledge sources conflict (say concept pack vs code context), orchestrator should detect and either ask user for clarification or choose one based on priority (perhaps code context overrides general knowledge, etc.). Implement a basic policy and test it (“What is X?” where X is defined differently in user’s project vs general knowledge).
    * Use remaining time to fix any critical issues or do another mini-test of full system (both IDE and Chat) ahead of final stretch.  
      *Checkpoint:* By end of Day 14, the multi-agent system for Chat should be more robust and intelligent. TORI Chat can incorporate multiple perspectives or specialized sub-agents (e.g., retrieval vs reasoning) in one conversation without confusing the userfile-bnjjcfgff6iusnbzdwqayo. We should also now have both IDE and Chat components largely feature-complete, setting the stage for final refinements and testing in the last week.
15. **Day 15 (May 26, 2025) – Documentation, Tutorials & UX Polish**  
    *Goal:* Prepare user-facing documentation and improve the user experience with small tweaks and fixes. Make TORI feel more polished.  
    *Tasks:*
    * Write a **quickstart guide** and in-app tutorial hints. E.g., when TORI IDE first launches, show a “Welcome to TORI” message that highlights key areas (concept explorer, chat panel, etc.). Also prepare a README or user manual covering installation, basic usage, and explanation of TORI’s key concepts (for early adopters trying it out).
    * Ensure all UI text is clear and user-friendly. Replace any placeholder labels or cryptic terms. For example, if a suggestion says “Phase conflict in module X,” perhaps rephrase to “Inconsistency detected between module X and Y (they may need synchronization)” – more understandable language.
    * Add tooltips or help icons where needed, especially for novel features. If a user hovers over the oscillator dashboard, a tooltip could explain “This visualization shows the synchronization of key components – when waves line up, components are in sync.”
    * Finalize persona and branding in UI: name, logos, color scheme consistent. Possibly incorporate a small TORI logo or avatar in the chat interface for personality.
    * If time, record or script some example **use-case scenarios** to validate workflow: e.g., “User story: debugging a problem with TORI’s help” – walk through it and ensure the UI/agents support it smoothly. Refine any steps that felt clunky.  
      *Checkpoint:* By end of Day 15, TORI should *feel* more like a polished product rather than a research demo. New users should have some guidance (docs/tutorial) and the overall UX should be refined. We want to be able to hand TORI to a friendly tester and have them understand what to do without devs explaining everything in person.
16. **Day 16 (May 27, 2025) – Internal Testing & Bug Bash**  
    *Goal:* Conduct intensive testing of all features (IDE and Chat), fix bugs, and ensure quality. Essentially a “bug bash” day.  
    *Tasks:*
    * Have team members (and possibly a small group of internal users) use TORI for real tasks for a few hours. Collect their feedback and any issues encountered.
    * Triage bugs: categorize into must-fix for launch vs can-fix-later. Focus on must-fix now.
    * Fix critical bugs across the board: crashes, incorrect analyses, UI misalignments, etc. Pay special attention to anything that could lead to wrong behavior (e.g., an agent suggesting something dangerous in code).
    * Cross-platform test if possible: run on Windows, Mac, Linux to ensure no environment-specific bugs (since local-first means environment differences matter).
    * Test performance in some edge cases (very large file, extremely long chat conversation) to make sure nothing catastrophic occurs (if it does, perhaps implement safety cut-offs, like “Results truncated…”).  
      *Checkpoint:* By end of Day 16, the bug count should be vastly reduced. Ideally, no showstopper bugs remain. TORI is now in a *release candidate* state for both IDE and Chat. We should have a list of known issues that are either minor or will be addressed with lower priority.
17. **Day 17 (May 28, 2025) – Deployment Preparation & Packaging**  
    *Goal:* Prepare TORI for distribution: installer, packaging, environment setup scripts, etc., and ensure it can be easily deployed by users or on servers.  
    *Tasks:*
    * Create an **installer or package** for TORI IDE (Thoughtboard). Possibly using PyInstaller for a one-file executable or creating an Electron app package. Ensure that the ALAN core (Python backend) starts up with the UI seamlessly.
    * Package TORI Chat standalone similarly, if it will be delivered separately (maybe a Docker image or a simple executable that launches the chat server and web UI).
    * Write deployment docs (for IT teams): e.g., how to install on an offline machine, required system resources, how to configure on-prem.
    * Test installation on a fresh system: does it run out of the box? Catch any missing dependencies or config steps and adjust the installer accordingly.
    * If licensing or activation is needed (for enterprise, maybe not yet), at least design where that would hook in. Possibly just stub an “enter license” dialog if appropriate (though might not be needed for v1 tech preview).
    * Ensure logging (from Day 12) goes to proper files in the installed locations and not to console, etc., as expected.  
      *Checkpoint:* By end of Day 17, we should have **installable builds** of TORI IDE and TORI Chat. This means we could send a file (or Docker image) to someone and they could get TORI running without our direct involvement. It’s a key step towards any sort of user testing or pilot program.
18. **Day 18 (May 29, 2025) – Documentation & Final Touches**  
    *Goal:* Finish comprehensive documentation (technical and user), and add any final touches or small features that were deferred but are quick wins.  
    *Tasks:*
    * Complete the **user manual** covering all features of v1.0. Include screenshots of the UI, examples of how to interpret the outputs (like an example of the oscillator dashboard and what it meant in a scenario), and FAQs for common questions.
    * Write a section in docs for “Under the hood: How TORI works” for the curious user or stakeholder, explaining concepts like the concept graph and phase reasoning in approachable terms (this leverages our deep knowledge but presents it simply).
    * Ensure the in-app help links (if any) point to the correct documentation or have tooltips (from Day 15).
    * Implement any **small deferred enhancements** if time permits. For example, maybe adding a keyboard shortcut to open the concept explorer, or a command palette entry for “Ask TORI” to quickly query something without clicking the chat panel. These little touches improve UX if easily done.
    * Final branding pass: ensure the app names, version numbers, and about dialogs are correct (“TORI IDE v1.0 (ALAN 2.x core)” etc.).  
      *Checkpoint:* By end of Day 18, documentation should be **complete and polished**. All those using TORI (developers, early adopters) have the materials to understand and troubleshoot it. The product should have a fit-and-finish that we’re proud to present. We should feel confident that no major feature is undocumented or totally unknown to the user.
19. **Day 19 (May 30, 2025) – Final Testing & UX Review**  
    *Goal:* Do a final end-to-end test pass as if we were the end-user. Refine any last UX issues and make sure we’re truly ready to release.  
    *Tasks:*
    * Perform a scenario-based test: e.g., “Use TORI IDE to improve a small project from scratch,” and “Use TORI Chat to learn about a domain.” Follow these scenarios step by step, noting any awkwardness or minor bugs.
    * Fix *minor bugs* that are quick (typos, alignment issues, log verbosity, etc.). Anything more than minor should already have been caught, but if something appears now that is critical, we still have a bit of buffer to address it.
    * UI/UX refinement: tweak colors, spacing, icons to ensure the interface looks clean and professional. Make sure the visual theme is consistent (fonts, sizes).
    * Confirm all *integrations* work one more time: The chat panel in IDE still works in the packaged build, the concept pack loading in Chat works, etc., in the final packaged environment.
    * By mid-day, *freeze the code*. Enter code freeze for release candidate except for emergency fixes. From this point, focus on stability.
    * If time, do a dry-run of a **demo or presentation**. Usually, we’d want to simulate what we will show stakeholders or potential users and ensure TORI can be showcased without hiccups. This doubles as a final test.  
      *Checkpoint:* End of Day 19, we have what we consider the **release candidate build** of TORI IDE and TORI Chat v1.0. The team should feel confident in doing a live demo to stakeholders without making excuses for rough edges. We likely will tag this version in our repo as v1.0-rc.
20. **Day 20 (May 31, 2025) – Buffer, Polishing & Pre-Launch Prep**  
    *Goal:* Use this day as a buffer for any last-minute critical issues and prepare for launch activities (like presentation, marketing collaterals).  
    *Tasks:*
    * If any critical bug was found late Day 19 or by additional testers (perhaps someone from another team trying it on Day 19 evening), fix it with priority.
    * Finalize the **presentation/demo script** for the launch meeting or recording. Ensure all materials (slides, if any, or sample projects to demo on) are ready.
    * Create some **marketing content** if applicable at this stage: perhaps a one-pager summary of TORI’s value prop to accompany the release, or internal announcement text. This may involve working with a product or marketing person, but the technical team provides inputs (features, differentiators).
    * Double-check that all **SWOT points** have been addressed in some form in the product or documentation (e.g., weakness of limited language support is documented and openly acknowledged with plans, etc., so it doesn’t catch anyone by surprise).
    * Rest (if everything is truly done). The team has pushed hard; if all is in order, a lighter day to recuperate ensures we approach launch with clear minds.  
      *Checkpoint:* End of Day 20 should have zero critical open issues. We are essentially ready to launch the product the next day. All that remains is the actual launch event or final sign-off.
21. **Day 21 (June 1, 2025) – Sprint Completion & Launch Readiness**  
    *Goal:* Final day to wrap up the sprint, do any last reviews, and *prepare for the launch/presentation*. If possible, also allow the team a brief respite before the big day.  
    *Tasks:*
    * **Final sanity test:** Run through the core functionality one more time quickly in the morning. If anything unexpected occurs, decide if it must be fixed now or can be deferred (ideally deferred unless it’s truly showstopping, as we shouldn’t be making code changes on launch day).
    * Package the final release builds, double-check their integrity (hashes, can be installed fresh, etc.).
    * Ensure all documentation and materials are accessible (upload to internal site or include in the installer package as appropriate).
    * Team meeting to recap the sprint accomplishments, and discuss support plan post-launch (e.g., how to handle user feedback, quick patches if needed).
    * If this is an internal launch, coordinate with any stakeholders on the launch event details. If external, perhaps push the repository to a private share or similar.
    * **Rest and mental preparation:** Encourage the team to take it easy in the latter half of the day if everything is done. We want everyone fresh for the launch or next phase.  
      *Checkpoint:* By end of Day 21, the 21-day sprint is **successfully completed**. TORI IDE and ALAN 2.x (with TORI Chat) are ready for deployment/presentation. The team can confidently say we have met the goals of the sprint and are prepared to introduce TORI to its initial users.

**TORI Chat – 21-Day Sprint Plan (May 12 – June 1, 2025)**

*(The TORI Chat development ran in parallel to the IDE sprint, with some shared efforts on core integration, multi-agent system, etc. The numbering here mirrors the days, but many Chat tasks were done concurrently by a sub-team focusing on the chat interface and capabilities.)*

**Sprint Goal:** Evolve TORI Chat from a rudimentary prototype into a robust conversational assistant with enterprise-ready features and seamless integration with the TORI cognitive core. Ensure TORI Chat can operate both embedded in TORI IDE and as a standalone product.

1. **Day 1 (May 12, 2025) – Chat Sprint Kickoff & Environment Setup**  
   *Goal:* Align on TORI Chat objectives and set up the development environment for the chat application.  
   *Tasks:*
   * Kickoff meeting in sync with the IDE team, focusing on Chat-specific deliverables (like persona handling, concept packs, etc.). Ensure cross-team understanding of what components are shared.
   * Set up the chat server framework (likely extending the same ALAN backend). If using a web UI for chat, set up the development environment for that (Node/Electron or pure web).
   * Ensure the concept network and core from ALAN can be accessed in a stateless way (since chat queries might be one-off calls to the core). Prepare scaffolding for maintaining chat conversation state (likely a context object storing recent conversation or a pointer to relevant concept subgraph).
   * Merge any baseline code for chat from prototypes into main, analogous to the IDE merging step.  
     *Checkpoint:* Chat team has a running development instance of TORI Chat (even if it’s just echoing or a basic template) and is in sync with the overall project plan. Environment issues resolved.
2. **Day 2 (May 13, 2025) – Core Alignment for Chat (Oscillators & Memory)**  
   *Goal:* Tie the updated core (oscillator sync, etc.) into the chat flow. Ensure that chat inputs create proper concept activations and that the conversation context is maintained in the core.  
   *Tasks:*
   * Modify the NLP pipeline so

**Comprehensive Master Plan and Strategic Analysis – TORI IDE & ALAN 2.x Cognitive OS**

**Executive Summary**

**TORI** is a next-generation *cognitive operating system* comprising two flagship products: **TORI IDE** (an AI-augmented Integrated Development Environment) and **TORI Chat** (an intelligent chat assistant, including an enterprise-grade variant). This master guide serves as a comprehensive report, technical specification, development roadmap, and marketing playbook for the TORI suite. It encapsulates the visionary paradigm behind TORI, the current architecture and module breakdown, a 21-day sprint plan for rapid development, SWOT analyses, and go-to-market strategies for each product.

* **Vision:** TORI embodies a paradigm shift from traditional AI tools. Rather than relying on rote token prediction, TORI’s core is grounded in **concepts over tokens** and \**memory over stateless prediction*[medium.com](https://medium.com/syncedreview/from-token-to-conceptual-the-rise-of-metas-large-concept-models-in-multilingual-ai-b32acbfeb792#:~:text=generalization%20across%20languages%2C%20outperforming%20existing,LLMs%20of%20comparable%20size)】. The vision is an “AI-native” development experience in which the system truly understands code and conversations at a conceptual level, enabling unprecedented levels of assistance and insight. Developers move from using tools to collaborating with a *cognitive partner*.
* **Architecture:** At TORI’s heart is the **ALAN 2.x cognitive engine**, a spectral-phase reasoning core that maintains a rich **Large Concept Network (LCN)** of knowledge. This core powers both the IDE and Chat, providing deterministic, auditable reasoning steps instead of opaque neural guesswor[ibm.com](https://www.ibm.com/think/topics/black-box-ai#:~:text=White%20box%20AI%2C%20also%20called,Users)】. Surrounding the core are modular subsystems for language processing, user interaction, and multi-agent orchestration. In the IDE, this means deeply understanding code structure and evolution; in Chat, it means maintaining contextual, persona-driven dialogue. Both products share a **local-first, privacy-conscious design** that can operate on-premises without cloud dependence.
* **Current Status:** The TORI codebase has been through early prototyping phases. Key modules for concept parsing, memory graph management, and a basic front-end exist, though some components remain mock or placeholder implementations pending integration of the full ALAN engine. For example, a document ingestion pipeline successfully indexes PDFs into the concept network, but the **phase oscillator network** underpinning dynamic reasoning is only *partially implemented*. The current TORI IDE UI is functional as a prototype “**Thoughtboard**” dynamic code canvas, albeit without full polish or all expected IDE features. TORI Chat’s core reasoning back-end is in place, but its user-facing chat interface and multi-agent persona system are at a rudimentary stage. These gaps are identified in the module-by-module analysis, and the plan is to replace placeholder code with production-ready logic during the upcoming development sprints. *Before sprinting on new features, completing these core components (oscillator sync, spectral engine, ELFIN parser, orchestrator) is imperative, to ensure the foundation is solid.*
* **Roadmap:** A detailed **21-day sprint plan** for each product outlines the path from the current state to a viable v1.0 release. Each sprint plan is organized into daily milestones with one rest/review day built-in per week. (For reference, **Day 1 corresponds to May 12, 2025**.) The TORI IDE plan focuses on integrating the ALAN core with the IDE front-end, implementing conceptual debugging tools, and eliminating any “mock” data paths. The TORI Chat plan emphasizes developing a robust conversational interface, domain-specific **“concept pack”** integration, and enterprise features like audit logs and on-prem deployment support. Both plans culminate in a synchronized launch, ensuring the IDE and Chat leverage each other (e.g. the IDE generating chat-ready knowledge graphs, and the Chat agent assisting within the IDE).
* **SWOT Highlights:** TORI’s **Strengths** lie in its unprecedented approach: a phase-synchronized reasoning core that yields deterministic, explainable insights (a “glass-box” AI approach[ibm.com](https://www.ibm.com/think/topics/black-box-ai#:~:text=White%20box%20AI%2C%20also%20called,Users)】, and a local-first architecture that sidesteps the compliance and cost issues of cloud-only AI solutions. **Weaknesses** relate to typical early-stage challenges: limited UI polish compared to incumbent tools, narrow language support at launch (only Python and the custom ELFIN DSL are first-class), and the need for extensive real-world validation of its novel techniques. **Opportunities** include growing demand for on-premises AI solutions (for privacy and control), and synergy between development and conversational AI (TORI uniquely bridges these). **Threats** include rapid moves by major IDE vendors to integrate similar AI capabilities, and skepticism toward unproven AI paradigms in conservative industries. Our strategy and roadmap specifically address these points through aggressive development, transparency, and engagement with early adopters.

**Vision and Guiding Principles**

**TORI’s vision** is to create a *cognitive operating system* that transforms how we interact with software development and knowledge work. This vision is grounded in two core ideas: focusing on **conceptual understanding** rather than surface-level output, and building **persistent knowledge** rather than stateless interactions. Traditional AI coding assistants operate at the level of syntax and tokens, predicting text without true understanding. TORI takes a different path – it builds and manipulates rich *conceptual models*. In the IDE, this means understanding the abstract architecture and intent behind code; in Chat, it means grasping the semantics and context of a conversation, not just producing plausible sentences. By elevating to the conceptual level, TORI provides insights and assistance that feel genuinely intelligent and context-aware, rather than generative but shallow. As one early reviewer noted, TORI represents not just a toolset, but *a paradigm shift in how we think about code, conversations, and AI assistance*.

In addition, instead of treating each user query or coding session as an isolated event, TORI maintains a **persistent memory** of interactions and learned concepts. This is not simply a chat history or cache, but a structured memory in the form of the LCN, phase states, and accumulated patterns. TORI remembers *why* a piece of code was written or *how* a conclusion in chat was reached, enabling it to explain its reasoning and build on past knowledge. This contrasts with stateless large language model tools that have no inherent memory of past interactions. TORI’s approach yields an AI that can truly learn and adapt over time – accumulating **wisdom** rather than resetting after each prompt.

From this vision, we derive a single unifying goal: **TORI is not just a development tool or a chatbot; it is a thinking partner that grows with the user.** The IDE is envisioned as a *“Conceptual Canvas”* where code and design interplay dynamically with the AI’s understanding. The Chat is envisioned as a multi-mind conversational agent that can reason through complex problems with traceable logic. Together, they form a unified cognitive workbench.

Several **guiding principles** direct our design and implementation:

* **Human-Centric Design:** TORI keeps the human user in control and informed at all times. We adhere to the principle of *“human in the loop”*. Features like explainable suggestions, the ability to inspect the concept graph, and toggles for AI assistance are built-in. The user should feel TORI is *amplifying* their abilities, not automating tasks away without consent or clarity.
* **Transparency and Explainability:** Every AI-driven action by TORI is explainable via its concept graph or a rationale. For example, if TORI IDE suggests a code refactor, it can point to the specific concepts and oscillator signals that led to that suggestion (e.g., a function concept that is out-of-phase with related functions, indicating a design inconsistency). Similarly, TORI Chat doesn’t just answer questions – it can provide a brief explanation or source (e.g., which internal knowledge or which agent’s logic contributed to the answer). This fosters trust and positions TORI as a “glass-box” AI, in contrast to opaque *black-box* model[ibm.com](https://www.ibm.com/think/topics/black-box-ai#:~:text=White%20box%20AI%2C%20also%20called,Users)】.
* **Deterministic Core, Creative Edges:** TORI’s architecture emphasizes *deterministic reasoning* in its core (the spectral-phase logic, rule-based multi-agent decisions, etc.) to ensure reproducibility and auditability. However, we allow for stochastic creativity at the periphery. For instance, a language model may be used to generate natural language summaries or code comments under TORI’s guidance. The rule is to use randomness only where it’s safe and adds value (e.g., phrasing a response kindly), but never in the core understanding of the code or conversation. This yields a system that is both **reliable** and **inventive** – core logic is stable and verifiable, while user-facing interactions can still have the flexibility and richness of generative AI.
* **Local-First and Privacy-Preserving:** TORI is built with a *local-first* mindset. Users retain complete ownership of their code, data, and the AI’s knowledge about them. Both TORI IDE and TORI Chat can run fully locally or on a private network. No data leaves the user’s machine unless explicitly configured to (for example, if the user chooses to enable cloud connectivity or external APIs). This addresses enterprises’ needs to keep proprietary code and discussions in-house. It also has a technical upside: operating locally yields lower latency and tighter integration with the user’s environment (files, processes, dev tools) since TORI can directly interface with them without network overhead.
* **Modularity and Extensibility:** TORI’s cognitive OS is modular by design. Each subsystem – parsing, concept storage, reasoning agents, UI – is a separate module with well-defined interfaces. This allows future improvements or swaps (e.g., plugging in a new parser for a different language, or adding a specialized agent) without overhauling the entire system. It also means parts of TORI can be extended by third-parties; for instance, developers might create new ALAN agents or integrate TORI’s concept graph with external tools via APIs. This modularity also aids maintainability: given TORI’s broad scope, we must be able to update one component (say, the NLP pipeline) independently of others.
* **Consistency with Developer Workflow:** Especially for TORI IDE, we design to *“meet developers where they are.”* This means TORI’s enhancements are built into familiar workflows. The IDE still allows editing code as text, using Git for version control, running tests, etc. – but TORI augments these activities with cognitive insight. For example, a developer can code normally and optionally inspect TORI’s concept graph for deeper understanding, or receive proactive warnings about design issues. TORI Chat is designed to integrate into the developer’s environment too (for instance, accessible as an in-IDE chat panel or via tools like Slack through integrations) so that conversing with TORI feels like a natural extension of their workflow. The guiding motto is *“low floor, high ceiling, wide walls”* – easy to get started, immense power as you explore, and broadly applicable to many tasks.
* **Inspiration and Playfulness:** While TORI is grounded in rigorous concepts from cognitive science and spectral theory, the project embraces creativity and approachability. This appears in our internal code names (e.g., the oscillator core nicknamed “**Banksy**” for its artful sync patterns, conceptual caches called “cartridges,” etc.) and in the product’s personality. TORI aims to feel engaging, even fun – we believe a degree of playfulness encourages exploration and learning. For example, TORI might include easter eggs or a witty persona remark (tastefully and only when appropriate) to remind users this AI is *approachable*. This helps differentiate TORI: it’s advanced technology with a human touch and a bit of charm, not a cold enterprise robot.

Balancing these principles ensures that as we build TORI, we maintain both **intellectual rigor** (truly innovating and solving hard problems in AI reasoning) and **user empathy** (crafting an experience that empowers and delights). The following sections translate this vision and these principles into TORI’s concrete architecture and development plan.

**Cognitive OS Architecture Overview**

TORI’s architecture is a modular, layered cognitive system designed to support both the IDE and Chat applications through a shared intelligent core. At a high level, the architecture consists of four integrated subsystems:

* **ALAN 2.x Cognitive Core:** The “brain” of TORI – responsible for conceptual understanding, inference, and maintaining the knowledge network.
* **Language/Domain Integration Layer:** Bridges the core with domain-specific inputs (code, documents, natural language) and outputs, translating between raw data (files, text) and TORI’s concept representations.
* **User Interface & Tools:** The front-end components through which users interact with TORI (the Thoughtboard IDE UI, debugging and visualization tools, chat interfaces, etc.).
* **Multi-Agent Orchestration:** The layer managing a collection of specialized AI agents and coordinating their efforts via the shared core (the “mind society” within TORI).

Each of these is detailed below, followed by how they collaborate in the TORI IDE and TORI Chat contexts.

**ALAN 2.x Cognitive Core (Conceptual Engine)**

At the heart of TORI lies the **ALAN 2.x cognitive core**, which maintains a rich, interconnected representation of knowledge called the **Large Concept Network (LCN)**. This core can be thought of as TORI’s brain: it encodes concepts (from code constructs to conversation topics) as nodes in a graph and relationships between those concepts as edges. Beyond static relationships, ALAN 2.x introduces a novel *spectral-phase inference mechanism* that enables dynamic reasoning over time.

In essence, the core treats reasoning as a live process of **synchronization and resonance** among concepts. Every concept node in the LCN is accompanied by an internal state, including a **phase oscillator** representing its current “state of mind” or context. When the system processes new information or code changes, these oscillators are perturbed and evolve. The core monitors the collective behavior of all concept oscillators to identify emerging patterns of coherence or conflict. Intuitively, if two concepts should logically agree (say two functions that need to work in concert), the system expects their oscillators to synchronize. If they drift apart (out of phase), that signals a potential inconsistency or conflict.

This approach draws inspiration from **neural oscillation theories** in cognitive science – where synchronized neural firing is thought to underlie unified perceptions and thought[neurosciencenews.com](https://neurosciencenews.com/brain-rhythm-cognition-25941/#:~:text=It%20could%20be%20very%20informative,%E2%80%9D)】. By analogy, TORI’s synchronized concept oscillators imply a coherent understanding or stable solution, whereas desynchronized oscillators flag issues requiring attention. For example, if a function’s implementation is inconsistent with its specification, the concept representing the implementation may oscillate out-of-sync with the specification concept, alerting TORI that something is off.

**Phase-Coupled Oscillator Matrix (“Banksy Core”):** The engine that manages these oscillators is nicknamed *Banksy* for its focus on rhythmic, artful coordination. Technically, it’s a matrix of coupled oscillators associated with the concept graph. In earlier prototypes we considered using a standard Kuramoto model for synchronizing these oscillators, but we have since moved to a more advanced **ψ-based coupling approach** that leverages spectral analysis for stability and alignment. In this approach, each concept oscillator’s update is influenced not just by direct neighbors (related concepts), but by a global perspective from the system’s spectral mode analysis (explained shortly). The goal is to achieve robust consensus where appropriate, and deliberate divergence where a conflict genuinely exists (as opposed to noise causing false alarms).

Each concept oscillator has a phase (and possibly amplitude) that can be interpreted: two concepts in phase indicate they are in harmony or agreement; out-of-phase might indicate discrepancy or different “contexts.” By adjusting coupling strengths and rules (the core might couple oscillators of related concepts strongly, and unrelated ones weakly or not at all), TORI dynamically enforces a consistency check across the entire knowledge network. If concept A (e.g., a variable’s expected type) and concept B (the variable’s actual usage) are linked, the oscillator engine will try to keep A and B in sync. A sustained phase difference would mean the engine cannot reconcile them – prompting, for example, a warning to the user about a type mismatch. Through this mechanism, TORI’s core provides a continuous, interpretable measure of consistency across the system.

Importantly, the simplistic Kuramoto coupling has been replaced with a more *intelligent synchronization strategy*. We incorporate insights from **Koopman operator theory** and control theory so that the oscillators not only sync or desync, but do so in a stable manner. The system computes collective properties (like an order parameter indicating overall sync) and uses feedback to ensure stability (preventing chaotic swings). It also means we can identify clusters of synchronization – perhaps a subset of concepts oscillate together (indicating a coherent subsystem or topic), separate from another cluster. This clustering can reveal modular structure in code or conversation.

*(Status:* As of now, the phase-coupled oscillator network is partially implemented. The basic data structures exist and oscillators update in a rudimentary way, but the full ψ-based synchronization algorithm is being finalized. Completing this and fully integrating it into the runtime loop is a top priority in the development plan.)

**Koopman Morphic Engine (Spectral Analyzer):** Complementing the time-domain oscillator view, ALAN 2.x employs a **Koopman operator-based spectral analyzer** to study the evolution of the concept network state in a higher-dimensional space. The **Koopman operator** is a theoretical construct that allows us to treat a nonlinear dynamical system (like our knowledge state updates) via an infinite-dimensional linear operator acting on observation function[mdpi.com](https://www.mdpi.com/2227-7390/9/19/2495#:~:text=The%20Koopman%20operator%20theory%20is,are%20amenable%20to%20systematic%20analysis)】. In simpler terms, we lift the system’s state into a space of features (potentially complex eigenfunctions denoted ψ) where the dynamics can be approximated as linear. By doing so, we can apply linear spectral analysis: find eigenvalues and eigenvectors (modes) that describe recurring patterns or steady states in the reasoning process.

Practically, the Koopman engine monitors changes in the concept graph (like concept activation levels, oscillator phases) over time or iterations. It then tries to identify **spectral modes** – for instance, an eigenmode might correspond to a particular pattern of concept activation that repeats or a combination of concepts that tends to increase or decay together. The eigenvalues associated with these modes can tell us about stability: eigenvalues on the unit circle imply a repeating cycle, eigenvalues less than 1 in magnitude indicate a decaying mode (converging to stable state), and greater than 1 would indicate a growing instability. This analysis provides TORI with a predictive foresight. For example, if a particular conceptual inconsistency tends to amplify each iteration (an unstable mode), the system can flag it early as something that will likely cause problems if not addressed (like a bug that will cascade). If a mode corresponds to a settled understanding (convergence), TORI knows those aspects of the problem are resolved and stable.

In essence, the Koopman engine allows TORI to reason not just in the now, but about the *trajectory* of the reasoning process. It can answer questions like: *Is the system of concepts converging to a coherent understanding?* or *Are there oscillatory disagreements that persist?* It’s analogous to how an experienced engineer might sense if a project’s design discussions are converging or endlessly oscillating between options. Mathematically, it provides a global, linearized view of the nonlinear interactions happening in the cor[mdpi.com](https://www.mdpi.com/2227-7390/9/19/2495#:~:text=The%20Koopman%20operator%20theory%20is,are%20amenable%20to%20systematic%20analysis)】.

To ensure reasoning remains controlled and safe, we incorporate **Lyapunov stability measures** into this analysis. A Lyapunov function (think of it as an energy or potential function for the system) is used to prove stability of an equilibrium. In TORI’s context, we design Lyapunov-like metrics to ensure our reasoning process tends toward a stable equilibrium (a consistent state) and not diverge. If the system’s computed Lyapunov function is decreasing over reasoning iterations, it indicates the system is *stable in the sense of Lyapunov*, meaning small perturbations (like minor code edits or rephrasing of a question) will dampen out and not blow u[ocw.mit.edu](https://ocw.mit.edu/courses/16-30-feedback-control-systems-fall-2010/463e9559c6ef70f13ea51c3464bdc9c1_MIT16_30F10_lec22.pdf#:~:text=%E2%80%A2%20Stable%20in%20the%20sense,%E2%80%9D%20November%2027%2C%202010)】. The orchestrator can use these metrics to decide when to stop iterating on an internal problem (e.g., the agents have reached a consensus when the system is stable).

*(Status:* A scaffold of the spectral analysis exists – it can take snapshots of concept states and compute a basic decomposition (e.g., via an SVD or FFT). Currently it returns dummy data for testing the pipeline. Implementing the full Koopman-based analysis and integrating real stability checks is in progress. It’s slated for completion early in the development sprint, after which the core will regularly generate these spectral insights to guide reasoning.)

**Static Knowledge Base & Symbolic Reasoning:** Alongside the dynamic oscillator and spectral components, the ALAN core also maintains a static knowledge base and performs more traditional symbolic reasoning. For example, TORI stores facts and rules (like “if concept X implies concept Y, and X is true, then Y is true”) within the LCN. It can run logical inference on this static graph as needed. Moreover, TORI uses *embedding-based similarity* to relate concepts: each concept may have a vector embedding (learned from code context or language data) so that semantically similar concepts can be identified via cosine similarity. This is useful for suggestions (like finding analogies in code or clarifying ambiguous questions in chat). These static or precomputed relationships act as a baseline layer of “common sense” and domain knowledge, upon which the dynamic reasoning operates.

**Determinism and Auditability:** A key design choice is that, given the same inputs and initial state, the ALAN core’s reasoning process is *deterministic*. That means TORI will produce the same sequence of internal states and same outcomes every time for reproducibility. The apparent randomness or creativity in output (like phrasing) is separated out (e.g., handled by a language model at the very end, if at all) and does not affect the underlying reasoning trace. We log the sequence of reasoning steps (concept activations, agent decisions, etc.) as an **audit trail**. This allows users (especially enterprise users) to inspect *why* TORI arrived at a suggestion – they can trace through a sequence like: *“Function A’s concept was out-of-phase -> Refactoring agent noted duplication with Function B -> Suggested to unify them.”* This level of introspection is rarely possible with traditional AI assistants and is a differentiator for TORI’s cognitive OS approach.

**Language & Domain Integration Layer**

The Language & Domain Integration Layer is responsible for translating real-world inputs into TORI’s conceptual form and vice versa. It ensures that whether the input is source code, a natural language sentence, or a formal specification, TORI can incorporate it into the LCN; and when TORI produces an output (like an answer or code change), it can be rendered in a human-usable format.

**Programming Language Frontends (for TORI IDE):** TORI IDE supports multiple programming languages by using frontends that parse code into the LCN. For each supported programming language, we employ either a custom parser or an existing parser/AST generator to break code down into its structure. For example, for Python, TORI uses the Python AST module to get the syntax tree, then traverses it to create concept nodes for entities like functions, classes, variables, as well as relationships (e.g., “function *foo* calls function *bar*” becomes a link between the concept nodes for *foo* and *bar*). We attach additional semantic information when available: docstrings become documentation concept nodes, types (if inferred or declared) become type-concept links, etc. Our custom DSL ELFIN (detailed below) has its own parser that directly produces a rich concept graph (since ELFIN is designed to express high-level concepts natively).

For other languages like C/C++ or JavaScript/TypeScript, we integrate with established parsing tools (e.g., tree-sitter or language server protocols) to get an AST or symbol table, then map those into TORI’s concepts. In these cases, because we may not capture every nuance, the integration might be a bit shallower (for instance, TORI might not fully understand C++ template metaprogramming yet, but it will still record the relations it can, like function calls and inheritance). Our design makes Python and ELFIN *first-class* (with the richest understanding), while still supporting other popular languages sufficiently to be useful. Over time, we plan to extend first-class support to more languages.

Whenever code is edited or a new file is added, this layer updates the LCN incrementally. For example, adding a new function in code creates a new concept node and appropriate links without rebuilding the entire graph from scratch. This integration is efficient and continuous, feeding changes into the ALAN core’s reasoning loop (which then perhaps triggers re-synchronization or agents to check the implications of that new function).

**Domain Knowledge Ingestion (for TORI Chat and Enterprise):** TORI’s usefulness amplifies when it’s informed about the user’s domain. The integration layer can ingest structured knowledge bases such as **ontologies**, **taxonomies**, or corporate knowledge graphs and integrate them as part of the LCN. We call these packaged knowledge sets **Concept Packs**. For instance, an enterprise could load a “Financial Regulations Pack” that contains concepts like regulations, compliance procedures, definitions of banking terms, etc., all connected in a graph. TORI merges this with its own concept network (or keeps it in a linked subgraph) so that both TORI IDE and Chat can leverage that domain knowledge.

In the IDE, this could mean recognizing that a certain variable or function relates to a concept in the ontology (e.g., variable customerAccount is linked to the “Customer Account” concept in a finance ontology). In Chat, it means if the user asks “What is the KYC process for a new client?”, TORI has the concept of “KYC (Know Your Customer)” in its graph and can reason with it directly, producing a knowledgeable answer. The ingestion process typically involves reading a structured file (like OWL or CSV data for ontologies) and creating concepts and relations accordingly. We preserve any hierarchy (e.g., is-a relationships become concept edges) and key attributes (which might become metadata on concept nodes).

For unstructured but relevant data (like a company’s internal PDFs or manuals), TORI can use an offline process to index those into the concept graph as well (essentially performing entity extraction and linking to build a mini knowledge graph of the documents). This was prototyped with the PDF ingestion pipeline mentioned in the current status. TORI Chat can then retrieve and cite that information during conversation.

**Natural Language Processing (for TORI Chat):** When TORI Chat receives a user query in plain English (or other supported languages), the integration layer’s NLP component kicks in. It parses the sentence to understand its structure (using, for example, a combination of dependency parsing and named entity recognition). Key elements of the query are mapped to TORI’s known concepts. For instance, if the question is “How does the encryption function work in the login module?”, the NLP will identify “encryption function” and “login module” as likely referring to concepts. It will query the LCN to see if concepts exist that match those names or synonyms. If yes (say there’s a concept for a function encryptPassword and a concept for module Login), those will be activated and handed to the core to focus on. If not, the NLP can create placeholder concepts (to represent the user’s query topics) so that the reasoning engine can still work with them (perhaps the reasoning agent will then say “I don’t have information on encryption in login”).

The NLP also handles context continuation – TORI Chat keeps track of the conversation so far (via the LCN as well, where each utterance can be a concept or linked to topics). Pronouns or implicit references (“Can you explain *it* further?”) are resolved by looking at the context concepts (what is *it* likely referring to given what was just discussed). Because TORI uses a concept graph for memory, context handling is robust: instead of raw text, it knows the subject under discussion.

For generating responses, TORI has two pathways: If the answer can be constructed logically (like a series of steps or facts), TORI can compose it from the concept graph relations (ensuring each statement is grounded in a known concept or link). Alternatively, for more open-ended answers or simply to make the output fluent, TORI can utilize a template or a controlled language model to turn the structured answer into a natural-sounding one. The priority is always correctness and faithfulness to the underlying reasoning. We avoid any generative fluff that isn’t backed by the LCN or agents’ outcome. The user can, if desired, ask TORI Chat *why* it gave an answer, and the system can then reveal the chain of concepts or rules (this might be presented as a graph visualization or a step-by-step explanation, leveraging our audit trail).

**Bidirectional Interaction:** Integration isn’t just one-way. TORI not only ingests data into the concept graph, but can also effect changes based on its reasoning. For example, in the IDE, an agent might decide a piece of code should be refactored. TORI could (with user permission) perform that refactoring: the integration layer would then take the concept changes (like “Function X and Y merged into function Z”) and apply them to the actual source code (pretty-printing the new code via an AST transformation). Similarly, in Chat, if TORI comes to a conclusion that requires retrieving more info (like “to answer that, I need the user’s account history”), an agent might prompt the integration layer to fetch a file or call an API if allowed. These actions are mediated through well-defined interfaces so that security and permissions are respected (TORI will not perform an action unless configured to, especially in Chat – it won’t execute arbitrary code or calls just from a question, unless explicitly authorized as, say, a tool-using extension).

*(Current Implementation:* The integration layer is in a mixed state of completion. The Python and ELFIN language frontends are largely operational – they produce the concept graph for code and update it on changes. The C/C++ and JS integrations are basic (parsing via external tools works, but deeper semantic mapping is limited). The NLP pipeline for English is functional at a basic level (entity extraction and mapping to known concepts), using libraries like spaCy for parsing. Domain concept pack ingestion is prototyped (we tested with a small ontology, which TORI successfully merged into its graph). Much of the integration logic will be refined during the sprint as we encounter more real-world cases from user testing. Ensuring that this layer robustly handles imperfect input (e.g., code with syntax errors, or poorly worded questions) and interfaces smoothly with the rest of the system is an ongoing effort.)

**The ELFIN DSL: A Domain-Specific Language for Cognitive Coding**

One special element of TORI’s architecture is the inclusion of **ELFIN**, a custom domain-specific language designed alongside the ALAN core. ELFIN (an acronym loosely standing for *Extensible Language for Integrative Networks*, though informally named to suggest something small and magically helpful) serves as both a demo of TORI’s concept-oriented approach and a tool for users to explicitly script cognitive operations.

**Purpose and Philosophy:** ELFIN’s purpose is to allow developers to write code or specifications in a way that directly informs TORI’s concept model. Traditional programming languages require TORI to infer the concepts and intents from the code. With ELFIN, the language is constructed so that the *concepts are first-class citizens*. In practice, this means ELFIN syntax might allow the programmer to declare intentions or relationships that would be comments or implicit in other languages. For example, one might write in ELFIN:

arduino

Copy

concept DataPrivacy relates UserData, AccessControl

process scrubData(input UserData) ensures no PersonalInfo in output

In this hypothetical snippet, the programmer explicitly introduces a concept DataPrivacy and relates two other concepts to it, and defines a process with a postcondition. TORI would take this and create a DataPrivacy concept node linking UserData and AccessControl, and for the process scrubData, it knows a guarantee (no personal info in output) as a concept relation (perhaps linking the output concept to a “sanitized” concept).

Such expressiveness is valuable in domains where requirements and logic need to be very clear (financial contracts, safety-critical systems, etc.). ELFIN allows capturing those as part of the code itself, eliminating the gap between documentation and implementation. It’s similar in spirit to languages like Alloy or Z (formal specification languages), but integrated into TORI’s runtime reasoning. Indeed, one use-case for ELFIN is writing **specifications or rules** that TORI’s agents can then use. For example, an ELFIN rule might specify a design constraint (“every database query must go through the DataAccess layer concept”) and TORI’s Refactoring or Compliance agent can continuously check that this holds in the code.

**Integration with TORI:** The integration layer treats ELFIN as a first-class language. We have a defined grammar (encoded in something like a ANTLR or a custom parser) that directly produces concept graph structures. Unlike other languages, where we have to interpret the AST into concepts somewhat heuristically, ELFIN’s grammar is essentially *one-to-one mapped* with concept graph operations. If you define a concept or a relation in ELFIN, the parser calls the LCN API to create that node or edge. If you define a process or function, it creates a concept for it and also perhaps a state machine or logical relation for its pre/post conditions.

Because ELFIN is under our control, we can evolve it in tandem with features in ALAN. For instance, if we introduce a new kind of reasoning artifact (say, a “scenario” concept for test cases), we could extend ELFIN to allow writing those scenarios in code. ELFIN thus serves the dual purpose of showcasing TORI’s capabilities and giving power users a way to script or influence TORI’s reasoning. A developer could write an ELFIN snippet to guide the AI: *“Whenever concept X and Y are used together, treat them as interchangeable”* – effectively giving TORI a new rule on the fly.

**Current State:** ELFIN is currently in a prototype stage. The initial grammar is defined, covering core constructs like concept declarations, relationships, simple process definitions, and invariants. The parser can handle these and we’ve tested it on small examples (the resulting concept graph matches expectations). What remains is fleshing out the standard library of ELFIN (predefined concept types, perhaps some syntactic sugar) and integrating the ELFIN runtime semantics. Since ELFIN can describe processes, if one writes an algorithm in ELFIN, we might want to actually execute it or at least verify it. Executing ELFIN code could either mean translating it to Python/another language or interpreting it within TORI (likely out of scope for v1.0 except for trivial cases). For now, ELFIN’s emphasis is on *declarative knowledge*. Users can embed ELFIN snippets in comments or separate files to enhance TORI’s understanding of the system.

We include documentation for ELFIN in the TORI user guide as an advanced feature. We expect early adopters might experiment with it, giving us feedback. Over time, if ELFIN proves useful, it could become a cornerstone of how users explicitly communicate with the AI (almost like an API for knowledge).

**Example:** To illustrate ELFIN, consider a scenario where an enterprise wants to enforce data privacy rules in code. In ELFIN, they write:

cpp

Copy

concept PersonalData

concept EncryptedData

rule "Personal data must be encrypted at rest":

forall x: PersonalData, storageFunc f:

uses(f, x) -> ensures(EncryptedData)

This fictional syntax declares two concepts and a rule: any usage of personal data in a storage function must ensure it’s encrypted. TORI would incorporate this rule into the concept graph (perhaps as a constraint relationship). The Compliance agent, when scanning code, would then look for any function concept labeled as a storageFunc using a concept that is a PersonalData, and check if the output or result is marked as EncryptedData. If it finds a violation, it can flag it. Without ELFIN, such a domain-specific rule would be hard-coded or not known to TORI; with ELFIN, the user has effectively programmed the AI’s reasoning.

This level of customization is part of TORI’s long-term vision: an AI platform that organizations can directly imbue with their policies and knowledge through a language interface, rather than just hoping the AI “learned” it from training data. ELFIN is our first step in that direction.

**User Interface & Tools**

The User Interface and Tools layer encompasses the ways users interact with TORI’s intelligence. This includes the **TORI IDE “Thoughtboard” application**, various UI panels within it (like the concept explorer, spectral dashboard), and the **TORI Chat interfaces** (integrated and standalone).

The guiding design for UI is that it should present TORI’s capabilities in a clear, non-intrusive way, enhancing the user’s workflow without overwhelming them. The UI is also where we surface the inner workings of TORI in an understandable form (closing the loop on explainability).

**TORI IDE – Thoughtboard Interface:** TORI IDE is delivered as an independent desktop application (nickname: *Thoughtboard*). It provides a familiar coding environment augmented with TORI’s cognitive features. Key components of the Thoughtboard UI:

* **Code Editor with Inline AI Annotations:** The main editor looks and feels like a modern text editor/IDE (supporting syntax highlighting, code completion via perhaps existing language servers). TORI’s contributions come as subtle overlays: e.g., underlining code segments that TORI finds semantically problematic (not just syntax errors). If an agent has a suggestion or insight about a line or function, a small icon or colored underline appears. The developer can hover to see a tooltip with TORI’s note (e.g., “This function duplicates logic from function Y.”) or click to get a detailed explanation/fix.
* **Concept Graph Explorer:** A sidebar or pop-up panel that visualizes the portion of the concept graph relevant to the current context. For instance, if the developer is editing a function, they can open the Concept Explorer to see all concepts related to that function (variables, other functions it calls, requirements, etc.) in a node-link diagram. They can navigate this graph: clicking a concept jumps the editor to the corresponding code or brings up its documentation. The graph can be filtered or expanded – e.g., “show me the chain of concepts connecting this module to the security policy concept.” By visualizing code semantically, developers can understand impacts and dependencies that are not obvious from file structure alone.
* **Spectral Panel (Phase/Debug Dashboard):** This panel shows the dynamic analysis from the ALAN core. For example, it might display a real-time gauge of overall system consistency (if oscillators are largely synced or not), or a timeline graph of a particular concept’s phase drift over time as the code changed. In debugging mode, if the user runs the program (note: TORI IDE might integrate with a runtime to get execution traces), this panel could align runtime events with concept oscillations – essentially giving a new dimension to debug: not just variable values, but concept-level “health” signals. Concretely, a graph might show multiple sine-wave-like lines (each representing a high-level component’s phase) and highlight when one goes out of sync (pointing to a possible bug introduction time).
* **Multi-Agent Console:** TORI’s agents might produce textual logs or dialogues for the user. For instance, the Refactoring agent might have a note “I noticed functions X and Y are similar. Would you like me to refactor them?” The Test agent might output “Generated 2 new test cases for module Z.” We provide a console (or notifications area) where such messages are listed, potentially categorized by agent. This console also doubles as a command interface: the user could accept or dismiss suggestions directly from here, or ask follow-up (“Why do you suggest that?” which the agent can respond to via the Chat integration).
* **Integrated Chat Panel:** On one side of the IDE, there is a chat interface where the developer can converse with TORI Chat without leaving the IDE. This is essentially TORI Chat embedded – contextually aware of the open project. The developer could highlight code and ask, “TORI, how can I optimize this?” and the chat panel will respond, leveraging both its general training and specific knowledge from the code. Because it’s in-IDE, TORI Chat can do things like automatically link to relevant code or even execute small analyses (like “find all uses of this function” before answering).
* **Standard IDE Features Enhanced:** We include things like project navigation, file explorer, etc., but enhanced by TORI. For example, the file explorer might have badges on files indicating concept density or anomalies (imagine a file icon tinted red if that file contains concepts that are out-of-phase or marked in an agent warning). Search is semantic – beyond literal text search, you can search “concept: Login process” and TORI will find code related to the login process concept, even if the word “login” isn’t in those files (maybe it’s spread in different names).

The Thoughtboard UI is built with **local-first** principles: it does not require an internet connection to function fully (all heavy lifting is done by the ALAN core locally). This addresses privacy, and also ensures responsiveness. The UI communicates with the ALAN core via a local API or IPC mechanism (maybe an HTTP REST API on localhost, or direct memory calls if integrated).

We aim for the IDE to feel like a **modern, polished environment**. While initial versions may not have every feature of VS Code or IntelliJ, the critical ones are there, and TORI’s unique features are presented cleanly. We also brand the UI appropriately – using TORI’s theme (colors, logo) to give it a distinct identity as the “Cognitive IDE.”

*(Current UI Status:* The Thoughtboard interface exists in prototype. It currently runs as a Python/Electron app that opens a basic editor window and communicates with the core. Key panels like the concept explorer and phase dashboard are implemented in a rudimentary way (with dummy data for now). Over the development sprint, a lot of effort is allocated to hooking these up to real data and refining the UX. For example, as of now, the concept explorer can display a graph but not dynamically update; by mid-sprint it will update live as the code changes or as the user explores. Performance is another focus – ensuring that even with thousands of concept nodes, the UI remains responsive via techniques like virtualization or focusing on relevant subgraphs.)

**TORI Chat Interfaces:** TORI Chat can be accessed in two main ways: embedded in the IDE (as described) and as a **standalone application** for general conversational AI use.

* *Embedded in IDE:* This version of the interface is tailored to a developer’s needs. It might automatically inject context about the code when you ask a question (so you don’t have to type out “in function X, ...” if you have that function selected). It can also present answers with references into the code. For example, if TORI Chat explains an error, it might embed a snippet or a link that when clicked, navigates the IDE to the relevant code location. The chat UI inside IDE is minimalist (to not clutter coding space) but easily expandable.
* *Standalone Chat App:* For more general use (or for non-developers interacting with TORI’s knowledge), we provide TORI Chat as a separate desktop or web application. This has a chat window with history, persona selection, and possibly a way to manage knowledge packs. One could deploy this in an enterprise as an internal assistant: employees could query it about company policies, or troubleshoot issues by describing them in natural language. The standalone UI might allow logging in (if connecting to a central knowledge base), but in our local-first scenario, it could simply run locally with whatever knowledge packs are loaded. It will highlight that all data is local (to build trust for users worried about sensitive queries). The design is akin to popular chatbots, but with an emphasis that it *shows its work*. Perhaps an “inspect reasoning” button is available for power users to see the concept graph path or agent reasoning that led to an answer.

In both interfaces, **persona and multi-agent feedback** are important. TORI Chat can adopt different styles or roles (e.g., “explain like I’m a junior developer” vs “give me a strictly factual answer with references”). The UI might offer a toggle or prompt for that. Under the hood, this corresponds to activating different subsets of agents or applying certain response formating. Also, if multiple agents contribute to an answer (like a compliance check agent adds a caution), the UI could indicate that (“According to PolicyAgent: ...”).

**Visualization and Feedback Tools:** Aside from core features, we have tools for the user to visualize and correct TORI’s behavior. For example, an **Agent Manager** panel could list the agents, allow the user to toggle certain agents on/off or set their aggressiveness (maybe a slider for how proactive the Refactoring agent should be). This gives advanced users control and trust – they can turn off help they don’t want.

Another tool is the **Knowledge Browser**, which is essentially a read-only view into the LCN. This would list all concepts TORI knows about (perhaps grouped by type or source). A user could search their domain concepts here to see if TORI has captured them. This doubles as a debugging tool for concept integration – if TORI missed a key concept, the user might notice it here and know to feed TORI more info.

Finally, a **Settings/Privacy Panel** ensures the user can confirm that all operations are local (or connect only to specified resources), aligning with our privacy principle.

*(Current UI Status:* The standalone chat UI is at a very early stage – currently just a simple web page interface used for testing. We will use part of the sprint to build it out (likely leveraging web technologies for cross-platform ease). The integrated chat in IDE currently just opens a window to a dummy chatbot; hooking it to the real TORI Chat backend will happen once the core QA pipeline is solid. We anticipate rapid progress here once the rest is in place, since it’s largely an integration task with perhaps some front-end coding for polish.)

**Multi-Agent Orchestration**

One of TORI’s defining features is its **multi-agent system** – the idea that TORI’s intelligence is composed of multiple specialized agents working together under an orchestrator. This design mimics the human cognitive strategy of dividing problem-solving into sub-tasks (and even reflects Minsky’s Society of Mind theory, where many simple processes together yield intelligenc[en.wikipedia.org](https://en.wikipedia.org/wiki/Society_of_Mind#:~:text=In%20his%20book%20of%20the,2)】).

**Central Orchestrator:** At the center of the multi-agent layer is the orchestrator, which acts like a conductor for an orchestra of agents. The orchestrator’s responsibilities include: activating the appropriate agents for a given situation, mediating conflicts between agents, and integrating their outputs into a coherent result or user-facing action. It runs within the ALAN core and has access to the global state (concept graph, oscillator states, etc.), which it uses to inform decisions. For instance, when a code analysis event occurs (user saved a file), the orchestrator might simultaneously notify the Refactoring Agent, Documentation Agent, and Test Agent to do their analysis. If they produce outputs, the orchestrator collects them. If two outputs conflict (one agent says “remove this function” and another says “improve this function”), the orchestrator uses predefined rules or meta-reasoning to resolve it (maybe it decides improvement is the safer route and discards the removal suggestion, or flags the conflict for the user to decide).

The orchestrator can also sequence agents. Suppose answering a chat query might involve multiple steps: retrieving info, then reasoning, then simplifying the answer. Instead of one monolithic agent doing all, the orchestrator can call a KnowledgeRetrieval agent first, then pass the findings to a Reasoning agent, then perhaps to an AnswerFormatter agent. This pipeline is orchestrated such that each agent does what it’s best at. The user just sees the final answer, but behind the scenes it was a team effort.

**Specialized Agents:** We have several agents implemented or planned:

* *Refactoring Agent:* Focuses on code quality and maintainability. It scans the concept graph for code smells: duplicated logic, long functions, inconsistent naming, etc. It leverages the semantic info – e.g., if two concepts (functions) have very similar subgraphs (same calls, similar documentation), that’s a hint to refactor. It then suggests a refactoring (or even generates the refactored code via transformation rules). It can also respond to user commands like “refactor this file” by applying known refactoring patterns.
* *Documentation Agent:* Ensures that the knowledge in the code is reflected in documentation. When code changes, it updates or marks outdated any concept documentation nodes. It can generate summaries for functions by looking at related concept linkages (like pre/post conditions, usage examples from code, etc.). It might also assist in creating architecture docs by summarizing clusters of concepts (for example, produce a brief description of the “Payment subsystem” concept cluster).
* *Testing (QA) Agent:* (Not to be confused with the Q&A nature of chat – here QA means Quality Assurance.) It looks at the concept graph to find untested concepts or potential bugs. If a concept (like a function or a requirement) has no associated “test case” concept, it flags it. It can even attempt to generate test cases. For example, it might identify boundary conditions for a function by analyzing its preconditions concept, then propose a test for each boundary. It works closely with the core analysis; if, say, the Koopman analysis finds an unstable mode related to input size, the Test agent could suggest a stress test on that input range.
* *Security/Compliance Agent:* (Planned) Focuses on domain rules like “no sensitive data logged” or “function X must always be called before Y.” These could be programmed via ELFIN or policy files and this agent checks them continuously. It highlights any violations. For instance, if concept “PersonalData” flows into a function not marked as secure, it will warn.
* *Knowledge Retrieval Agent:* (Primarily for Chat) When a question is asked, this agent searches the LCN and external sources (if allowed) for relevant information. It might fetch definitions, or recall that “Oh, the user’s asking about Concept Z, which was referenced in Document Q.” It basically does the research part, so the reasoning agent can focus on logic.
* *Reasoning Agent:* (Primarily for Chat) This agent takes the assembled facts and tries to logically deduce or answer the question. It might chain together multiple facts (kind of like a theorem prover or rule engine). For code questions, it could perform static analysis steps to find an answer (like simulate execution in concept-space, etc.).
* *Persona/Stylist Agent:* This agent modulates how answers are delivered. For example, if the user selected an “Expert” persona vs a “Tutor” persona, the stylist agent will adjust wording and the level of detail. It doesn’t change the content (which comes from the reasoning agent), just the packaging. In multi-agent terms, it might even be considered a post-processing agent on the answer.
* *MPorter (Meta-Strategy) Agent:* This is the strategy agent mentioned earlier. It monitors high-level goals and progress. In the IDE, it might notice the developer has been ignoring a particular critical warning and might gently resurface it or reprioritize agent outputs (ensuring important things don’t get lost in noise). In Chat, it could break down complex user requests. If a user asks a vague multi-part question, MPorter agent might internally split it into parts and ask the knowledge agent to handle each, then aggregate results. It’s essentially thinking about *how to think* – planning the orchestration of other agents for complex tasks.
* *Attachment (Emotional) Agent:* This agent monitors the user’s emotional state (inferred from their interactions or message tone) and adjusts TORI’s responses accordingly. It introduces an empathic element. For example, if a user in chat says “I’m really stuck and this is frustrating me,” the Attachment agent will influence the response to be more supportive or apologetic (“I understand this can be frustrating. Let’s break it down together.”[en.wikipedia.org](https://en.wikipedia.org/wiki/Affective_computing#:~:text=Affective%20computing%20is%20the%20study,interpret%20the%20emotional%20state%20of)】. In the IDE, if a user seems to be thrashing (repeatedly changing code back and forth), it might suggest taking a step back or offer an explanation of the underlying issue to calm the confusion. This agent does not solve technical problems – it watches the *interaction* and ensures the AI’s behavior maintains a good rapport with the user.

**Conflict Resolution and Synergy:** With many agents, conflicts or redundant efforts can occur. The orchestrator uses a few strategies to handle this:

* *Priority rules:* Some agents have higher priority outputs (e.g., a security violation from the Compliance agent might override a suggestion from the Refactoring agent in terms of urgency).
* *Merging:* If two agents essentially suggest the same thing in different words, the orchestrator merges that into one suggestion to the user (giving credit to both, perhaps).
* *Competition:* In some cases, the orchestrator might let agents “debate.” For instance, one agent might propose solution A to a problem, another proposes B. The orchestrator could let both run through a simulation or argument and see which is more convincing (similar to how adversarial agents work in some AI frameworks). This is experimental, but TORI Chat’s multi-answer consideration could use this – generating answers via different reasoning paths and then choosing the best.
* *User input:* Ultimately, if agents disagree, TORI can surface the disagreement to the user: “There are two ways to resolve this issue: method X (suggested by DesignAgent) and method Y (suggested by PerfAgent). Would you like to consider one?” This turns a conflict into a choice for the human, which is often the right approach when design trade-offs are involved.

**Orchestration in Practice:** Suppose a developer hits “Analyze Project”. The orchestrator triggers: RefactoringAgent (checks code structure), TestAgent (looks for test coverage), ComplianceAgent (checks rules), DocumentationAgent (updates docs), and they all run in parallel. The orchestrator gathers their findings: maybe two refactor suggestions, one compliance warning, and documentation updates. It then prioritizes the compliance warning at top (marked in red in UI), lists refactor suggestions next (maybe in yellow), and notes doc updates silently (perhaps auto-applied). The developer sees a summary in the IDE’s agent console and can address them as needed.

In a chat scenario, say the user asks “How can I improve the performance of the data import process?” Orchestrator engages: KnowledgeAgent (find data import code and any known performance issues), RefactoringAgent (since it’s performance, maybe PerfAgent if we had one, to simulate usage or identify bottlenecks conceptually), ReasoningAgent (to formulate the improvement steps), then PersonaAgent (to format answer). The orchestrator sequences these, ensures the reasoning agent gets the findings from knowledge search (like “the data import reads line by line, which is slow”), and then the reasoning agent might conclude “use bulk operations or streaming”. The persona agent then crafts a response: “I notice the data import process reads data line-by-line. Combining those reads or using batch processing could significantly improve performance. For example, ...” The orchestrator sends that to the user. If the user asks “Why?”, the orchestrator might direct that to the ReasoningAgent or KnowledgeAgent to provide the conceptual reason (maybe citing that line-by-line I/O is conceptually linked to high overhead).

*(Current Status:* The multi-agent framework is in place with basic agents (Refactoring and Documentation running in the IDE context, Knowledge and a simple Q&A reasoning in chat context). Conflict resolution currently is rudimentary (essentially, we have a fixed priority order and we haven’t had complex conflicts in the prototype). Part of the sprint is dedicated to expanding these agents’ logic (as described in the 21-day plan) and refining the orchestrator policies. We’ll also be user-testing the agent outputs to adjust how verbose or aggressive each agent should be by default. The ability for a user to configure agents (turn off or tune) is planned in the UI, as mentioned.)

**Development Roadmap – 21-Day Sprint Plan (Starting May 12, 2025)**

To achieve a fully functional TORI IDE and TORI Chat by the end of the sprint, we have devised a **21-day development plan** divided into daily milestones. The plan assumes **Day 1 is Monday, May 12, 2025**. It is structured as three weeks of focused development, with each week allocating one day (Day 7, Day 14, Day 21) for rest, review, or buffer. Below is the plan for TORI IDE, followed by the plan for TORI Chat. Time-based references (Day 1, Day 2, etc.) are annotated with actual dates for clarity.

**Note:** The first week of the IDE sprint heavily emphasizes completing the partially implemented core features (oscillator sync, spectral analysis, ELFIN integration, orchestrator basics). These are foundational tasks that must be finished early to unblock dependent features. The chat sprint runs in parallel, often focusing on integration and interface aspects while core logic is firmed up in the IDE context.

**TORI IDE – 21-Day Sprint Plan (May 12 – June 1, 2025)**

**Sprint Goal:** Transform TORI IDE from a prototype into a polished v1.0 product, with all core cognitive features implemented, tested, and integrated into the Thoughtboard UI. By Day 21, the IDE should reliably provide intelligent assistance (refactor suggestions, conceptual warnings, etc.), and be ready for initial deployment to friendly users.

* **Day 1 (May 12, 2025)** – *Sprint Kickoff & Environment Setup*  
  **Goal:** Align team on objectives, finalize work assignments, and ensure the development environment is ready for rapid iteration.  
  **Tasks:** Kick off with a team meeting reviewing the Master Plan’s highlights (vision, architecture, critical components to finish). Set up version control, branch strategy, and CI pipelines. Ensure everyone can run the prototype (backend + UI) locally. Address any environment issues (e.g., if someone’s on Windows, make sure dependencies like spaCy, PyQt, etc., install properly). Clear the slate of trivial known bugs from prototypes so baseline is stable.  
  **Checkpoint:** By end of Day 1, the team is synchronized, the latest prototype code is merged to main, and each developer has a working instance of TORI IDE to build upon. Any blocking setup issues are resolved.
* **Day 2 (May 13, 2025)** – *Core Engine Implementation: Phase Synchronization*  
  **Goal:** Implement and verify the **Banksy phase-coupled oscillator synchronization** in ALAN core (replacing any placeholder logic for concept syncing).  
  **Tasks:** Develop the new synchronization algorithm based on the ψ-coupling design. Code the oscillator update loop that runs after each analysis tick: for each linked concept pair, adjust phases toward alignment (or intended phase offset). Incorporate global feedback: e.g., use a convergence factor from spectral analysis (if available) to modulate coupling strength (this ties in Day 3’s tasks). Write unit tests: create a mini concept graph in isolation and simulate oscillator steps, asserting that phases converge for connected nodes and remain independent for unconnected nodes. Remove the old dummy update code and integrate the new one into the main loop (ensuring it runs, say, whenever the concept graph changes).  
  **Checkpoint:** End of Day 2, oscillators in the core actually move and reflect influences. If concept A and B are linked, and A’s state changes, B’s phase will respond (rather than doing nothing as before). We likely log some debug info to see this in action. The code should be free of obvious stability issues (not diverging or oscillating wildly unless intended). The Kuramoto mention in documentation can be updated to note we now use the custom ψ-sync method.
* **Day 3 (May 14, 2025)** – *Core Engine Implementation: Spectral Analysis Pipeline*  
  **Goal:** Implement the **Koopman spectral analysis and Lyapunov stability checks** in the core to enable predictive reasoning.  
  **Tasks:** Build out the data pipeline for capturing state snapshots of the concept network (phases, active concept counts, etc.) over time or events. Implement a function to compute a spectral decomposition on recent state history – this could start simple: e.g., take the last N states, perform an SVD or eigen-decomposition, identify the dominant mode. Mark stable vs unstable trends (maybe define a Lyapunov function L and check L(state\_{t+1}) < L(state\_t) for stability). Integrate this so it runs periodically (not necessarily every code edit, maybe on a timer or when changes stabilize). Use results: e.g., if an unstable mode is detected, store an “alert” in core memory (the orchestrator can later translate that to a suggestion: “System architecture is oscillating – possible unresolved design decision.”). Also, feed spectral insights back into oscillator coupling if applicable (for instance, if one mode corresponds to two subsystems, perhaps temporarily decouple them by lowering coupling strength).  
  **Checkpoint:** End of Day 3, the core produces some spectral output. We should be able to see logs or debug info like “Mode1 eigenvalue = 0.95 (stable), Mode2 = 1.1 (growing unstable)”. If possible, a simple scenario where we *know* a pattern (like we oscillate a small graph input back and forth) confirms the spectral detection (it should spot the oscillation frequency). The code still might use a simplified analysis (e.g., linear trending) if full Koopman integration is complex, but the structure is in place and returning meaningful data.
* **Day 4 (May 15, 2025)** – *Integration of Core Components & Partial Testing*  
  **Goal:** Integrate the newly implemented core features (oscillator + spectral) with the rest of the system and run preliminary tests. Finish any loose ends in core (ELFIN parsing integration, etc.).  
  **Tasks:** Connect the ELFIN DSL parser into the build: ensure that if an .elfin file is present in a project, the integration layer processes it and populates concepts accordingly (this might have been partly done, make sure it’s fully working). Test the full core loop: e.g., load a sample project that has some known conceptual inconsistencies and see if the oscillator + spectral system picks them up (in logs or via some debug UI). Fine-tune parameters (coupling strengths, threshold for instability warnings) based on these tests. Address any core crashes or performance hiccups: e.g., if spectral calc is slow, maybe reduce frequency or data length for now. Essentially, stabilize the core. By now, the core is feature-complete, so begin writing documentation comments for it (to help others understand the complex logic).  
  **Checkpoint:** End of Day 4, the ALAN 2.x core is **fully operational** within the application. We have a stable build where the concept graph updates, oscillators sync, and spectral analysis runs without major issues. We likely produce some internal metrics (maybe a “core health” printout indicating system consistency) that will later be surfaced in UI. The team can proceed confident that this core foundation is ready to support UI and agent features.
* **Day 5 (May 16, 2025)** – *Thoughtboard UI: Concept Explorer & Core Data Wiring*  
  **Goal:** Hook up the Thoughtboard UI to the live data from the ALAN core (concept graph, agent outputs, etc.), focusing on the **Concept Graph Explorer** panel and inline annotations.  
  **Tasks:** Implement an API in the backend to serve concept graph data (e.g., a REST endpoint /concepts returning JSON of nodes/edges, or use an in-memory shared model if in Electron). In the front-end, replace the placeholder graph in the Concept Explorer with data from this API. Enable basic interactions: clicking a node in the graph highlights the corresponding code element in the editor (this requires mapping concept -> source location, which the backend can provide if the concept came from code). Conversely, when a user navigates to a different function in the editor, update the Concept Explorer to focus that function’s node (and its immediate relationships). Also, surface core analysis warnings: if the core has flagged an unstable mode or conflict concept, ensure the UI can highlight it (e.g., concept node might have a red outline, or the editor gutter might get an icon at relevant lines). Start displaying simple agent messages in an output panel (even if just logging them).  
  **Checkpoint:** By end of Day 5, the UI is truly “talking” to the backend: for instance, a developer can open the Concept Explorer and see an accurate graph of, say, the current file’s concepts and their links. It updates reasonably promptly after code changes or navigating to a new file. This is a big step as now TORI’s unique info is visible. Any glaring UI performance issues (like graph too slow on moderate number of nodes) are noted to optimize on Day 10. Inline annotations might still be few (since agents haven’t been fully enabled yet), but the pipeline for showing them (e.g., an underlined range corresponding to a concept with a warning) is in place.
* **Day 6 (May 17, 2025)** – *Multi-Agent System Activation (IDE context)*  
  **Goal:** Activate key **IDE agents** using the now-functional core, and integrate their outputs into the UI. This begins TORI’s intelligent behavior in the IDE.  
  **Tasks:** Enable the **Refactoring Agent** on code analysis events. Remove any placeholder logic; use real graph queries (e.g., find duplicates by concept similarity) to generate suggestions. For now, log these suggestions or list them in the agent console. Do the same for the **Documentation Agent** – e.g., after each analysis, have it update documentation concept nodes and perhaps suggest adding docstrings where missing. Ensure the orchestrator is coordinating: wire up a simple orchestrator that on each “analysis cycle” calls the active agents and collates their suggestions. Then implement UI rendering of these suggestions: e.g., a pane or popup listing “Suggestions: Function foo duplicates bar (RefactorAgent) – [Accept]/[Ignore]”. If accept is clicked, we won’t fully automate refactoring yet (unless trivial), but we could at least merge concept or mark it resolved. Also implement a way to dismiss suggestions. Start small: maybe just printing them to console initially, then improving UI as time allows. The aim is to see at least one concrete agent output by the user.  
  **Checkpoint:** End of Day 6, TORI IDE is **making intelligent suggestions**. For example, on our test project, the refactoring agent might flag two similar functions. We can see this either in the console or in an overlay UI. Multi-agent orchestration is rudimentary (likely sequential or priority-based), but working – meaning no crashes and suggestions seem relevant. This effectively is the first visible “AI assist” moment in the IDE.
* **Day 7 (May 18, 2025)** – *Review & Buffer (Core/Backend)*  
  **Goal:** Review progress of Week 1, fix bugs, and adjust plans if needed. Use as buffer for any unfinished tasks.  
  **Tasks:** Conduct a team demo of what we have so far (core working, UI showing graph, maybe one suggestion coming through). Collect any obvious issues: e.g., “the concept graph for large file is too cluttered – consider auto-grouping”, or “oscillator sync seems slow for 1000 concepts – maybe optimize coupling calc”. Spend time fixing critical bugs discovered in integration (like UI race conditions, or an agent suggestion misidentifying code). Polish some behind-the-scenes aspects: cleanup the code, add comments, perhaps improve logging structure. Update documentation (internal developer docs) to reflect how new core/agents work. Given this is a planned lighter day, ensure the team doesn’t burn out – possibly a half day of work and an afternoon of rest or learning.  
  **Checkpoint:** By end of Day 7, the team has a clear picture of what’s working well and what needs focus next. Any core or integration issues that could impede new feature work are resolved. The plan for Week 2 might be adjusted slightly based on Week 1 outcomes (for instance, if core is slower than anticipated, allocate some optimization time). The project is on track, or if behind, we have identified where to catch up using buffer days.
* **Day 8 (May 19, 2025)** – *Thoughtboard UI Polish & Advanced Features*  
  **Goal:** Improve the UI/UX of the Thoughtboard with better visualizations and controls, focusing on the **Spectral Dashboard** and user controls for agents.  
  **Tasks:** Now that the core provides spectral data, design and implement the **Spectral Debug Panel**. This could be a small graph in the corner showing overall system “coherence” over time (e.g., a line trending upward as things stabilize). Or perhaps list current active modes: e.g., “Mode: ‘Memory Leak potential’ – unstable”. Tie this with UI cues: e.g., if an unstable mode is detected, highlight the related concepts (maybe flashing icon on related modules). Also, implement **user controls for multi-agent**: a simple settings dialog where agents can be toggled on/off or set to auto-apply trivial fixes. Ensure these preferences persist in the session. Enhance the Concept Explorer by adding filtering options (checkboxes to show/hide certain concept types, like maybe turn off showing local variables to declutter). Add the ability to double-click a concept node and keep the explorer centered there (for deep dives). Finally, refine the **suggestion UI**: use icons (a lightbulb for refactor, a book for documentation suggestion, a shield for compliance, etc.) to label each suggestion’s source agent so the user knows why it’s suggested.  
  **Checkpoint:** End of Day 8, the Thoughtboard interface should feel more mature. The spectral panel is live, though it might be fairly simple in this version, it at least gives some feedback (like “System Stable” or “Instability detected in X”). Users (testers) can now control agent involvement somewhat. The concept explorer is more usable with filters. We are essentially moving from raw functionality to a cleaner user experience.
* **Day 9 (May 20, 2025)** – *Agent Enhancements (IDE)*  
  **Goal:** Increase the sophistication of the IDE agents and their coordination. Make their outputs more accurate and useful.  
  **Tasks:** Revisit the **Refactoring Agent** logic with fresh eyes after seeing initial outputs. Improve its detection algorithms: maybe implement a simple AST similarity metric for functions to catch duplicates that textual similarity might miss. Also, add new refactor types: e.g., detect when a class has too many responsibilities (if concept graph shows it’s linked to very disparate concepts, maybe suggest splitting class). Introduce a **Performance Agent** (if not separate, maybe part of refactoring agent): using concept metrics (like if a loop concept is heavily used or if a function’s complexity is high), suggest performance improvements (this can dovetail with insights from spectral if an unstable mode is related to performance). Enhance the **Documentation Agent** to auto-generate missing doc comments using available info (perhaps call a templating system or even an LLM for natural language, constrained by known facts). For multi-agent orchestration, implement a basic conflict resolution: e.g., if the Documentation agent wants to add a docstring to function X that the Refactoring agent wants to remove entirely, orchestrator should prioritize the refactor suggestion and hold off on documentation for now. Additionally, integrate the **Compliance/Security Agent** if any rules are readily available (even a dummy example rule via ELFIN). Test these in combination on sample scenarios.  
  **Checkpoint:** End of Day 9, the IDE’s intelligence feels sharper. We should see, for example, that it not only points out duplicate functions but maybe also says “Function Y is very large (50 lines) – consider refactoring for single responsibility.” Documentation agent might start filling in blanks which we can see in a diff view. We likely now have multiple suggestions possibly affecting the same code, so how orchestrator orders them is set (and visible to user with the labeling). The system’s advice breadth has increased (touching design, performance, etc. in addition to trivial duplicates).
* **Day 10 (May 21, 2025)** – *Performance Optimization & Scalability*  
  **Goal:** Ensure TORI IDE runs efficiently on larger projects and optimize any slow components (core or UI).  
  **Tasks:** Conduct profiling sessions. Load a codebase with, say, a few thousand concept nodes (maybe the TORI codebase itself) and measure UI render time for concept explorer, time taken for analysis loop, memory usage, etc. Optimize **core algorithms**: e.g., if oscillator update is O(n^2) on number of concepts, consider optimizations (since many concept pairs might have zero coupling, we could optimize by tracking edges). Possibly integrate a graph library or C extension for heavy math. Optimize **UI rendering**: use canvas or WebGL for concept graph if DOM approach lags, implement incremental graph updates instead of full redraws. Also optimize multi-agent runs: maybe stagger some agent work to not all hit at once if that causes CPU spikes (or run in parallel if CPU cores free). Check memory: ensure concept nodes are being freed if removed, etc. The goal is that medium projects (hundreds of files) do not bog down the experience. Also, consider toggling off some verbose debug logs now to reduce overhead.  
  **Checkpoint:** End of Day 10, performance is improved. Ideally, initial analysis of a moderately sized project should complete in a matter of seconds, and interactive operations (like editing a file) trigger responses that feel near-instant. If some heavy operations remain (maybe full project re-analysis), they are either optimized or demoted to a background thread so the UI stays responsive. We should document current performance metrics and decide if any features need to be cut or scaled back for v1.0 to maintain responsiveness (for instance, maybe not analyzing *every* file in huge projects, or adding a preference for scope of analysis).
* **Day 11 (May 22, 2025)** – *TORI Chat Integration (IDE & Standalone)*  
  **Goal:** Integrate the TORI Chat back-end with the IDE (embedded chat panel), and bring the standalone TORI Chat application to a functional state using the shared core.  
  **Tasks:** Connect the embedded chat panel in Thoughtboard to the ALAN core’s chat endpoint. Test end-to-end: the user types a question in the IDE, it goes to the chat pipeline (Knowledge agent finds relevant code concepts, reasoning agent forms answer), and a reply appears referencing code. Work on formatting: code excerpts in answers should be displayed nicely (monospaced, maybe syntax highlighted if feasible). Ensure the chat has access to the project’s LCN; this might involve saving a copy of the LCN or querying the live one. For the **standalone Chat app**, implement any missing pieces: perhaps a simple UI (we can reuse some web components from the embedded panel). Focus on the ability to load concept packs: create a UI control to load a file (like a finance ontology JSON), and when loaded, verify chat answers incorporate that knowledge. If the standalone is meant for non-coders, test it on general Q&A and refine response style to be more explanatory since no code context there. Tie up persona selection: add a dropdown or preset buttons in chat UI (e.g., “Detailed mode” vs “Brief mode”) which the Persona agent will interpret to modify answers.  
  **Checkpoint:** End of Day 11, TORI Chat works in both contexts. In the IDE, you can ask something like “What does this function do?” and get a meaningful answer that TORI derived from understanding the code (with maybe references or an offer to navigate to related code). Standalone, you can ask something from a loaded knowledge pack, like “What is KYC?” and get an answer drawn from the ontology loaded (with the answer clearly reflecting that domain knowledge). There may still be room to improve natural language fluency, but correctness and integration are the focus.
* **Day 12 (May 23, 2025)** – *Enterprise Readiness Features*  
  **Goal:** Add features important for enterprise deployments: **audit logging, privacy verification, configuration**.  
  **Tasks:** Implement an **Audit Log** system in the backend that records all suggestions made by TORI and actions taken (by the user or automatically). For example, log lines like “2025-05-23T10:05: RefactoringAgent suggested combining functions A and B [file names], user accepted.” or “ChatAgent provided answer about X using source Y.” Ensure these logs are stored locally in a secure way. Add a **toggle** in settings for “Send anonymous usage stats” (for our telemetry if any) defaulted off, and a clear indicator that TORI doesn’t send code to cloud (maybe a note on first launch to reassure privacy). Build a **configuration UI** to adjust performance-related settings or agent aggressiveness. Possibly integrate a **license management stub** if needed (e.g., a place to input a license key or check subscription, though for an internal plan this might not be needed yet). If time, implement a **safe mode** for Chat where it will not answer certain categories of questions (maybe irrelevant for on-prem, but if needed for liability, e.g., disallow advice on topics outside provided knowledge packs).  
  **Checkpoint:** End of Day 12, TORI is more enterprise-friendly. There is an audit log being generated that an admin could review for compliance (which also doubles as a debugging aid for us). The user can configure the tool’s behavior in meaningful ways through a preferences UI. We double-check that no data is going out that shouldn’t (run tests with firewalls, etc., to see that TORI doesn’t try to reach external servers unless explicitly asked for an update check perhaps). At this point, we are feature-complete for core functionality and we switch into fix/polish/test mode for the final stretch.
* **Day 13 (May 24, 2025)** – *User Testing and UX Refinement*  
  **Goal:** Conduct in-depth internal testing of TORI IDE & Chat with fresh eyes, and refine the UX based on feedback.  
  **Tasks:** Bring in a few colleagues (not on the dev team) for a user testing session. Have them use TORI IDE on a small project and TORI Chat on some questions. Observe where they get confused or where UI is unintuitive. Collect feedback: e.g., maybe they didn’t notice the chat panel icon, or found the concept graph overwhelming. Triage issues into “must-fix” vs “could improve later”. Focus on low-hanging fruit improvements from this feedback: adjusting iconography, adding a one-time tutorial pop-up (like highlighting “This is the Concept Explorer, which shows relationships.”), improving message wording (e.g., suggestions might say “(Refactor)…” which a user might not get; maybe say “TORI Suggestion:” instead). Fix any minor bugs discovered (like UI alignment issues, or an agent firing at the wrong time). Ensure consistency: check all button labels, terminology (“concept” vs “knowledge” – decide on user-facing terms and stick to one). If testers encountered any serious functional bugs (crashes, features not working as expected), address those immediately.  
  **Checkpoint:** End of Day 13, TORI should feel more **user-friendly**. Perhaps we added a quick start tooltip walkthrough that appears on first launch. We likely rename or clarify a few UI elements based on feedback. The application should handle common user behaviors more gracefully now (for example, if a user deletes a whole file, does the concept graph update without errors? Now it should).
* **Day 14 (May 25, 2025)** – *Comprehensive Testing & Bug Fixing (Bug Bash)*  
  **Goal:** Rigorously test all features (IDE and Chat) for bugs or edge cases and fix them.  
  **Tasks:** Do a “bug bash” where the dev team tries to break TORI in every way they can think of. This includes: extremely weird code constructs (does the parser choke?), concurrent actions (edit code while asking a chat question – does anything race or deadlock?), large inputs (open a huge file – does UI lag, and can we mitigate maybe by not highlighting concept for every single token?), error handling (shut down core unexpectedly, does UI show an error or freeze?). Test installation paths (fresh machine setup from our installer, testing user without Python installed if our bundle should cover it, etc.). List out all discovered bugs. Prioritize by severity: crashes and data loss/ corruption issues are highest, then major functional bugs, then minor cosmetic ones. **Fix aggressively** all the severe ones. For moderate ones, fix as many as time permits, especially if low risk. Some minor ones might be deferred if they’re not critical (we keep note to possibly address in a patch). Also run our automated test suite (if we have one by now, which we should for core logic) and ensure all pass. Perhaps incorporate last-minute test cases that cover new ground we learned from user tests. If any agent is misbehaving (like false positives or too many suggestions making noise), fine-tune thresholds or turn it down for release.  
  **Checkpoint:** End of Day 14, we ideally have zero known crash bugs and zero known major malfunctions. The product should be stable in heavy use. There may still be a list of minor quirks or potential improvements, but nothing that would embarrass or seriously frustrate an end user in a v1.0 trial. We are nearly code-complete; as we go into final days, focus shifts to documentation and final polish.
* **Day 15 (May 26, 2025)** – *Documentation & Final Feature Freeze*  
  **Goal:** Prepare comprehensive documentation (user guide, developer notes for internal use) and declare a feature freeze (no new features, only necessary fixes).  
  **Tasks:** Write the **User Guide** for TORI IDE and Chat. This includes: installation instructions, quickstart tutorial (covering basic workflow, e.g., “open your project, TORI will analyze and show suggestions; you can chat with TORI for help; here’s how to interpret the concept graph…”), explanation of each major UI component (maybe with screenshots), and an FAQ for common issues. Also document the **ELFIN DSL** usage (with a section describing how to write an ELFIN file and what TORI does with it). Write a section on **Troubleshooting** (like if suggestions aren’t appearing, check that agents are enabled; if performance is slow, try toggling X, etc.). Ensure the tone is accessible to the target users (developers, and possibly technical leads). Simultaneously, update the internal **Developer Documentation** – especially details of the ALAN core, so future maintainers (or open-source contributors, if it goes that way) can understand the complex parts (phase sync, spectral). This likely means adding comments in code, and maybe a short design doc summary to accompany the code. As feature freeze is now, ensure any pending merge requests for minor fixes are in, and then cut a release candidate build for final testing.  
  **Checkpoint:** End of Day 15, all documentation for release is ready or in final draft form. We have a Release Candidate (RC1) build that’s feature-frozen. The team (and perhaps some test users) will rely on the user guide to use the product – a good test if the guide is clear. We’re confident that if someone new picks up TORI with this documentation, they can get started and understand the basics.
* **Day 16 (May 27, 2025)** – *Final Validation & Beta Release*  
  **Goal:** Validate the RC build against real-world use cases one last time, and if all is good, designate it as the beta/release version.  
  **Tasks:** Do a final full run-through of the user experience: from installation, to opening a project, to acting on suggestions, to using chat, to checking logs and settings. Ensure the audit log is indeed capturing events. Try it on a couple of different projects (maybe one open-source repo, one internal project) to see how it behaves with code we didn’t specifically tailor for. If any critical issues appear, fix them (only critical fixes at this stage). Once satisfied, prepare the build artifacts for release: e.g., compile the Electron app to an installer for Windows/Linux/macOS as needed. Perform hash checks, virus scans, etc., as part of release prep. Label the build version (v1.0.0-beta perhaps). If there’s time, have a brief “beta release rehearsal” – writing the announcement or notes summarizing what’s in this version, known limitations (for transparency).  
  **Checkpoint:** End of Day 16, we have what we believe is the release build. It’s been tested in scenarios beyond our unit tests (some integration tests with sample user behaviors). We have not found any showstopper issues; any that were found are fixed or documented with workarounds if minor. The product is ready to deliver to the initial set of users (which could be internal stakeholders or a closed beta group).
* \*\*Day
* **Day 17 (May 28, 2025)** – *Release Preparation & Packaging*  
  **Goal:** Finalize all packaging and distribution aspects for the upcoming release.  
  **Tasks:** Prepare the installer or distribution packages for TORI IDE (Thoughtboard) on all target platforms (e.g., create a Windows installer, a Mac app bundle, etc.). Verify that each package includes all necessary dependencies (the AI models, knowledge packs, etc.) and that installation is smooth. Double-check that the audit logs and configuration files go to appropriate paths (e.g., user’s AppData or ~/.tori). Create a **Release Notes** document highlighting new features, known issues, and usage tips. Coordinate with any internal stakeholders about the release schedule.  
  **Checkpoint:** By end of Day 17, installable packages for TORI IDE and TORI Chat are ready. We have tested installing and uninstalling them on fresh machines. The release notes and any internal briefing materials are prepared. We’re essentially ready to roll out the software pending final sign-off.
* **Day 18 (May 29, 2025)** – *Internal Launch & Training*  
  **Goal:** Conduct an internal launch of TORI with a controlled group of users and provide training.  
  **Tasks:** Host an internal presentation or demo for stakeholders and potential pilot users. Walk through the vision, demonstrate the product live (showing how Thoughtboard identifies a bug and how Chat can explain a piece of code). Distribute the user guide and have a Q&A session. Set up a feedback channel (like an internal forum or email) for pilot users to report issues or suggestions. This day also serves as a buffer in case any last-minute critical bug appears during the demo – we can address it immediately.  
  **Checkpoint:** End of Day 18, the pilot users have the software in hand and are excited about its capabilities. They know how to use it at a basic level thanks to the training. We gather their initial reactions (which seem positive, say, enthusiasm about seeing conceptual links in their code). The team is on standby to support them as they start using TORI in daily work.
* **Day 19 (May 30, 2025)** – *Feedback Analysis & Minor Fixes*  
  **Goal:** Collect and analyze initial feedback from pilot users, and address any minor issues that surface.  
  **Tasks:** Monitor the feedback channels. If a pilot user reports a bug (“the concept graph viewer crashes on my project”), attempt to reproduce and fix it promptly. If any usability issues are noted (e.g., “I didn’t realize I had to click the lightbulb for suggestions”), consider quick tweaks or clarifications to the UI or documentation. Since we are technically post-release internally, limit changes to low-risk, high-value fixes (no new features). Possibly issue a quick patch update to pilot users if needed. Also, log enhancement ideas that are out of scope now but valuable for future versions (e.g., more agents for other domains).  
  **Checkpoint:** End of Day 19, any immediate post-launch fires are put out. Pilot users are not blocked by any major issues. The initial feedback is incorporated into our planning backlog for future improvements, but the current version remains stable.
* **Day 20 (May 31, 2025)** – *Go/No-Go Meeting & Launch Prep*  
  **Goal:** Decide on proceeding to broader release and prepare any external launch materials (if applicable).  
  **Tasks:** Hold a meeting with stakeholders to review pilot phase results. If the software has proven stable and valuable, get approval to move to the next stage (wider enterprise deployment or perhaps an external beta). If any serious issues arose, decide if they must be fixed before wider release or if documentation/training can mitigate them. Assuming a “Go” decision, prepare the external-facing materials: a polished announcement highlighting TORI’s strengths (e.g., a blog post or press release draft emphasizing “first cognitive OS IDE with explainable AI” etc.), and any marketing collateral like screenshots or short demo videos. Ensure legal compliance for release (open-source licenses in order, disclaimers included).  
  **Checkpoint:** End of Day 20, we have a green light to launch TORI to its intended wider audience. All necessary materials and logistical considerations for the launch day are ready. We’re essentially counting down to introducing TORI beyond the pilot group.
* **Day 21 (June 1, 2025)** – *Launch & Retrospective*  
  **Goal:** Officially launch TORI IDE and TORI Chat v1.0 and conduct a team retrospective on the sprint.  
  **Tasks:** Flip the switch on distribution – e.g., publish download links internally or externally, send out the announcement communications. The team will be actively monitoring for any issues as new users onboard (ready to provide support). Also, hold a **sprint retrospective** meeting: discuss what went well during the 21-day execution, what could be improved in our process, and gather lessons learned for the next development cycle. Acknowledge the team’s accomplishment (perhaps a small celebration for delivering this ambitious project on schedule!).  
  **Checkpoint:** By the end of Day 21, TORI IDE and ALAN 2.x (with TORI Chat) are officially released to their initial audience. The development sprint concludes with a functioning, innovative product in users’ hands. The team has documented insights to guide future sprints. We have successfully turned the TORI vision into a reality, setting the stage for further evolution.

**SWOT Analysis**

After completing the sprint and preparing for launch, we revisit TORI’s **Strengths, Weaknesses, Opportunities, and Threats** in the current context (mid-2025):

* **Strengths:** TORI’s core strength is its fundamentally different approach to AI assistance. Instead of relying solely on black-box predictions, TORI builds a *conceptual model* of the code and conversation, enabling **deep semantic understanding**. This yields more insightful and context-aware assistance than pattern-matching tools. Moreover, TORI’s design is **transparent and explainable** – every suggestion can be traced to a reasoning path in the concept graph, addressing a key demand in industry for explainable AI solutio[ibm.com](https://www.ibm.com/think/topics/black-box-ai#:~:text=White%20box%20AI%2C%20also%20called,Users)0】. The system’s **phase-synchronized reasoning** provides deterministic, reproducible results, which is a boon for enterprise users who need consistent behavior. Another major strength is TORI’s **local-first architecture**: all processing can be done on-premises, meaning companies can use TORI without exposing proprietary code or data externally. This privacy-preserving design is a differentiator, as it sidesteps cloud compliance issues and drastically reduces security concerns. Additionally, TORI’s multi-agent modularity makes it *extensible* – new capabilities can be added as separate agents without overhauling the core. Finally, TORI effectively combines development tooling with AI assistance; it’s a unified platform, which means users don’t juggle separate tools for coding, debugging, and asking questions – boosting productivity by keeping the workflow in one place.
* **Weaknesses:** TORI is an early-stage product and faces several limitations. The most immediate weakness is the **unproven nature** of its approach – while we’ve tested it on pilot projects, it lacks extensive real-world validation across diverse codebases. Its novel algorithms (oscillator networks, spectral analysis) may have edge cases we haven’t encountered; in contrast, traditional tools are very battle-tested. In terms of product maturity, TORI IDE currently lacks many of the ecosystem conveniences that established IDEs have (e.g., a rich plugin ecosystem, ultra-refined UI elements). We support only a handful of programming languages robustly (Python and ELFIN fully, others like C++ in a basic way), so users working in other languages might find TORI less helpful – this **limited language support** could narrow our initial user base and give an impression of incompletenefile-bnjjcfgff6iusnbzdwqayo2】. Performance, while much improved, could become an issue on extremely large projects – our advanced analysis might be slower than simpler tools for codebases with tens of thousands of files (something we will need to optimize as we scale). Another weakness is **user adoption risk**: developers have ingrained workflows, and some might be resistant to adopting a new IDE, especially one that introduces a radically different paradigm of AI involvement. Convincing them to trust TORI’s suggestions (and not see it as overly intrusive or gimmicky) will require education and gradual proof of value. Lastly, as a small team/new entrant, we have limited resources for support – any bugs or shortcomings early on could sour user perception if not addressed rapidly, so we must be very responsive, which can be challenging.
* **Opportunities:** The environment in 2025 is ripe for TORI’s unique value proposition. **Enterprise demand for AI** that is both powerful and private is high – many companies hesitate to use cloud-based AI coding tools due to intellectual property concerns. TORI can fill that gap by offering a **secure, on-prem AI assistant** that integrates with their internal knowledge (via concept packs) and doesn’t leak data. This is an opportunity to become the AI solution of choice for industries like finance, healthcare, or defense, where data control is non-negotiable. There is also a broader industry trend toward **augmented developer tooling**; TORI’s emphasis on explainability and determinism could position it as the *trusted* alternative to tools like GitHub Copilot (which sometimes outputs code without rationale). We can capitalize on growing awareness of AI’s “Clever Hans” problems (when models give the right answer for the wrong reasons) by showing how TORI avoids that and provides proof for its answe[ibm.com](https://www.ibm.com/think/topics/black-box-ai#:~:text=Reduced%20trust%20in%20model%20outputs)8】. Another opportunity is to extend TORI’s concept-based approach beyond code – for example, into data science notebooks, or integration with design tools – essentially, TORI could evolve into a general “cognitive assistant” platform. The multi-agent architecture means we could relatively easily add specialization for other domains (documentation writing, requirement analysis, etc.), expanding our market reach. Additionally, there is potential for **community and open-source contributions**: by sharing some of TORI’s DSL (ELFIN) or concept libraries, we could build a community that contributes domain packs (imagine domain experts encoding best practices in ELFIN for others to use). Lastly, if TORI demonstrates success, there is an opportunity for partnerships with larger IDE vendors or cloud providers – either integrating TORI’s technology into their offerings or collaborating to offer a hybrid solution (with TORI providing on-prem cognition and a partner providing cloud-scale resources if needed).
* **Threats:** The software development tooling market is highly competitive and is seeing rapid innovation in AI. A key threat is that **incumbent IDE vendors and big tech companies** will quickly advance their own AI-assisted tools. For example, Microsoft’s GitHub Copilot and Visual Studio integration is continuously improving, and JetBrains (IntelliJ) is introducing AI features – these players have enormous user bases and resources. If they solve the explainability or on-premise issues before TORI gains traction, TORI’s unique selling points could be diminished. We already see early moves in this direction, with competitors adding features to inspect and constrain AI suggestions. Another threat is **user skepticism or resistance** to TORI’s novel concepts. Developers might trust traditional linters or simpler AI assistants more because TORI’s internal workings (oscillator phases, spectral modes) are new and might seem arcane. We have to prove that these concepts genuinely translate to better results; otherwise users may dismiss TORI as over-engineered. There’s also the risk of **technical adoption barriers**: some IT departments might be slow to approve a new tool that has so many components (they might worry “Is it stable? Is it secure?” – thorough testing and certification may be demanded). Additionally, as TORI handles critical knowledge, any mistake (like an incorrect refactoring suggestion that introduces a bug) could undermine confidence – while our aim is to minimize such errors via careful design, no system is perfect. Larger competitors might also leverage their ecosystems to lock users in (for instance, if a competitor’s AI integrates seamlessly with their cloud devops pipeline, that convenience might outweigh TORI’s benefits for some teams). Finally, from a business perspective, if TORI gains attention, others may attempt to emulate our conceptual approach (we must continue innovating and possibly secure intellectual property, though patents in this space are tricky). In summary, the threats are the fast-moving competitive landscape and the need to win hearts and minds in a conservative segment of users.

**Conclusion:** Over the 21-day master plan, we have brought the TORI vision to life – creating a **cognitive operating system** that turns development and chat interfaces into truly intelligent, collaborative experiences. The resulting TORI IDE and TORI Chat (powered by the ALAN 2.x engine) demonstrate the feasibility and value of concept-oriented AI: they can understand code at a deeper level, explain their reasoning, and adapt to user needs, all while keeping the user in control. Initial testing indicates that TORI can catch subtle issues and provide insights that traditional tools miss, validating our paradigm shift towards *“concepts over tokens, memory over stateless prediction.”*

As we launch TORI to a broader audience, we will closely support our users and incorporate their feedback. Our SWOT analysis shows we are entering the market with a strong, differentiated solution, though we remain mindful of the challenges ahead. By maintaining our focus on technical rigor (grounded in proven science like Koopman theory and Lyapunov stability) and user-centric design (transparency, privacy, integration into workflows), TORI is well-positioned to carve out a niche of loyal users who need trustworthy AI assistance.

In the coming months, completing any partially implemented components and broadening language support will be priorities before resuming rapid feature development. With the foundation built in this master plan, those next steps will be on solid ground. TORI aims to herald a new era of developer tools – one where our IDE is not just a static environment, but a **cognitive partner** that evolves with us, ultimately making software development and knowledge work more intuitive, reliable, and powerful than ever before.